

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 2021-2022

LE HACHAGE

I. Principe général

DIFFÉRENTES REPRÉSENTATIONS DES ENSEMBLES

Pour rappel, les cours précédents ont montré qu'on pouvait obtenir les complexités suivantes pour les opérations usuelles sur les ensembles :

	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, les cours précédents ont montré qu'on pouvait obtenir les complexités suivantes pour les opérations usuelles sur les ensembles :

	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) ?

Réponse (*naïve*) fréquente : 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 : 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 : 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 $\Theta(\text{max-min})$ 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 » 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'}
```


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)

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)

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 en fait 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...

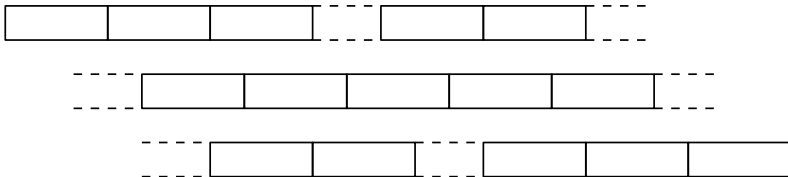
- allouer un (grand) tableau **T** de taille **m**



PRINCIPE DU HACHAGE

Ensembles et dictionnaires sont en fait 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

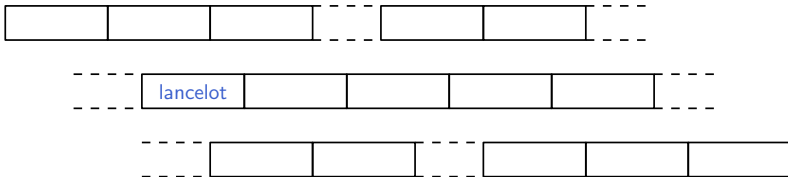


$$h(\text{lancelot}) = 12$$

PRINCIPE DU HACHAGE

Ensembles et dictionnaires sont en fait 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...

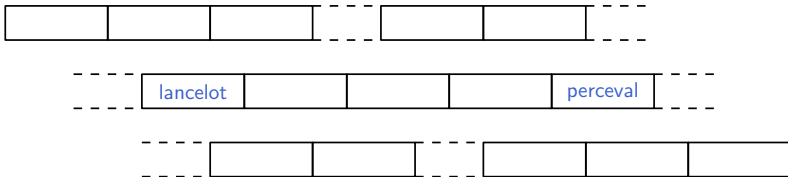
- allouer un (grand) tableau T de taille m
- transformer n'importe quelle clé en entier plus petit que m à l'aide d'une fonction de hachage h
- stocker chaque élément elt dans la case $T[h(elt)]$ (on parle de *boîte* ou *bucket*)



PRINCIPE DU HACHAGE

Ensembles et dictionnaires sont en fait 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*)

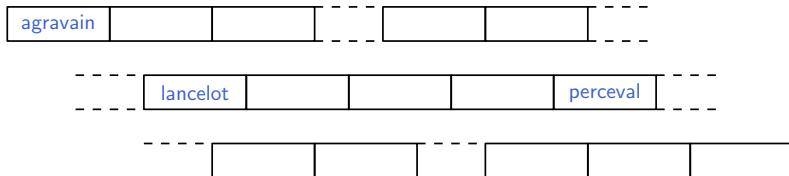


$$h(\text{perceval}) = 16$$

PRINCIPE DU HACHAGE

Ensembles et dictionnaires sont en fait 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*)

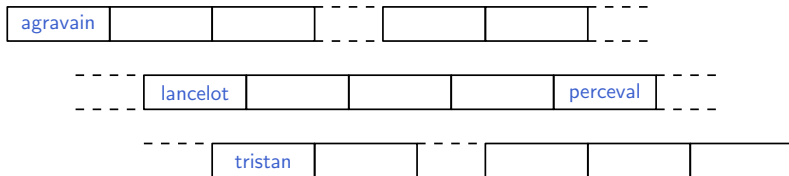


$$h(\text{agravain}) = 1$$

PRINCIPE DU HACHAGE

Ensembles et dictionnaires sont en fait 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

PRINCIPE DU HACHAGE

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

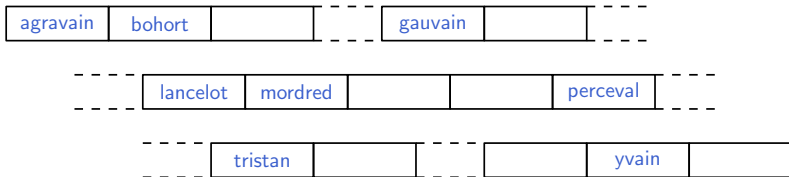
COLLISIONS

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.



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

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*.

pour cela, il y a deux grandes méthodes :

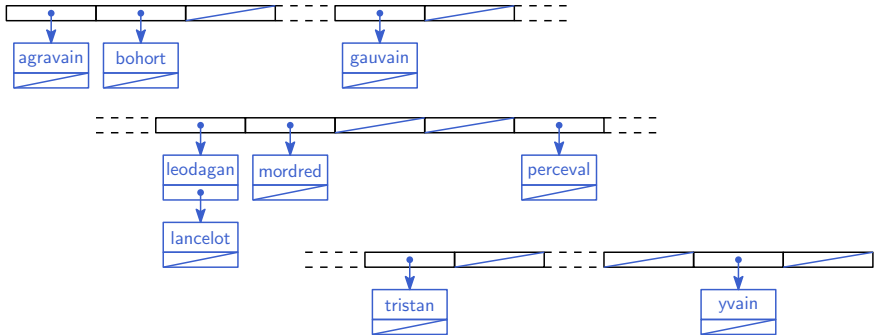
- la résolution des collisions par *chaînage* ;
- la résolution des collisions par *sondage*.

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

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.

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.

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 T_1 de longueur m
et choisir une fonction de hachage h_1 à valeurs dans $\llbracket 1, m \rrbracket$

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 $\llbracket 1, m \rrbracket$

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 T_1 de longueur m et choisir une fonction de hachage h_1 à valeurs dans $\llbracket 1, m \rrbracket$

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 $\llbracket 1, m \rrbracket$

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 $2 \cdot m$ et choisir une nouvelle fonction de hachage $h2$ à valeurs dans $\llbracket 1, 2 \cdot m \rrbracket$

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 $\llbracket 1, m \rrbracket$

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 $2 \cdot m$ et choisir une nouvelle fonction de hachage $h2$ à valeurs dans $\llbracket 1, 2 \cdot m \rrbracket$
- 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 $\llbracket 1, m \rrbracket$

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 $2 \cdot m$ et choisir une nouvelle fonction de hachage $h2$ à valeurs dans $\llbracket 1, 2 \cdot m \rrbracket$
- 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 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 ; heureusement, lui aussi est linéaire en n)

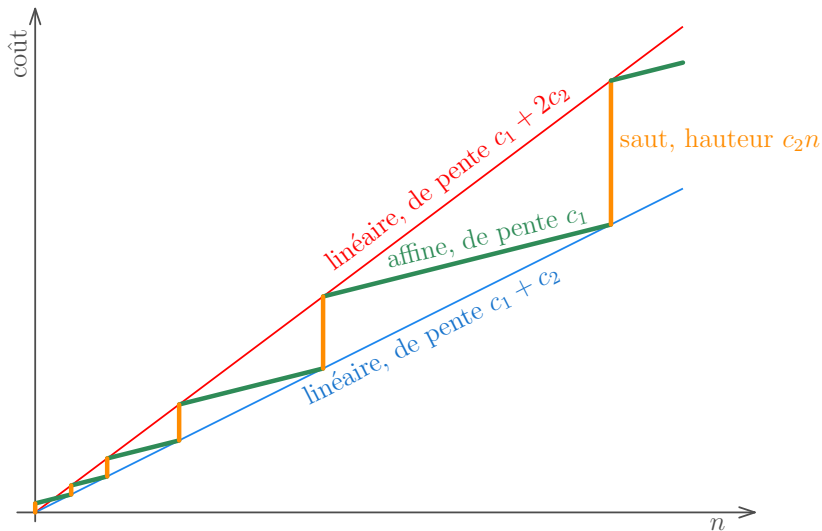
On dit que le coût **amorti** (c'est-à-dire le coût total réparti sur les opérations précédentes) du redimensionnement est constant.

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

COMPLEXITÉ DU REDIMENSIONNEMENT



Évolution du coût de n insertions successives