Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

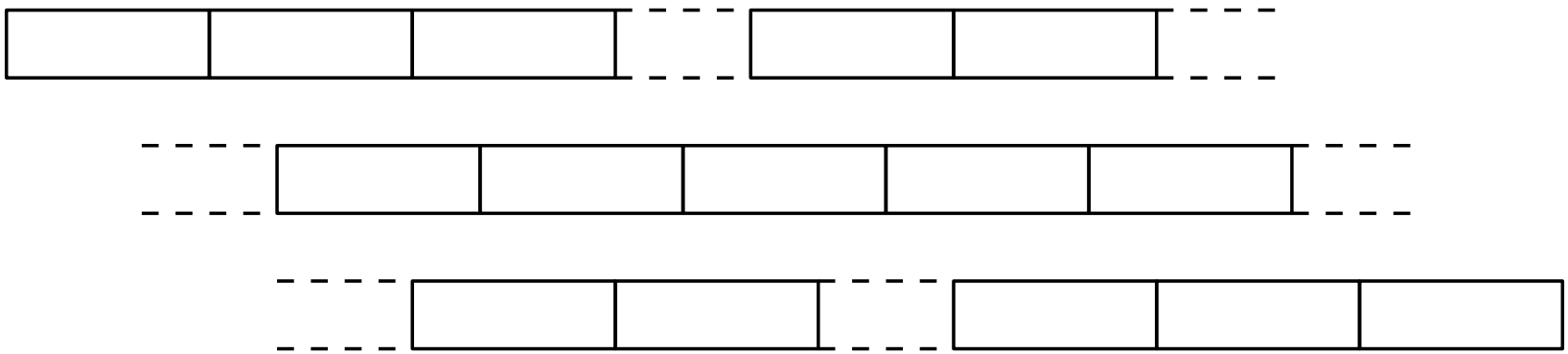
Université de Paris
L2 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2023-2024

Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` 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*)

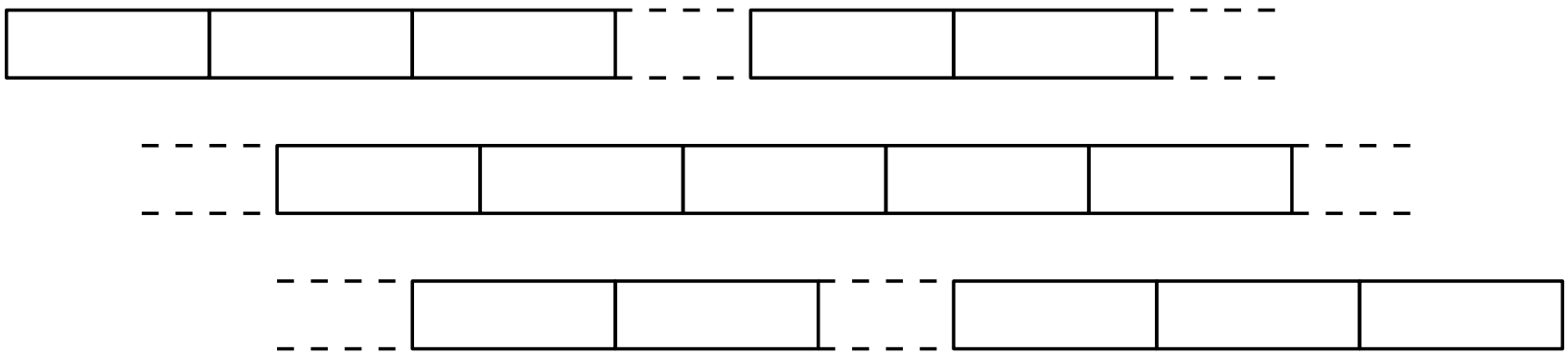
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` 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

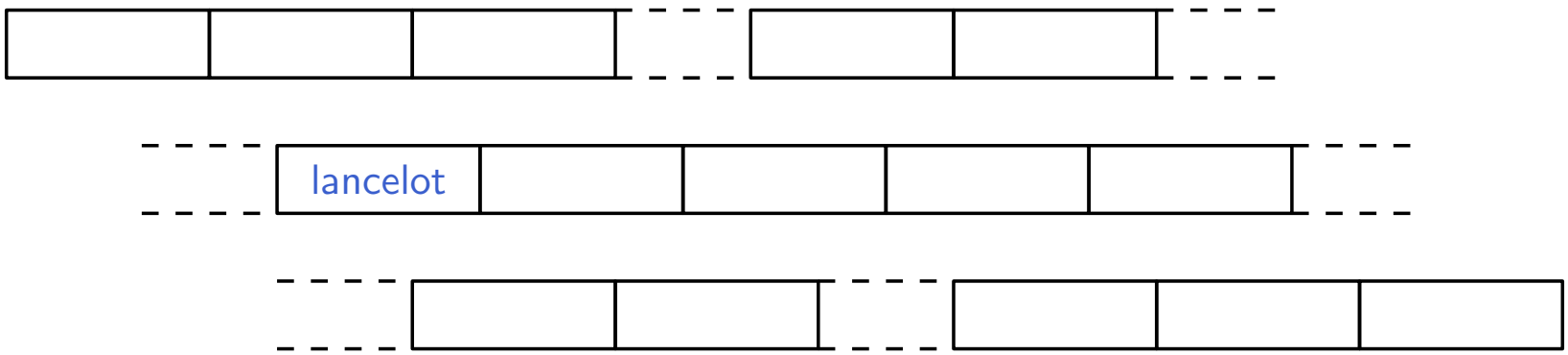
- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



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

Rappel – principe du hachage

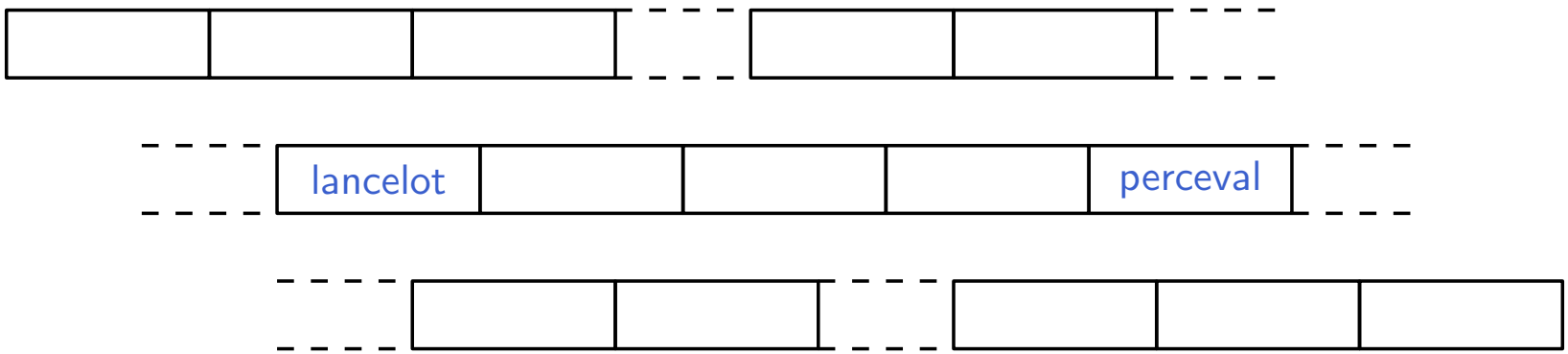
- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



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

Rappel – principe du hachage

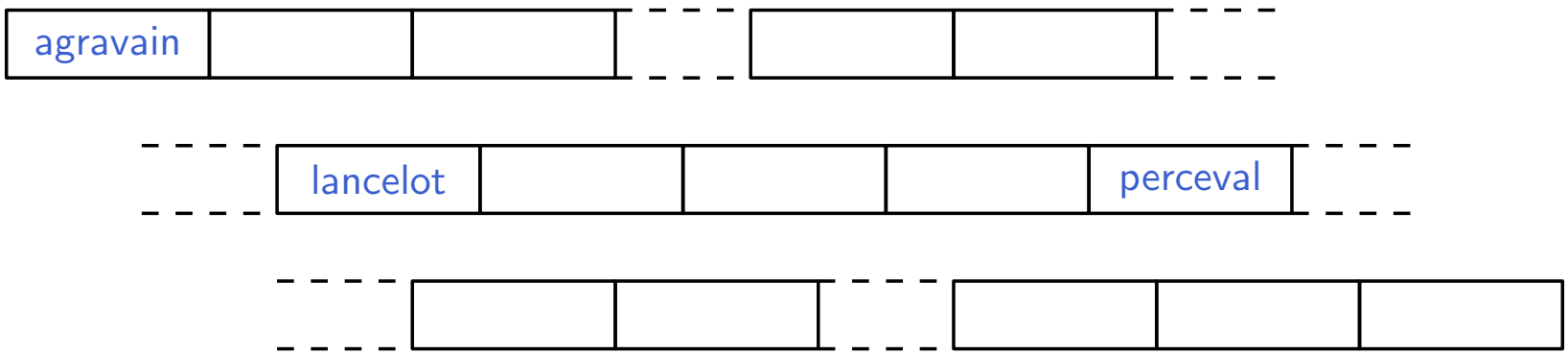
- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



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

Rappel – principe du hachage

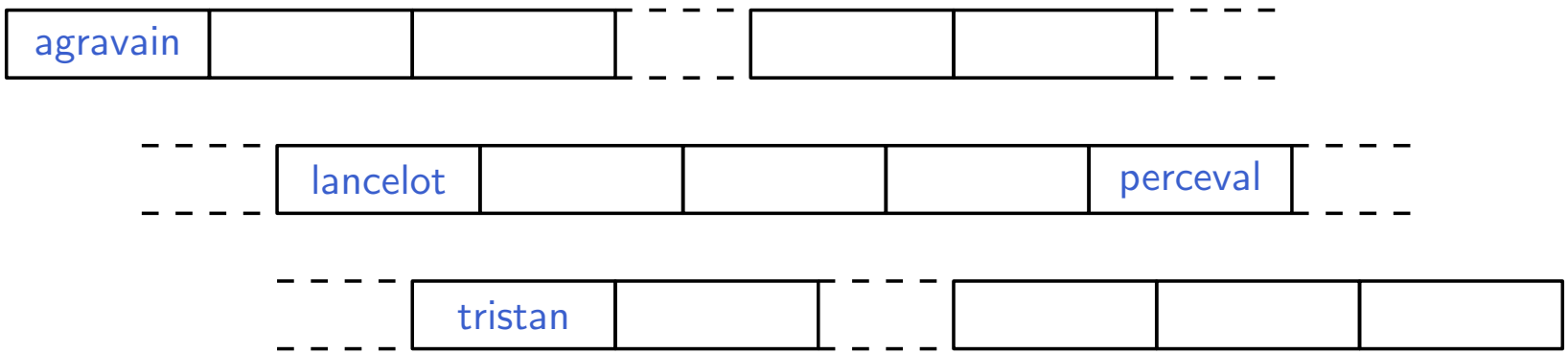
- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



$h(\text{agravain}) = 1$

Rappel – principe du hachage

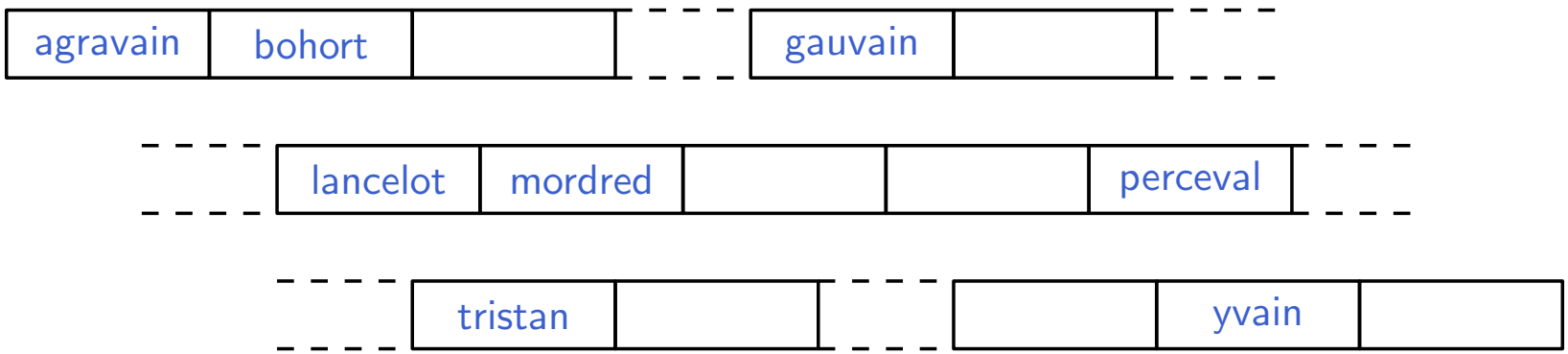
- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` à l'aide d'une fonction de hachage `h`
- stocker chaque élément `elt` dans la case `T[h(elt)]` (on parle de *boîte* ou *bucket*)



$h(\text{tristan}) = 16$

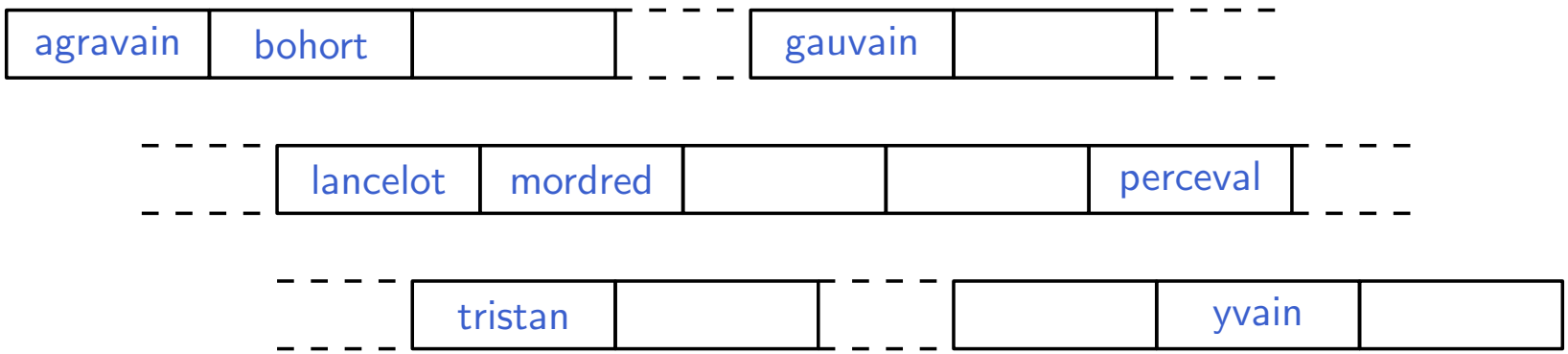
Rappel – principe du hachage

- allouer un (grand) tableau `T` de taille `m`
- transformer n'importe quelle clé en entier plus petit que `m` 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 `m`
- transformer n'importe quelle clé en entier plus petit que `m` 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*)



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

Problème : il peut se produire des ***collisions***

Rappel – hachage par chaînage

Principe : la table est un tableau de listes chaînées d'éléments

on note m la longueur du tableau, n le nombre d'éléments dans la table, $\alpha = \frac{n}{m}$ le taux de remplissage

Hypothèse de hachage uniforme ***simple*** **:** pour tout $i < m$, une clé aléatoire est hachée vers la case i avec proba $\frac{1}{m}$

Théorème

sous l'hypothèse de hachage uniforme simple, *le coût* ***moyen*** *d'une recherche est* $O(1 + \frac{n}{m})$.

Corollaire

si la longueur m *de la table est choisie supérieure à* n/α *pour un* α ***fixé***, *alors le coût* ***moyen*** *d'une recherche (ou d'un ajout, ou d'une suppression) est* $O(1 + \alpha) = \Theta(1)$.

Rappel – redimensionnement

Principe : si n n'est pas connu à l'avance, on fixe un taux de remplissage α à ***ne pas*** dépasser et on choisit m arbitrairement
lorsque le taux de remplissage effectif atteint α, on crée une nouvelle table de longueur $2m$ dans laquelle on transfère toutes les données

chaque redimensionnement a un coût linéaire (en la taille n atteinte au moment du redimensionnement)... mais a lieu après ***au moins*** n ajouts *(et peut-être aussi d'autres redimensionnements : il faut donc considérer le coût* ***cumulé*** *de ces redimensionnements – lui aussi linéaire en* n*)*

Théorème

si la répartition des éléments est uniforme *dans une table à taux de remplissage borné, le* ***coût moyen amorti*** *des accès (recherche, ajout, suppression) est* $\Theta(1)$.

Le hachage

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

Résolution des collisions 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 des collisions 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 des collisions 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 $h(k, i)$

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 des collisions 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.

Hachage (par sondage) linéaire

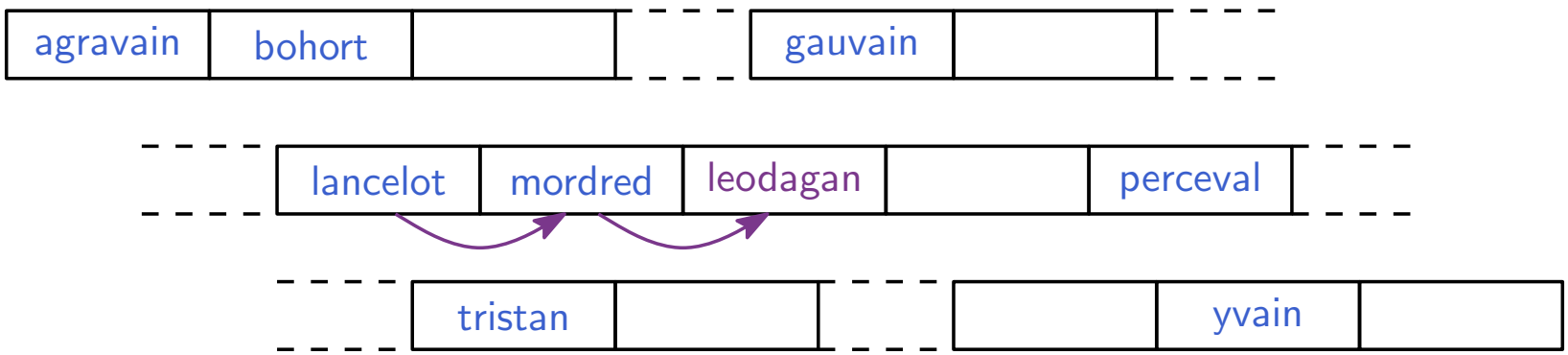
Si `T[h(cle)]` est occupée, tester itérativement

`T[h(cle) + 1]`, `T[h(cle) + 2]`, *etc.*

C'est-à-dire que le $(i+1)^e$ sondage va tester la case d'indice :

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

Exemple :



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

Hachage (par sondage) linéaire

```
# version "dictionnaire" ie couples (cle, valeur)
def ajouter(table, cle, valeur) :
  for i in range(h(cle), len(table)) : # parcours à partir de la case h(cle)
    if table[i] is None or table[i][0] == cle : break
  else : # si la fin est atteinte, reprise au début
    for i in range(h(cle)) :
      if table[i] is 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) # stockage pour éviter les recalculs multiples
  for i in range(m) :
    if table[(k+i)%m] is None or table[(k+i)%m][0] == cle : break
  table[(k+i)%m] = (cle, valeur)
```

ou encore, avec la fonction `chain` du module `itertools` :

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

Première version des autres opérations :

attention, tel quel c'est faux ! !

```
def chercher(table, cle) :
  k, m = h(cle), len(table)
  for i in itertools.chain(range(k, m), range(k)) :
    if table[i] is None : return None
    # arrêt à la première case vide trouvée : échec de la recherche
    if table[i][0] == cle : return table[i][1]

def supprimer(table, cle) : ## (attention, tel quel c'est faux)
  k, m = h(cle), len(table)
  for i in itertools.chain(range(k, m), range(k)) :
   if table[i] is None : return
   if table[i][0] == cle :
      table[i] = None
      return
```

Propriétés du hachage avec résolution des collisions par sondage

Lemme

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

Propriétés du hachage avec résolution des collisions par sondage

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)

Propriétés du hachage avec résolution des collisions par sondage

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

Il doit même être ***strictement inférieur à 1****.*

Propriétés du hachage avec résolution des collisions par sondage

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

*Il doit même être **strictement inférieur à 1**.*

*(il doit toujours rester au moins une case vide, car les cases vides sont les **témoins** des recherches infructueuses)*

Propriétés du hachage avec résolution des collisions par sondage

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

*Il doit même être **strictement inférieur à 1**.*

*(il doit toujours rester au moins une case vide, car les cases vides sont les **témoins** des recherches infructueuses)*

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**.*

Propriétés du hachage avec résolution des collisions par sondage

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

*Il doit même être **strictement inférieur à 1**.*

*(il doit toujours rester au moins une case vide, car les cases vides sont les **témoins** des recherches infructueuses)*

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)

Propriétés du hachage avec résolution des collisions par sondage

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

*Il doit même être **strictement inférieur à 1**.*

*(il doit toujours rester au moins une case vide, car les cases vides sont les **témoins** des recherches infructueuses)*

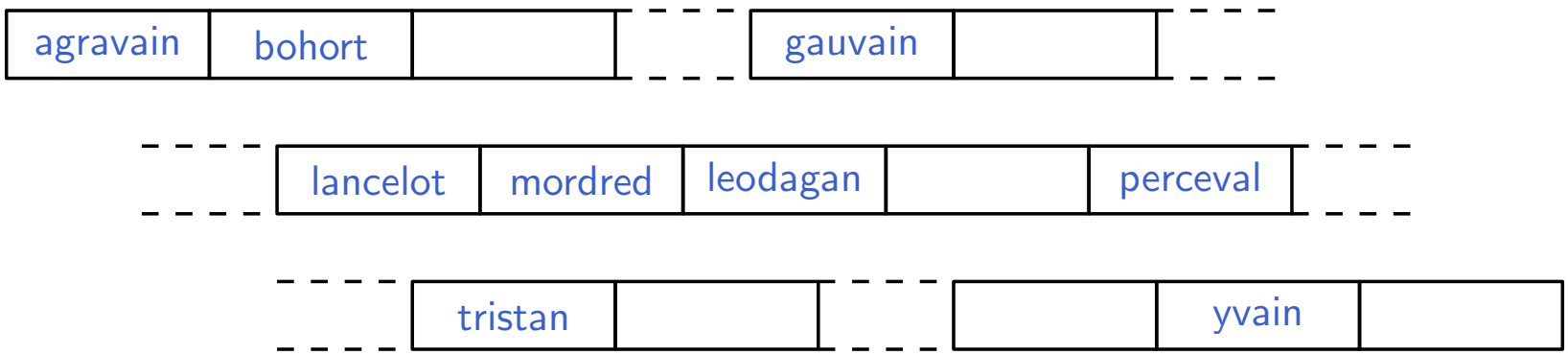
Lemme

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

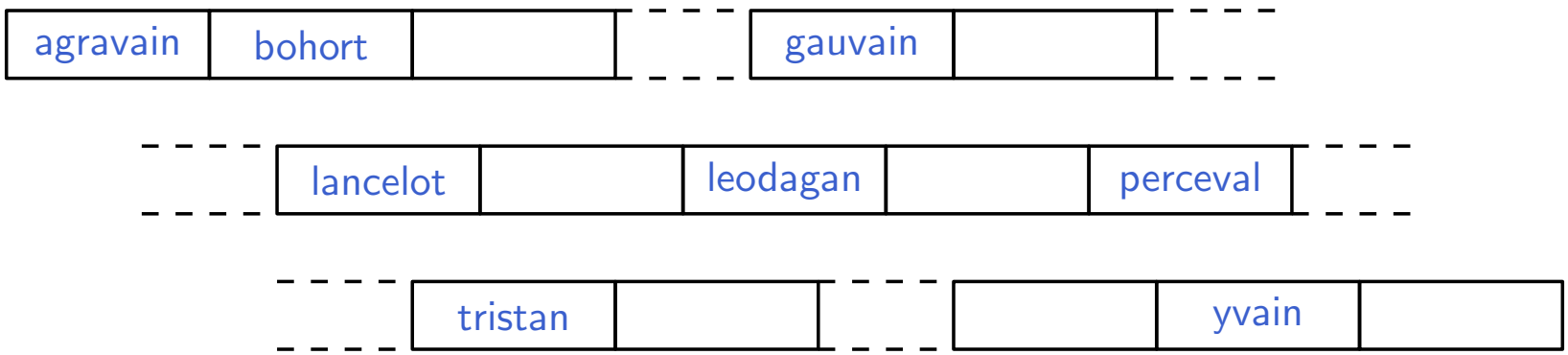
(dans le cas contraire, les sondages pourraient échouer à trouver une case libre bien qu'il en reste)

C'est bien le cas pour le sondage linéaire.

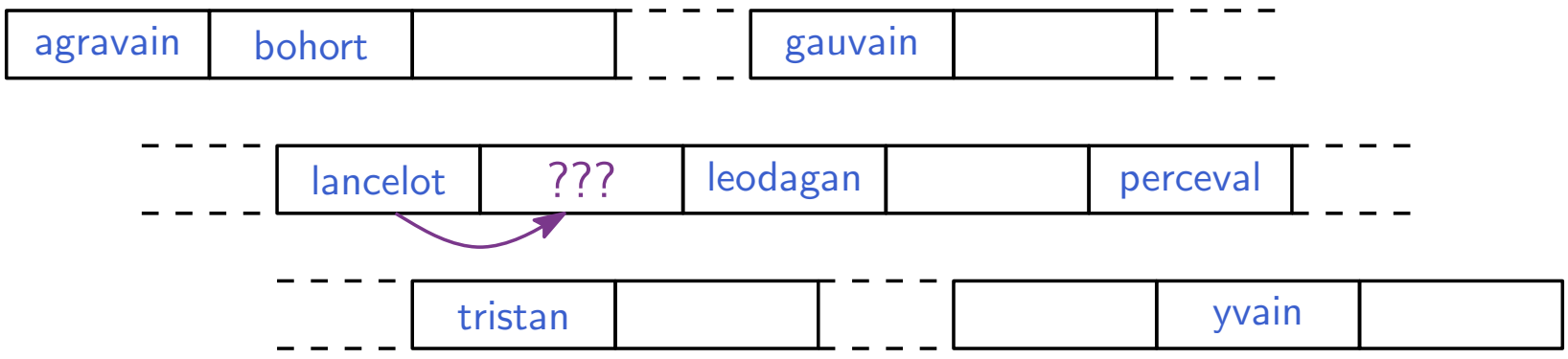
Subtilité très importante : on ne peut pas simplement effacer les éléments à supprimer ; par exemple, si on supprime `mordred` avant de chercher `leodagan`...



Subtilité très importante : on ne peut pas simplement effacer les éléments à supprimer ; par exemple, si on supprime `mordred` avant de chercher `leodagan`...



Subtilité très importante : on ne peut pas simplement effacer les éléments à supprimer ; par exemple, si on supprime `mordred` avant de chercher `leodagan`...



GLOUPS !!!

Lorsqu'un élément est retiré de la table, il est donc indispensable 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).

Lorsqu'un élément est retiré de la table, il est donc indispensable 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) :
  k, m = h(cle), len(table)
  for i in itertools.chain(range(k, m), range(k)) :
    if table[i] is None : return
    if table[i][0] == cle :
      # la case n'est pas vidée, mais libérée et marquée
      table[i][1] = None
      return
```

Lorsqu'un élément est retiré de la table, il est donc indispensable 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) :
  k, m = h(cle), len(table)
  for i in itertools.chain(range(k, m), range(k)) :
    if table[i] is None : return
    if table[i][0] == cle :
      # la case n'est pas vidée, mais libérée et marquée
      table[i][1] = None
      return
```

(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) :
  k, m = h(cle), len(table)
  for i in itertools.chain(range(k, m), range(k)) :
    if table[i] is None : return None
    if table[i][0] == cle and table[i][1] is not None :
      return table[i][1]
```

mais aussi l'ajout :

```
def ajouter(table, cle, valeur) :
  k, m, tmp = h(cle), len(table), None
  for i in itertools.chain(range(k, m), range(k)) :
    if table[i] is None : break
    if table[i][1] is None and tmp is None : tmp = i
  if tmp is not None : i = tmp
  table[i] = (cle, valeur)
```

En résumé :

En résumé :

- 3 types de cases : vides, occupées, marquées

En résumé :

- 3 types de cases : vides, occupées, marquées
- les cases vides servent de « barrières » lors des recherches infructueuses (et donc des insertions réussies et suppressions infructueuses)

En résumé :

- 3 types de cases : vides, occupées, marquées
- les cases vides servent de « barrières » lors des recherches infructueuses (et donc des insertions réussies et suppressions infructueuses)
- les cases marquées sont traitées comme des cases occupées lors des recherches, et comme des cases vides lors des insertions

En résumé :

- 3 types de cases : vides, occupées, marquées
- les cases vides servent de « barrières » lors des recherches infructueuses (et donc des insertions réussies et suppressions infructueuses)
- les cases marquées sont traitées comme des cases occupées lors des recherches, et comme des cases vides lors des insertions
- plus précisément, une insertion réussie se fait en deux étapes : sondage jusqu'à la première case vide, et insertion dans la première case marquée ou vide rencontrée

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.*

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 (*preuve à suivre*)

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 $\frac{1}{1-\alpha}$.*

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 (*preuve à suivre*)

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 $\frac{1}{1-\alpha}$.*

Théorème (*admis*)

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

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 (*preuve à suivre*)

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 $\frac{1}{1-\alpha}$.*

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 : dans une table à adressage ouvert, avec ***taux de remplissage maximal $\alpha < 1$ fixé***, et sous l'***hypothèse de hachage uniforme forte***, les accès ont un ***coût moyen amorti*** constant.

Complexité de la résolution par sondage

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* $\frac{1}{1-\alpha}$

Démonstration

Lors d'une recherche infructueuse, on sonde successivement des cases occupées avant de s'arrêter sur une case vide. Notons S le nombre de sondages nécessaires (qui est une variable aléatoire), et considérons la probabilité que $S > k$.

$S > 1$ signifie que la première case sondée est l'une des n cases occupées parmi m, donc $\mathbb{P}(S > 1) = \frac{n}{m}$.

$S > 2$ signifie que la première case sondée est occupée, et que la deuxième, distincte de la première, est l'une des $n-1$ autres cases occupées parmi $m-1$, donc $\mathbb{P}(S > 2) = \frac{n}{m} \cdot \frac{n-1}{m-1}$ d'après l'hypothèse d'uniformité.

Plus généralement, on obtient $\mathbb{P}(S > k) = \dfrac{n}{m} \cdot \dfrac{n-1}{m-1} \cdots \dfrac{n-k+1}{m-k+1}$.

On remarque que $\frac{n}{m} \geqslant \frac{n-i}{m-i}$ car $n(m-i) \geqslant m(n-i)$ dès que $n \leqslant m$, ce qui est le cas.

Donc $\mathbb{P}(S > k) \leqslant \alpha^k$.

Pour obtenir l'espérance (i.e. la valeur moyenne) de S, on se sert ensuite de la formule

$$\mathbb{E}(S) = \sum_{k=1}^{n} k \cdot \mathbb{P}(S = k) = \sum_{k=1}^{n} \mathbb{P}(S \geqslant k) = \sum_{k=0}^{n} \mathbb{P}(S > k),$$

qu'on peut majorer par la somme de la série géométrique $\displaystyle\sum_{k=0}^{\infty} \alpha^k = \frac{1}{1-\alpha}$.

Problème

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

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.

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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

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

(dit autrement : on part de $h_1(k)$, puis on fait des sauts de longueur $h_2(k)$)

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*

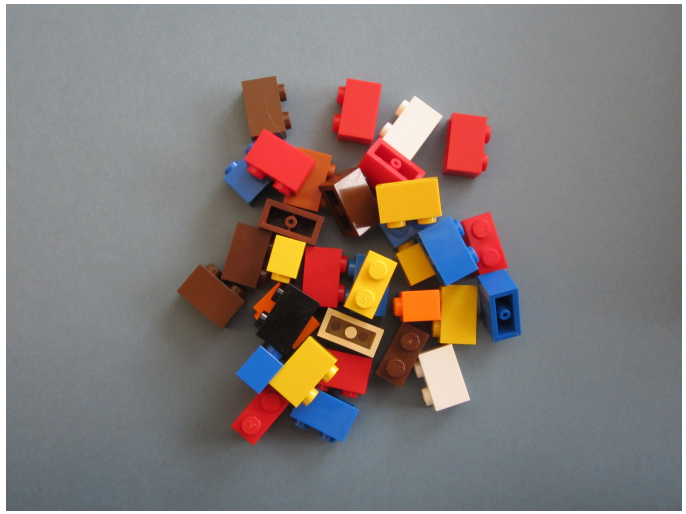
on obtient alors $\Theta(m^2)$ ***séquences*** de sondage...
c'est encore loin de $m!$, mais nettement meilleur que m

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 :

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 *(ou les boîtes à sonder successivement, par exemple les boîtes contiguës danns le cas du sondage linéaire)* ;

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.

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

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

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

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 (au sens usuel, genre le Larousse).

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

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 (au sens usuel, 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`...

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

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 (au sens usuel, 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.

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

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 (au sens usuel, 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.

Construction de fonctions de hachage

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 $t_0 \dots t_\ell$ comme l'entier

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

(en Java : $b = 31$)

Construction de fonctions de hachage

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)

Construction de fonctions de hachage

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