Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

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

Aménagements pour la semaine prochaine :

- TD groupe INFO 1 : mardi 8h30-10h30, salle SG 2011
- TD groupe INFO 5 : mercredi 8h30-10h30, salle SG 1003

Le hachage

I. Principe général

Différentes représentations des ensembles

Pour rappel,

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	(en moyenne)
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$

Différentes représentations des ensembles

Pour rappel,

	tableau		liste chaînée		ABR
	non trié	trié	non triée	triée	(en moyenne)
recherche	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
insertion	$+\Theta(1)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$
suppression	$\Theta(n)$	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$	$\Theta(\log n)$

Question : est-ce vraiment optimal, si on ne demande ***que*** ces trois opérations ou si on accepte d'être moins efficace pour les autres (union, intersection, sélection...) ?

Soyons fous :

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Solution simple : l'adressage direct

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Solution simple : l'adressage direct

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Réponse (***naïve***) fréquente *(de certains d'entre vous au partiel – mais peu, c'est bien !)* : oui, évidemment ! il suffit :

- d'allouer un tableau `T` suffisamment grand
- d'indiquer par `True` ou `False` dans `T[i]` si `i` est dans l'ensemble

Solution simple : l'adressage direct

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Réponse (***naïve***) fréquente *(de certains d'entre vous au partiel – mais peu, c'est bien !)* : oui, évidemment ! il suffit :

- d'allouer un tableau `T` suffisamment grand
- d'indiquer par `True` ou `False` dans `T[i]` si `i` est dans l'ensemble

une solution, vraiment ? ? ?

- ***il faut*** que les éléments de l'ensemble soient des nombres, et même des entiers, sinon parler de `T[i]` n'a tout simplement pas de sens ;

Solution simple : l'adressage direct

Peut-on implémenter ces trois opérations en temps $O(1)$ (en sacrifiant éventuellement la complexité en espace, et/ou la complexité en temps pour les autres opérations) ?

Réponse (***naïve***) fréquente *(de certains d'entre vous au partiel – mais peu, c'est bien !)* : oui, évidemment ! il suffit :

- d'allouer un tableau `T` suffisamment grand
- d'indiquer par `True` ou `False` dans `T[i]` si `i` est dans l'ensemble

une solution, vraiment ? ? ?

- ***il faut*** que les éléments de l'ensemble soient des nombres, et même des entiers, sinon parler de `T[i]` n'a tout simplement pas de sens ;
- ***il faut*** que le nombre de valeurs possibles soit raisonnable : la complexité en espace est $\Theta($`max-min`$)$ où `min` et `max` sont les valeurs extrémales possibles, ***indépendamment*** de la taille `n` de l'ensemble

Pourtant, en Python par exemple…

… deux types de données sont « vendus [1] » pour assurer des accès avec une complexité en $O(1)$: les *ensembles* et les *dictionnaires*.

Exemples :

```
>>> S = { 'a', 2, 4, (1,1) }          # exemple d'ensemble
>>> 'a' in S; 'b' in S
True
False
>>> S.add('b'); S; 'b' in S
{2, 4, (1, 1), 'a', 'b'}
True
>>> S.remove(2); S.remove(4)
>>> S
{(1, 1), 'a', 'b'}
>>> S.add('b')
>>> S
{(1, 1), 'a', 'b'}
```

[1] *et certains d'entre vous croient trop facilement les publicités…*

Pourtant, en Python par exemple...

... deux types de données sont « vendus » pour assurer des accès avec une complexité en $O(1)$: les *ensembles* et les *dictionnaires*.

Exemples :

```
>>> D = { 'a' : 2, 'b' : 5, (1,2) : 'toto' } # exemple de dictionnaire
>>> 'c' in D; 'a' in D
False
True
>>> D['a']
2
>>> D['a'] = 'nouvelle␣valeur'; D['a']
'nouvelle␣valeur'
>>> D.pop('a'); D
'nouvelle␣valeur'
{'b': 5, (1, 2): 'toto'}
>>> D['c'] = 'coucou'; D
{'b': 5, (1, 2): 'toto', 'c': 'coucou'}
```

(non spécifique à Python *bien sûr, cf les* `HashMap` *en* Java *par exemple)*

une petite différence...

- les ensembles contiennent seulement des éléments (clés)
- dans les dictionnaires, des valeurs sont attachées aux clés (on parle parfois de données satellites)

Ensembles *vs* dictionnaires

une petite différence...

- les ensembles contiennent seulement des éléments (clés)
- dans les dictionnaires, des valeurs sont attachées aux clés (on parle parfois de données satellites)

pour beaucoup de points communs :

- les clés peuvent être de n'importe quel type ***non mutable*** : entiers, réels, chaînes de caractères, tuples (mais ***pas des listes***, par exemple)
- ... ce qui fait que l'ensemble de clés possibles est infini
- il n'y a pas de contrainte d'homogénéité
- la recherche est ***réputée*** coûter $O(1)$

Ensembles *vs* dictionnaires

une petite différence...

- les ensembles contiennent seulement des éléments (clés)
- dans les dictionnaires, des valeurs sont attachées aux clés (on parle parfois de données satellites)

pour beaucoup de points communs :

- les clés peuvent être de n'importe quel type ***non mutable*** : entiers, réels, chaînes de caractères, tuples (mais ***pas des listes***, par exemple)
- ... ce qui fait que l'ensemble de clés possibles est infini
- il n'y a pas de contrainte d'homogénéité
- la recherche est ***réputée*** coûter $O(1)$

en fait, les dictionnaires sont essentiellement des ensembles de couples (`clé`, `valeur`), rangés en tenant compte seulement de la clé

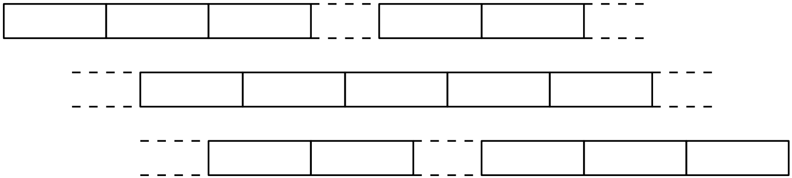
Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

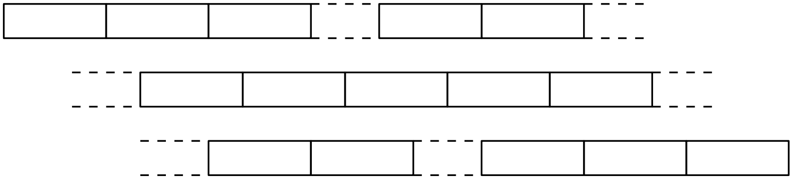
- allouer un (grand) tableau `T` de taille `m`



Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- 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`

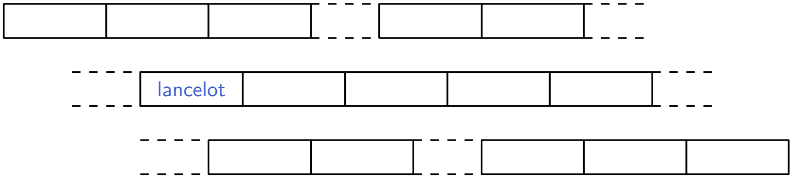


h(lancelot) = 12

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- 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*)

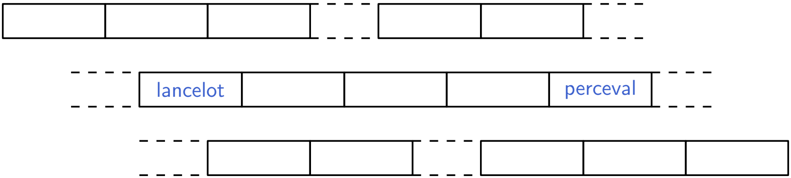


h(lancelot) = 12

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

- 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*)

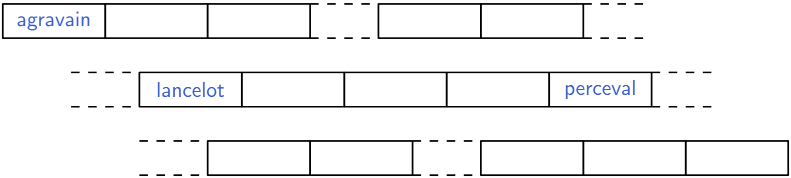


h(perceval) = 16

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- 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*)

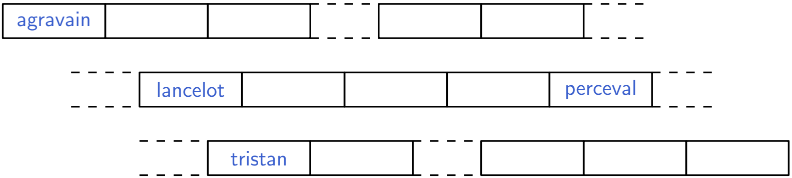


h(agravain) $= 1$

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier...

- 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*)

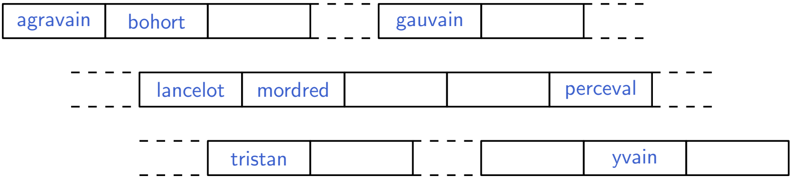


h(tristan) $=16$

Principe du hachage

Ensembles et dictionnaires sont des ***tables de hachage***, construites en généralisant le principe de l'adressage direct : puisqu'il faut des indices entiers, il suffit de tout transformer (ou presque) en entier…

- 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*)



Principe du hachage

Donc, en gros, les opérations d'ajout, suppression et recherche correspondent aux fonctions suivantes :

attention, version grossière – donc (vraiment très) très fausse

Pour les ensembles :

```
def ajouter(table, elt) :
  table[h(elt)] = True

def supprimer(table, elt) :
  table[h(elt)] = False

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

Pour les dictionnaires :

```
def ajouter(table, cle, valeur) :
  table[h(cle)] = valeur

def supprimer(table, cle) :
  table[h(cle)] = None

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

attention, version grossière – donc (vraiment très) très fausse

Version un tout petit peu moins naïve, tenant (un peu) compte du fait que chaque case peut correspondre à plusieurs clés différentes :

attention, version grossière – donc (vraiment très) très fausse

Pour les ensembles :

```
def ajouter(table, elt) :
  table[h(elt)] = elt

def supprimer(table, elt) :
  table[h(elt)] = None

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

Pour les dictionnaires :

```
def ajouter(table, cle, valeur) :
  table[h(cle)] = (cle, valeur)

def supprimer(table, cle) :
  table[h(cle)] = None

def chercher(table, cle) :
  return table[(h(cle)][1]
```

attention, version grossière – donc (vraiment très) très fausse

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très ***gros problème*** :

ensemble des clés possibles ***infini*** *vs* ensemble des valeurs hachées ***fini***

Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très ***gros problème*** :

ensemble des clés possibles ***infini*** *vs* ensemble des valeurs hachées ***fini***

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

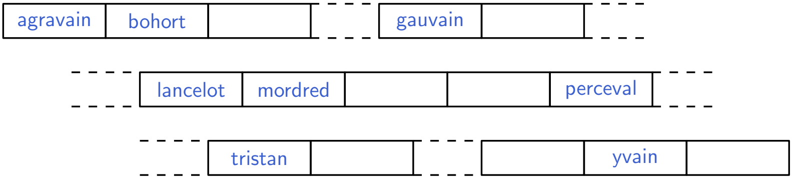
Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très ***gros problème*** :

ensemble des clés possibles ***infini*** *vs* ensemble des valeurs hachées ***fini***

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

dès qu'on cherche à insérer deux telles clés, il se produit une ***collision*** : deux éléments doivent être placés dans la même boîte.



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

Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

ensemble des clés possibles *infini* *vs* ensemble des valeurs hachées *fini*

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

dès qu'on cherche à insérer deux telles clés, il se produit une ***collision*** : deux éléments doivent être placés dans la même boîte.

le principe du hachage ne peut donc pas fonctionner sans prévoir un mécanisme additionnel de ***résolution des collisions***.

Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

ensemble des clés possibles ***infini*** *vs* ensemble des valeurs hachées ***fini***

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

dès qu'on cherche à insérer deux telles clés, il se produit une ***collision*** : deux éléments doivent être placés dans la même boîte.

le principe du hachage ne peut donc pas fonctionner sans prévoir un mécanisme additionnel de ***résolution des collisions***.

pour cela, il y a deux grandes méthodes :

Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

ensemble des clés possibles ***infini*** *vs* ensemble des valeurs hachées ***fini***

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

dès qu'on cherche à insérer deux telles clés, il se produit une ***collision*** : deux éléments doivent être placés dans la même boîte.

le principe du hachage ne peut donc pas fonctionner sans prévoir un mécanisme additionnel de ***résolution des collisions***.

pour cela, il y a deux grandes méthodes :

- la résolution des collisions par ***chaînage*** ;

Collisions

Il n'est pas nécessaire de réfléchir bien longtemps pour comprendre qu'il y a un très *gros problème* :

ensemble des clés possibles *infini* *vs* ensemble des valeurs hachées *fini*

donc nécessairement, il existe des clés `cle1` et `cle2` tq `h(cle1) = h(cle2)`.

dès qu'on cherche à insérer deux telles clés, il se produit une ***collision*** : deux éléments doivent être placés dans la même boîte.

le principe du hachage ne peut donc pas fonctionner sans prévoir un mécanisme additionnel de ***résolution des collisions***.

pour cela, il y a deux grandes méthodes :

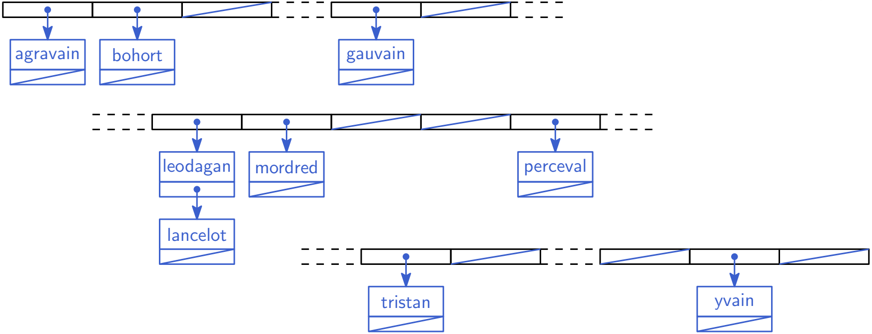
- la résolution des collisions par ***chaînage*** ;
- la résolution des collisions par ***sondage***.

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 – terminologie fortement déconseillée car ambiguë, voir plus loin

Résolution des collisions par chaînage

Cela donne donc, écrit avec des listes Python :

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

```
def ajouter(table, elt) :
  table[h(elt)].append(elt)


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


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

Résolution des collisions par chaînage

Cela donne donc, écrit avec des listes Python :

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

```
def ajouter(table, elt) :
  table[h(elt)].append(elt)
  # ?! sans même vérifier si elt est déjà dans table ??? admettons...

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


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

Résolution des collisions par chaînage

Cela donne donc, écrit avec des listes Python :

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

```
def ajouter(table, elt) :
  table[h(elt)].append(elt)
   # ?! sans même vérifier si elt est déjà dans table ??? admettons...
def supprimer(table, elt) :
  table[h(elt)].remove(elt)
   # ... mais il faut supprimer toutes les occurrences, remove ne suffit pas
def chercher(table, elt) :
  return elt in table[h(elt)]
```

Remarque : les `lists` Python *sont des tableaux ; des listes (doublement) chaînées* `deques` *sont définies dans le paquet* `collections` *; mais même les* `deques` *sont complètement inadaptées pour implémenter la fonction* `supprimer` *si on autorise les doublons car elles ne permettent pas de manipuler finement le chaînage ; de tels appels successifs à* `remove` *seraient une catastrophe en terme de complexité...*

Résolution des collisions par chaînage

Cela donne donc, écrit avec des listes Python :

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

```
def ajouter(table, elt) :
  table[h(elt)].append(elt)
  # ?! sans même vérifier si elt est déjà dans table ??? admettons...

def supprimer(table, elt) :
  table[h(elt)].remove(elt)
  # ... mais il faut supprimer toutes les occurrences, remove ne suffit pas

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

Remarque : les `lists` Python sont des tableaux ; des listes (doublement) chaînées `deques` sont définies dans le paquet `collections` ; mais même les `deques` sont complètement inadaptées pour implémenter la fonction `supprimer` si on autorise les doublons car elles ne permettent pas de manipuler finement le chaînage ; de tels appels successifs à `remove` seraient une catastrophe en terme de complexité...

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)) # attention, doublons potentiels

def chercher(table, cle) :
  for key, val in table[h(cle)][::-1] : # parcours à l'envers obligatoire !
    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
```

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)) # attention, doublons potentiels

def chercher(table, cle) :
  for key, val in table[h(cle)][::-1] : # parcours à l'envers obligatoire !
    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
```

(la recopie dans une nouvelle liste est un peu absurde mais permet 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 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 `h(cle)` 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) ?

Complexité du hachage par chaînage

on suppose dans la suite que le calcul de `h(cle)` *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.

Complexité du hachage par chaînage

on suppose dans la suite que le calcul de `h(cle)` 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.

Théorème

si la répartition des éléments est uniforme *dans la table, le coût moyen d'une recherche (réussie ou non) est* $O(1 + \frac{n}{m})$.

Complexité du hachage par chaînage

Plus précisément :

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

Complexité du hachage par chaînage

Plus précisément :

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

Complexité du hachage par chaînage

Plus précisément :

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

mais si n ***n'est pas connu à l'avance ?***

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

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

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

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

Redimensionnement : si β atteint α :

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

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 $2 \cdot \mathtt{m}$ et choisir une nouvelle fonction de hachage h2 à valeurs dans $[\![1, 2 \cdot \mathtt{m}]\!]$

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

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 $2 \cdot \mathtt{m}$ et choisir une nouvelle fonction de hachage `h2` à valeurs dans $[\![1, 2 \cdot \mathtt{m}]\!]$
- parcourir `T1` pour transférer tous ses éléments dans `T2`

Redimensionnement de table de hachage

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 $[\![1, \mathtt{m}]\!]$

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 $2 \cdot \mathtt{m}$ et choisir une nouvelle fonction de hachage `h2` à valeurs dans $[\![1, 2 \cdot \mathtt{m}]\!]$
- 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 $\Theta(m + n)$.

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 $\Theta(m+n) = \Theta(n)$.

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 un coût linéaire en n... 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 est linéaire en n)*

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 un coût linéaire en n… 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 est linéaire en* n*)*

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

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 un coût linéaire en n... 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 est linéaire en n)*

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* $\Theta(1)$.

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 un coût linéaire en n... 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 est linéaire en n)*

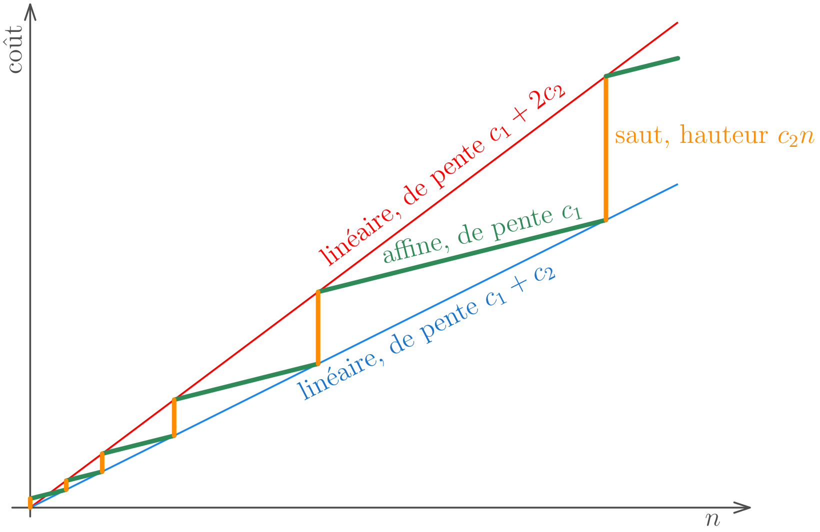
On dit que le coût ***amorti*** (c'est-à-dire le coût total réparti sur les opérations précédentes) du redimensionnement est constant.

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 !

Complexité du redimensionnement



Évolution du coût de n insertions successives