Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

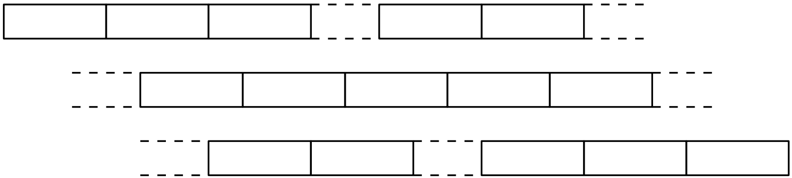
Dominique Poulalhon
`dominique.poulalhon@irif.fr`

Université de Paris

L2 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2020-2021

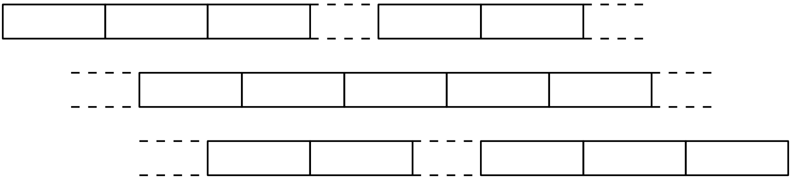
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



Rappel – principe du hachage

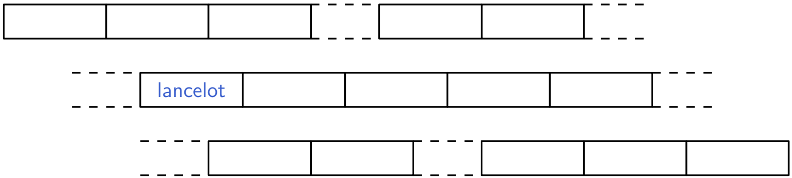
- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



h(lancelot) $=12$

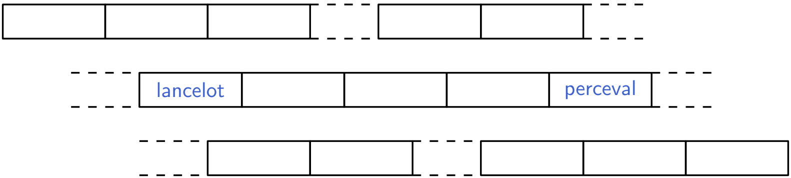
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



Rappel – principe du hachage

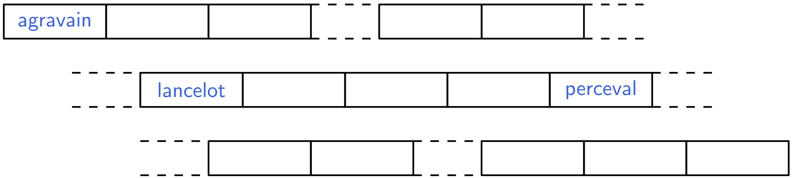
- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



$$\mathsf{h}(\mathsf{perceval}) = 16$$

Rappel – principe du hachage

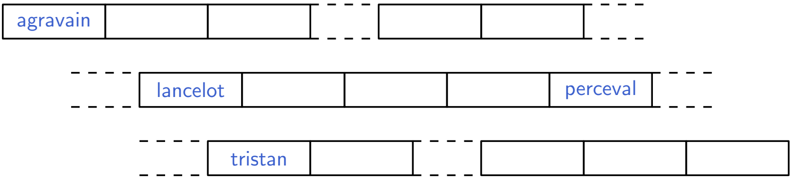
- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



h(agravain) $= 1$

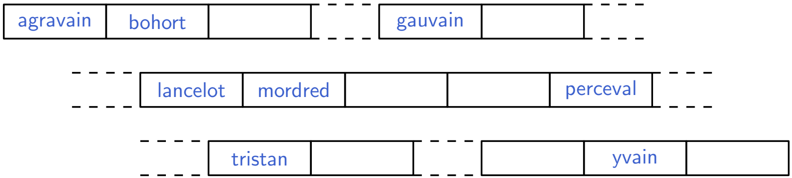
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



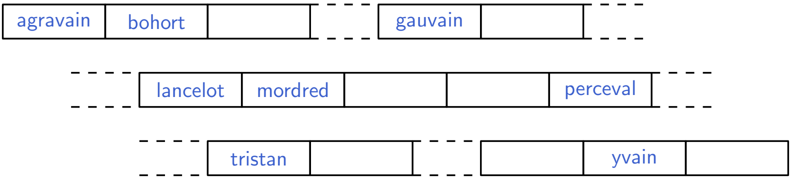
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `max`
- transformer n'importe quelle clé en entier plus petit que `max` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



$$\mathrm{h}(\mathrm{leodagan}) = 12 = \mathrm{h}(\mathrm{lancelot})$$

Le hachage

III. Résolution des collisions par sondage ou hachage « par adressage ouvert »

Résolution par sondage, ou par adressage ouvert (ou « hachage fermé » (*sic*))

Principe : utiliser directement la table [1] pour stocker les données :
si une cellule est occupée, essayer ailleurs !

1. couramment appelé « *espace d'adressage* », d'où l'appellation courante « par adressage ouvert » ; mais on peut au contraire considérer que le fait de se cantonner à l'espace d'adressage est une limite, d'où l'appellation « hachage fermé »... vu l'ambiguïté des terminologies « hachage ouvert » *vs* « hachage fermé », je déconseille fortement leur usage.

Résolution par sondage, ou par adressage ouvert (ou « hachage fermé » (*sic*))

Principe : utiliser directement la table [1] pour stocker les données :
si une cellule est occupée, essayer ailleurs !

Problème : comment retrouver ensuite cet « ailleurs » ?

1. couramment appelé « *espace d'adressage* », d'où l'appellation courante « par adressage ouvert » ; mais on peut au contraire considérer que le fait de se cantonner à l'espace d'adressage est une limite, d'où l'appellation « hachage fermé »... vu l'ambiguïté des terminologies « hachage ouvert » *vs* « hachage fermé », je déconseille fortement leur usage.

Résolution par sondage, ou par adressage ouvert (ou « hachage fermé » (*sic*))

Principe : utiliser directement la table [1] pour stocker les données :
si une cellule est occupée, essayer ailleurs !

Problème : comment retrouver ensuite cet « ailleurs » ?

si `T[h(cle)]` est occupée, *sonder* successivement d'autres cases jusqu'à en trouver une libre : pour la clé k, au i^e essai, on teste la case d'indice $h(k, i)$

L'exemple le plus simple est le sondage linéaire : si `T[h(cle)]` est occupée, tester successivement `T[h(cle) + 1]`, `T[h(cle) + 2]`, *etc.* (circulairement, en repartant au début de la table si toutes les cases au-delà de `T[h(cle)]` sont occupées).

1. couramment appelé « *espace d'adressage* », d'où l'appellation courante « par adressage ouvert » ; mais on peut au contraire considérer que le fait de se cantonner à l'espace d'adressage est une limite, d'où l'appellation « hachage fermé »... vu l'ambiguïté des terminologies « hachage ouvert » *vs* « hachage fermé », je déconseille fortement leur usage.

Propriétés du hachage avec résolution des collisions par adressage ouvert

Lemme

Le taux de remplissage α d'une table à adressage ouvert est au plus 1.

(forcément, puisque chaque case accueille au plus un élément)

Lemme

Pour tester toutes les cases sans redite, il faut que, pour chaque clé k, la fonction $i \mapsto h(k, i)$ soit une ***permutation****.*

(Dans le cas contraire, les sondages pourraient échouer à trouver une case libre bien qu'il en reste)

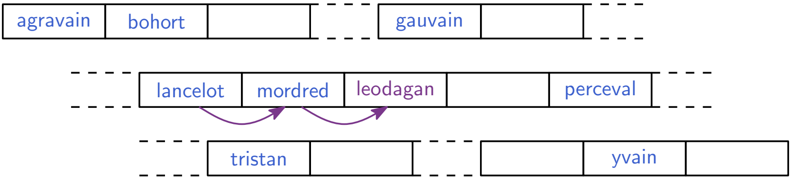
C'est bien le cas pour le sondage linéaire.

Hachage (par sondage) linéaire

Si `T[h(cle)]` est occupée, tester itérativement `T[h(cle) + 1]`, `T[h(cle) + 2]`, *etc.*
C'est-à-dire que le i^e sondage va tester la case d'indice :

$$h(k, i) = (h(k) + i) \mod m$$

Exemple :



$$h(\text{leodagan}) = 12 = h(\text{lancelot})$$

Cela peut s'écrire :

```
# version "dictionnaire" ie couples (cle, valeur)
def ajouter(table, cle, valeur) :
  for i in range(h(cle), len(table)) :
    if table[i] == None or table[i][0] == cle : break
  else : # si on atteint la fin, on recommence au début
    for i in range(h(cle)) :
      if table[i] == None or table[i][0] == cle : break
  table[i] = (cle, valeur)
```

ou de manière équivalente :

```
def ajouter(table, cle, valeur) :
  k, m = h(cle), len(table)
  for i in range(m) :
    if table[(k+i)%m] == None or table[(k+i)%m][0] == cle :
      break
  table[(k+i)%m] = (cle, valeur)
```

Première version des autres opérations :

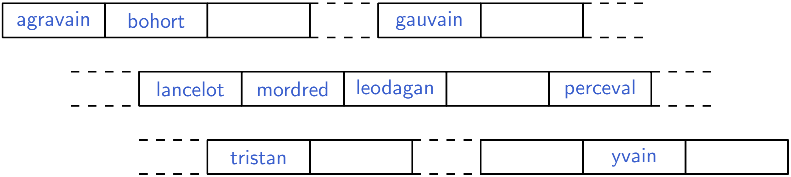
attention, tel quel c'est faux ! !

```
def chercher(table, cle) :
  for i in range(h(cle), len(table)) :
    if table[i] == None : return None
    # on s'arrête à la première case vide trouvée :
    # échec de la recherche
    if table[i][0] == cle : return table[i][1]
  # le cas échéant, on recommence au début
  ...

def supprimer(table, cle) : ## (attention, tel quel c'est faux)
  for i in range(h(cle), len(table)) :
    if table[i] == None : return
    if table[i][0] == cle :
      table[i] = None
      return
  # le cas échéant, on recommence au début
  ...
```

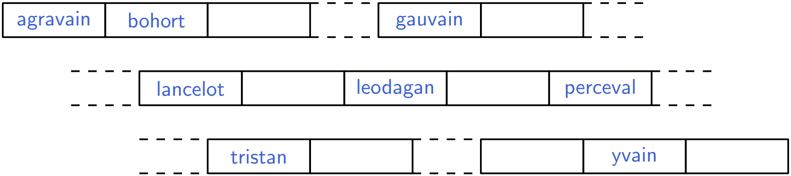
Hachage (par sondage) linéaire

Pourquoi est-ce faux ? Eh bien par exemple, si on supprime `mordred` avant de chercher `leodagan`...



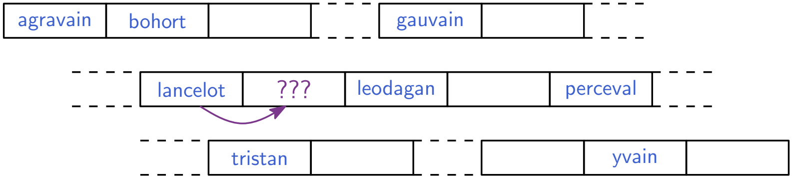
Hachage (par sondage) linéaire

Pourquoi est-ce faux ? Eh bien par exemple, si on supprime `mordred` avant de chercher `leodagan`…



Hachage (par sondage) linéaire

Pourquoi est-ce faux ? Eh bien par exemple, si on supprime `mordred` avant de chercher `leodagan`...



GLOUPS !!!

Hachage (par sondage) linéaire

Lorsqu'un élément est retiré de la table, il est important de *laisser une marque* pour que les recherches ultérieures tiennent compte du fait que la case a un jour été occupée (ce qui a pu provoquer la poursuite des sondages lors d'une insertion).

```
def supprimer(table, cle) :
  for i in range(h(cle), len(table)) :
    if table[i] == None : return
    if table[i][0] == cle :
      # la case n'est pas vidée, mais libérée
      table[i][1] = None
      return
  # le cas échéant, on recommence au début
  ...
```

(j'ai choisi de coder les cases vraiment vides par `None`, et les cases libérées par (`cle`, `None`) où `cle` est la clé ayant un jour occupé la case. Tout autre type de marque peut faire l'affaire, mais il faut différencier les cases vides depuis toujours et les cases libérées)

Cela modifie donc un peu la recherche :

```
def chercher(table, cle) :
  for i in range(h(cle), len(table)) :
    if table[i] == None : return None
    if table[i][0] == cle : return table[i][1]
  # le cas échéant, on recommence au début
  ...
```

mais aussi l'ajout :

```
def ajouter(table, cle, valeur) :
  for i in range(h(cle), len(table)) :
    if table[i] == None or table[i][1] == None :
    # pas tout à fait suffisant (pb si cle est déjà dans table)
      table[i] = (cle, valeur)
      return
  # le cas échéant, on recommence au début
  ...
```

Complexité de la résolution par adressage ouvert

Hypothèse de hachage uniforme (forte) :

pour une clé aléatoire, chacune des $m!$ permutations a la même probabilité $\frac{1}{m!}$ d'apparaître comme suite de sondages.

Théorème

*dans une table à adressage ouvert, taux de remplissage $\alpha < 1$, et hachage supposé uniforme, le nombre moyen de sondages pour une recherche **infructueuse** est au plus $\frac{1}{1-\alpha}$.*

(preuve à suivre)

Théorème (*admis*)

*sous les mêmes hypothèses, le nombre moyen de sondages pour une recherche **réussie** est au plus $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.*

Donc sous ces hypothèses, à taux de remplissage fixe, les accès ont un coût moyen constant.

Problème

Le sondage linéaire permet ***seulement*** m ***séquences de sondage différentes***... donc on est très très loin de l'hypothèse de hachage uniforme !

Constatation

À l'expérience, on observe un phénomène de ***clusterisation*** qui diminue rapidement les performances du hachage : les éléments s'agglutinent en gros amas, rendant les accès dans les zones concernées très lents puisqu'ils nécessitent souvent de parcourir une grande partie de l'amas.

Idée

Utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Double hachage

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k, i) = h_1(k) + i \cdot h_2(k) \mod m$$

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Lemme

la suite $i \mapsto h(k,i)$ *est une permutation si et seulement si :*

$h_2(k)$ *est premier avec* m

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Lemme

la suite $i \mapsto h(k,i)$ *est une permutation si et seulement si :*

$h_2(k)$ *est premier avec* m

Comment assurer cette propriété ?

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Lemme

la suite $i \mapsto h(k,i)$ *est une permutation si et seulement si :*

$h_2(k)$ *est premier avec* m

Comment assurer cette propriété ?

- avec m premier et $h_2(k) < m$ quelconque

  — *parfait si* n *est connu à l'avance (et donc* m *aussi)*

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Lemme

la suite $i \mapsto h(k,i)$ *est une permutation si et seulement si :*

$h_2(k)$ *est premier avec* m

Comment assurer cette propriété ?

- avec m premier et $h_2(k) < m$ quelconque

  — *parfait si* n *est connu à l'avance (et donc* m *aussi)*
- avec $m = 2^p$ et $h_2(k)$ impair

  — *si des redimensionnements sont nécessaires*

utiliser ***deux*** fonctions de hachage h_1 et h_2, et sonder successivement

$$h(k,i) = h_1(k) + i \cdot h_2(k) \mod m$$

Lemme

la suite $i \mapsto h(k,i)$ est une permutation si et seulement si :

$h_2(k)$ est premier avec m

Comment assurer cette propriété ?

- avec m premier et $h_2(k) < m$ quelconque
  — *parfait si n est connu à l'avance (et donc m aussi)*
- avec $m = 2^p$ et $h_2(k)$ impair
  — *si des redimensionnements sont nécessaires*

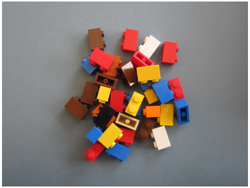
on obtient alors $\Theta(m^2)$ ***séquences*** de sondage...
c'est encore loin de $m!$, mais nettement meilleur que m

Le hachage

IV. Qu'est-ce qu'une bonne fonction de hachage ?

Qu'est-ce-qu'une bonne fonction de hachage ?

Si vous êtes adeptes de Lego, vous avez sûrement remarqué que les sachets ne ressemblent pas à ceux-là :



mais plutôt à ça :

Pourquoi ? C'est assez contre-intuitif peut-être, mais essayez donc de chercher une pièce orange 2x1 dans chaque paquet par exemple... Trouver rapidement le bon sachet n'est pas la seule chose importante, il faut aussi pouvoir le fouiller vite, et pour ça, il vaut mieux que les pièces « hachées » dans le même sachet soient le plus dissemblables possible...

Qu'est-ce-qu'une bonne fonction de hachage ?

pour une bonne efficacité (en temps), il faut :

- trouver rapidement la bonne boîte ;
- fouiller rapidement la boîte ;

et bien sûr, il faut éviter le gâchis en espace.

la fonction de hachage doit donc

- être facile à calculer ;
- idéalement, être sans collision – c'est impossible, mais à défaut, les éléments concernés doivent être facilement discernables ;
- remplir la table *uniformément* : éviter d'avoir de grandes zones vides et de grandes zones pleines ;
- pour cela, il faut disperser les données similaires.

pour répondre, il faut savoir ***à quoi ressemblent les données***

Exemple

toutes les chaînes de moins de 25 caractères ne peuvent pas être des entrées d'un dictionnaire (un vrai, genre le Larousse).

Exemple

un compilateur maintient une table des symboles référençant les identificateurs du programme en cours de compilation
or les programmeurs ont tendance à utiliser des identificateurs qui se ressemblent : `tmp`, `tmp1`, `tmp2`...

Par exemple, la fonction `h` que j'ai utilisée pour les chevaliers de la table ronde est très mauvaise : (dans une langue donnée), certaines lettres ont beaucoup plus de chance d'être l'initiale d'un nom que d'autres, et l'hypothèse de hachage uniforme simple n'est donc pas satisfaite.

en général, les données ne sont pas réparties uniformément dans l'univers des données possibles.

une fonction de hachage (primaire) doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

Construction de fonctions de hachage

une fonction de hachage (primaire) doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

deux étapes

- transformer toute donnée en valeur numérique (entière) : cette étape est spécifique aux données considérées
- hacher les nombres : c'est cette étape qui va assurer la dispersion

Construction de fonctions de hachage

une fonction de hachage (primaire) doit

- être facile à calculer
- remplir la table *uniformément*, donc *disperser les données similaires*

deux étapes

- *transformer toute donnée en valeur numérique (entière)* : cette étape est spécifique aux données considérées
- hacher les nombres : c'est cette étape qui va assurer la dispersion

pour du texte par exemple, le plus simple : remplacer chaque caractère par son code ASCII, et considérer le texte $t_0 \ldots t_\ell$ comme l'entier

$$h(t_0 t_1 \ldots t_\ell) = t_0 b^\ell + t_1 b^{\ell-1} + \cdots + t_{\ell-1} b + t_\ell$$

(en Java : $b = 31$)

deux étapes

- transformer toute donnée en valeur numérique (entière) : cette étape est spécifique aux données considérées
- ***hacher les nombres*** : c'est cette étape qui va assurer la dispersion

méthode *par division*

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

deux étapes

- transformer toute donnée en valeur numérique (entière) : cette étape est spécifique aux données considérées
- *hacher les nombres* : c'est cette étape qui va assurer la dispersion

méthode *par division*

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

- incontestablement simple à calculer...
- mais pas de dispersion des données similaires

$\Longrightarrow$ pas très adaptée comme fonction de hachage primaire
(mais tout à fait satisfaisante comme (base de) fonction de hachage secondaire)

deux étapes

- transformer toute donnée en valeur numérique (entière) : cette étape est spécifique aux données considérées
- ***hacher les nombres*** : c'est cette étape qui va assurer la dispersion

méthode *par division*

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

méthode *par multiplication*

$$h(x) = \lfloor m \times \{Ax\} \rfloor \qquad (\text{où } \{x\} = x - \lfloor x \rfloor)$$

- m a peu d'importance (par exemple une puissance de 2 ou un nombre premier conviennent)
- une bonne valeur (empirique) pour A est $\frac{\sqrt{5}-1}{2}$ (ou une approximation fractionnaire)

$\Longrightarrow$ très bien comme fonction de hachage primaire