Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

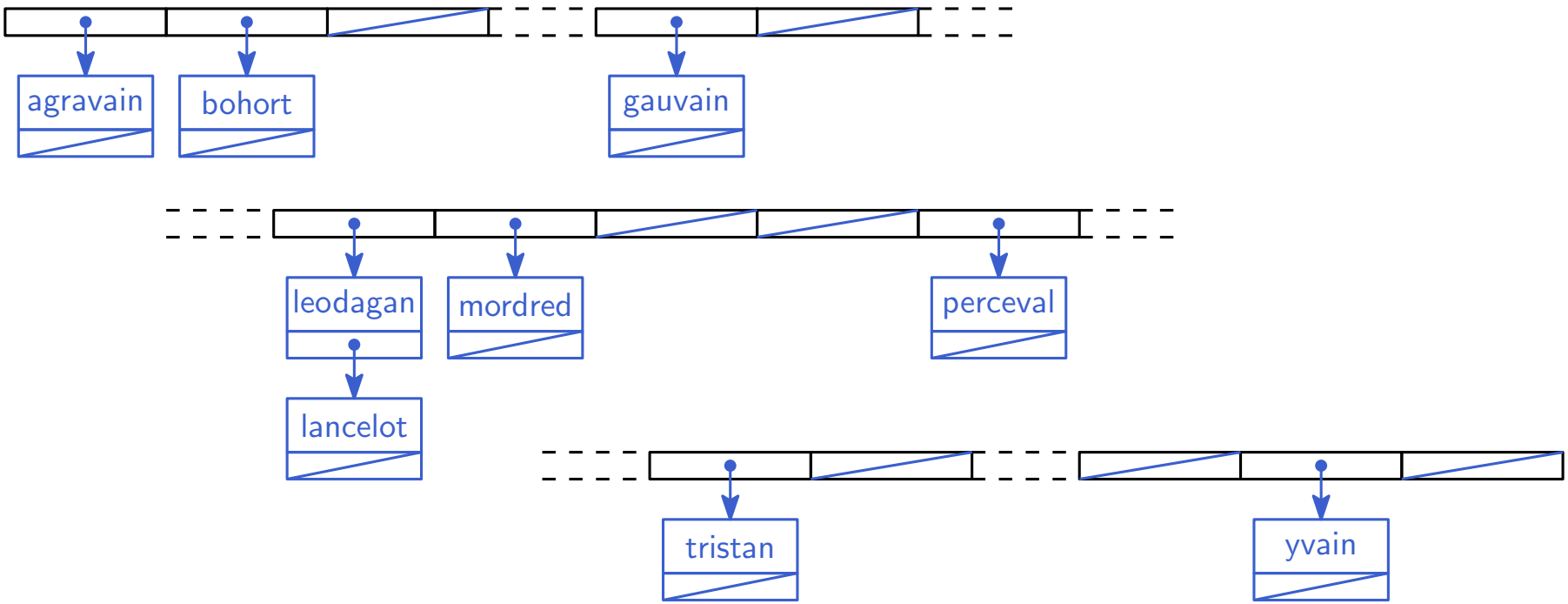
Le hachage

II. Résolution des collisions par chaînage

Résolution des collisions par chaînage (ou « hachage ouvert »)

Le principe est simple : pour pouvoir stocker plusieurs éléments dans la même case, il suffit d'utiliser un tableau de listes chaînées d'éléments.

Les éléments ne sont donc pas stockés directement dans la table, mais à l'extérieur, d'où la terminologie de « hachage ouvert » parfois utilisée. Je la déconseille fortement à cause de l'ambiguïté que vous constaterez lorsque nous parlerons de l'autre grande méthode de résolution des collisions.



Résolution des collisions par chaînage

Cela donne donc, avec des listes Python :

attention, c'est mieux mais c'est encore faux !!

```
def ajouter(table, elt) :
  table[h(elt)].append(elt)
  # quoi ?! sans même vérifier si elt est déjà dans table ???
  # admettons... mais nous avons déjà vu qu'il n'est pas très raisonnable
  # d'autoriser des doublons dans la représentation d'un ensemble, car cela
  # augmente sa complexité en espace, et complique la suppression

def supprimer(table, elt) :
  table[h(elt)].remove(elt) # !!! NON !!!
  # incohérent avec l'ajout ci-dessus, il faut supprimer toutes les occurrences

def chercher(table, elt) :
  return elt in table[h(elt)] # OK (une fois que supprimer() est corrigé)
```

Remarque : Python est ici assez inadapté pour écrire la fonction `supprimer` car il ne permet pas de manipuler finement le chaînage (d'ailleurs les listes ne sont en fait même pas des listes chaînées !). Faire des appels successifs à `remove` serait une catastrophe en terme de complexité, car chaque appel repartirait du début de la liste pour supprimer la première occurrence, d'où un coût cumulé quadratique et non linéaire. On peut utiliser `pop` en maintenant une variable pour l'indice de l'élément à supprimer, mais il faut faire attention à ne l'incrémenter que si nécessaire (et comme la liste est en fait un tableau, la complexité ne sera malgré tout pas linéaire...)

Résolution des collisions par chaînage

Variante (avec une suppression correcte cette fois) pour les dictionnaires, suivant la même logique autorisant les doublons : *attention*, c'est encore moins malin que pour les ensembles.

```
def ajouter(table, cle, valeur) :
  table[h(cle)].append((cle, valeur))
  # comme on ne fait pas attention aux doublons, il y a potentiellement
  # plusieurs valeurs correspondant à la même clé... vraiment pas terrible, ça.
  # l'utilisation d'append fait que la version la plus récente est la dernière

def chercher(table, cle) :
  for key, val in table[h(cle)][::-1] : # attention, parcours à l'envers impératif !
    if key == cle : return val          # (pourquoi, au fait ?)
  return None

def supprimer(table, cle) :
  tmp = []
  for key, val in table[h(cle)] :
    if key != cle :
      tmp.append((key, val))
  table[h(cle)] = tmp
```

`supprimer` supprime bien toutes les occurrences (tout simplement car la boucle n'est pas interrompue dès la découverte de la première ocurrence). La recopie dans une nouvelle liste est un peu absurde mais est une manière de supprimer les chaînons en temps linéaire et en 4 lignes de code...

Résolution des collisions par chaînage

Il est nettement plus raisonnable de faire en sorte de ne jamais avoir de doublon. Version « ensemble » :

```
def ajouter(table, elt) :
  hache = h(elt)
  # précalculé une fois car utilisé deux fois : ce calcul est peut-être long
  if elt not in table[hache] :
    table[hache].append(elt)

def supprimer(table, elt) :
  table[h(elt)].remove(elt)

def chercher(table, elt) :
  return elt in table[h(elt)]
```

Complexité du hachage par chaînage

(on suppose dans la suite que le calcul de la fonction de hachage est en $O(1)$)

`ajouter()`, `rechercher()` et `supprimer()` ont un coût linéaire en la taille de la boîte où se trouve l'élément *(mais si on y tient, `ajouter()` peut avoir un coût constant)*

Question : que vaut cette taille (en moyenne sur les boîtes occupées) ?

Elle dépend du taux de charge $\alpha = \frac{n}{m}$ de la table.

α est donc la taille moyenne d'une boîte. Mais les boîtes ne sont pas toutes occupées, donc la taille moyenne d'une boîte occupée est supérieure à α...

Elle dépend donc également de la façon dont les données sont bien (ou mal) réparties dans la table, donc de la « qualité » de la fonction de hachage.

Plus précisément, on fait l'hypothèse suivante :

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 $\Theta(1 + \frac{n}{m})$.

Complexité du hachage par chaînage

Théorème

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

Idée de la démonstration : considérer séparément les recherches fructueuses, lorsque l'élément `x` *cherché est bien présent dans la table, et les recherches infructueuses, lorsque l'élément cherché n'est pas dans la table.*

- *lors d'une recherche infructueuse, toutes les cases de la liste* `H[h(x)]` *sont testées, y compris le* `None` *final. D'après l'hypothèse de hachage uniforme simple,* `h(x)` *est uniforme parmi* m *valeurs, donc la moyenne du nombre de tests est égale à (1 plus) la somme des longueurs des listes divisée par* m*. Or cette somme vaut* n*.*
- *lors d'une recherche fructueuse, c'est un peu plus compliqué car* `h(x)` *ne peut prendre comme valeur que celle d'une case occupée, et il y en a au plus* n*. Heureusement dans ce cas, il n'est pas nécessaire d'aller systématiquement jusqu'au bout de la liste* `H[h(x)]` *: la recherche s'interrompt dès que* `x` *est trouvé et les éléments situés dans sa liste mais après lui ne sont pas considérés. La vraie question est donc : quelle était la longueur moyenne de la liste* `H[h(x)]` *au moment de l'insertion de* `x` *? Si la table n'a connu que des insertions et pas de suppression, alors* `x` *a été inséré après exactement* i *éléments avec proba* $\frac{1}{n}$ *pour chaque* $i < n$*, et d'après le cas infructueux, sa boîte avait alors une taille moyenne de* $\frac{i}{m}$*. Le coût moyen d'une recherche fructueuse est donc :*

$$\frac{1}{n}\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right) = 1+\frac{1}{mn}\sum_{i=1}^{n-1} i = 1+\frac{1}{mn}\frac{n(n-1)}{2} = \Theta(1+\frac{n}{m})$$

Complexité du hachage par chaînage

Théorème

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

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* $O(1 + \alpha) = O(1)$.

Autrement dit : si on considère un ensemble de taille n connue (fixe), on peut le représenter par une table de hachage dont on choisit la longueur m pour assurer un certain taux de remplissage maximal α, et si la fonction de hachage choisie permet de respecter l'hypothèse de hachage uniforme simple, alors le coût moyen des opérations est bien $O(1)$. Cela signifie que certaines opérations sont plus longues, bien sûr, mais en moyenne (sur toutes les recherches dans toutes les tables vérifiant ces hypothèses), le coût est constant.
Mais ces hypothèses ne sont pas très réalistes...

Qu'en est-il si n *n'est pas connu à l'avance ?*

Redimensionnement de table de hachage

Lorsque n n'est pas connu à l'avance, l'idée est de faire grossir la table lorsque cela devient nécessaire pour ne pas dépasser le taux de remplissage autorisé. Et pour que le coût (cumulé) de ces redimensionnements ne soit pas trop élevé, ils sont effectués de moins en moins souvent au fur et à mesure que la table grandit, c'est-à-dire qu'ils sont de plus en plus importants (précisément, on assure une croissance exponentielle).

Soit α le taux de remplissage à *ne pas* dépasser.

Initialisation : choisir un `m` arbitraire, allouer un tableau `T1` de longueur `m` et choisir une fonction de hachage `h1` à valeurs dans $[\![1, \texttt{m}]\!]$

Évolution : au fur et à mesure des ajouts ou suppressions, calculer le taux de remplissage effectif β

Redimensionnement : si β atteint α :

- créer une nouvelle table `T2` de longueur $2 \cdot \texttt{m}$ et choisir une nouvelle fonction de hachage `h2` à valeurs dans $[\![1, 2 \cdot \texttt{m}]\!]$
- parcourir `T1` pour transférer tous ses éléments dans `T2`
- faire : `m, T1, h1 = 2 * m, T2, h2`

Complexité du redimensionnement

Elle est nichée dans le parcours de T1 pour la recopie : si une table de longueur m contient n éléments, le parcours a une complexité $\Theta(m+n)$.

Ici, $n = \alpha m$ avec α fixé, donc $\Theta(m+n) = \Theta(n)$.

chaque redimensionnement a donc un coût important, linéaire en n.... mais il a lieu après au moins n ajouts – et peut-être aussi d'autres redimensionnements ; si le dernier a un coût cn, il faut aussi compter $\frac{cn}{2}$ pour l'avant-dernier, $\frac{cn}{4}$ pour le précédent, etc.
Le coût cumulé des redimensionnements effectués jusqu'au moment où la taille atteint n est donc au plus : $\sum_{k\geqslant 0} \frac{cn}{2^k} = 2cn$.
Donc, même si chaque redimensionnement occasionne ponctuellement un coût important, le cumul de ces coûts est linéaire, comparable à celui des ajouts qui les ont provoqués.

On dit que le coût *amorti* (c'est-à-dire le coût total réparti sur les opérations précédentes) du redimensionnement est constant.

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 $\Theta(1)$.

Cela donne un sens à l'affirmation « les opérations des tables de hachage sont de coût constant », dont il faut quand même être conscient qu'elle est stricto sensu fausse !