Structures mixtes

Yann Strozecki
yann.strozecki@uvsq.fr

Novembre 2016

Structure d'ensemble

Dans ce cours on veut généralement représenter un ensemble dynamique d'objets.

On veut pouvoir faire les opérations suivantes :

- insertion
- suppression
- recherche

Listes vs Tableaux

Défauts des listes :

- Temps d'accès à un élément à l'intérieur de la liste lent.
- Deux champs pour chaque élément : un pointeur et une valeur.
- Mémoire non contigüe, très mauvais avec les processeurs récents (cache).

Défauts des tableaux :

- Beaucoup d'espace gâché quand il est peu rempli.
- Taille fixe (structure statique).

Listes vs Tableaux

Défauts des listes :

- Temps d'accès à un élément à l'intérieur de la liste lent.
- Deux champs pour chaque élément : un pointeur et une valeur.
- Mémoire non contigüe, très mauvais avec les processeurs récents (cache).

Défauts des tableaux :

- Beaucoup d'espace gâché quand il est peu rempli.
- Taille fixe (structure statique).

On va proposer des structures qui ne souffrent d'aucun de ces défauts.

Listes vs Tableaux

Défauts des listes :

- Temps d'accès à un élément à l'intérieur de la liste lent.
- Deux champs pour chaque élément : un pointeur et une valeur.
- Mémoire non contigüe, très mauvais avec les processeurs récents (cache).

Défauts des tableaux :

- Beaucoup d'espace gâché quand il est peu rempli.
- Taille fixe (structure statique).

On va proposer des structures qui ne souffrent d'aucun de ces défauts.

Tableaux dynamiques

Ce sont des tableaux dont la taille est dynamique. Pour cette raison on les appelle parfois tableaux listes. Ils sont très courants dans les langages de programmation :
std : :vector en C++, ArrayList en JAVA ou list en Python.

Type

```
Enregistrement DynTableau {
   Tableau     : Tableau d'entier;
   Taille : entier;
   TailleMax : entier;
}
```

Lors de l'initialisation Taille est mise à 0, TailleMax à une constante disons 1000 et Tableau est un pointeur sur une zone mémoire de taille 1000.

Tableaux dynamiques

Ce sont des tableaux dont la taille est dynamique. Pour cette raison on les appelle parfois tableaux listes. Ils sont très courants dans les langages de programmation :
std : :vector en C++, ArrayList en JAVA ou list en Python.

Type

```
Enregistrement DynTableau {
   Tableau      : Tableau d'entier;
   Taille : entier;
   TailleMax : entier;
}
```

Lors de l'initialisation Taille est mise à 0, TailleMax à une constante disons 1000 et Tableau est un pointeur sur une zone mémoire de taille 1000.

Fonctionnement d'un tableau dynamique (insertion)

Algorithme 1 : Insertion

```
Data : T : DynTableau, e : entier
if T.Taille == T.TailleMax then
    T.TailleMax = 2*T.TailleMax ;
    allocation de mémoire pour T.Tableau
T.Tableau[Taille]=e;
T.Taille ++;
```

Dans le pire des cas on doit allouer de la mémoire à l'insertion donc la complexité est $O(T.Taille)$.

En moyenne sur une suite d'opérations, il y a peu d'allocations mémoire et l'insertion se fait en $O(1)$.

Fonctionnement d'un tableau dynamique (insertion)

Algorithme 2 : Insertion

Data : T : DynTableau, e : entier

if *T.Taille == T.TailleMax* **then**

    T.TailleMax = 2*T.TailleMax ;

    allocation de mémoire pour T.Tableau

T.Tableau[Taille]=e;

T.Taille ++;

Dans le pire des cas on doit allouer de la mémoire à l'insertion donc la complexité est $O(T.Taille)$.

En moyenne sur une suite d'opérations, il y a peu d'allocations mémoire et l'insertion se fait en $O(1)$.

Fonctionnement d'un tableau dynamique (suppression)

Algorithme 3 : Suppression

Data : T : DynTableau

```
if T.Taille < T.TailleMax / 4 et T.Taille > 1000 then
    T.TailleMax = T.TailleMax/2 ;
    désallocation de mémoire pour T.Tableau
T.Taille - -;
```

Dans le pire des cas on doit désallouer de la mémoire donc la complexité est $O(T.Taille)$.

En moyenne sur une suite d'opérations, il y a peu de désallocations mémoire et la suppression se fait en $O(1)$.

Fonctionnement d'un tableau dynamique (suppression)

Algorithme 4 : Suppression

Data : T : DynTableau

if *T.Taille < T.TailleMax / 4 et T.Taille > 1000* **then**
    T.TailleMax = T.TailleMax/2 ;
    désallocation de mémoire pour T.Tableau

T.Taille - -;

Dans le pire des cas on doit désallouer de la mémoire donc la complexité est $O(T.Taille)$.

En moyenne sur une suite d'opérations, il y a peu de désallocations mémoire et la suppression se fait en $O(1)$.

La quantité de mémoire allouée est *à un facteur 4 près* la quantité de mémoire réellement utilisée.

Fonctionnement d'un tableau dynamique (suppression)

Algorithme 5 : Suppression

Data : T : DynTableau

if *T.Taille < T.TailleMax / 4 et T.Taille > 1000* **then**

    T.TailleMax = T.TailleMax/2 ;

    désallocation de mémoire pour T.Tableau

T.Taille - -;

Dans le pire des cas on doit désallouer de la mémoire donc la complexité est $O(T.Taille)$.

En moyenne sur une suite d'opérations, il y a peu de désallocations mémoire et la suppression se fait en $O(1)$.

La quantité de mémoire allouée est *à un facteur 4 près* la quantité de mémoire réellement utilisée.

Analyse amortie

- Combien y a-t-il d'insertions “gentilles” entre deux insertions “mauvaises” sachant que la première mauvaise insertion a lieu pour un taille 10000 ?
- Quel est le temps moyen d'une insertion dans la séquence décrite ?
- Même question pour les suppressions.

Ce genre d'analyse de complexité s'appelle l'analyse amortie. La complexité d'une opération est évaluée en regardant son coût moyen pour un grand nombre de répétitions de l'opération.

Adressage direct et indirect

Une table de hachage est une généralisation de la notion de tableau. Nous considérons un ensemble d'éléments qui ont une valeur et une clé.

On peut stocker chaque élément à une position dans un tableau donné par sa clé. C'est qu'on appelle l'adressage direct. Il faut un tableau avec une entrée pour chaque clé possible.

Quand le nombre total de clés possibles est très supérieur au nombre d'éléments stockés on va utiliser une table de hachage. À chaque clé on va associer un indice dans un tableau en lui appliquant une fonction de hachage. C'est ce qu'on appelle l'adressage indirect.

Adressage direct et indirect

Une table de hachage est une généralisation de la notion de tableau. Nous considérons un ensemble d'éléments qui ont une valeur et une clé.

On peut stocker chaque élément à une position dans un tableau donné par sa clé. C'est qu'on appelle l'adressage direct. Il faut un tableau avec une entrée pour chaque clé possible.

Quand le nombre total de clés possibles est très supérieur au nombre d'éléments stockés on va utiliser une table de hachage. À chaque clé on va associer un indice dans un tableau en lui appliquant une fonction de hachage. C'est ce qu'on appelle l'adressage indirect.

Fonction de hachage

On note m le nombre de cases de la table de hachage. Les fonctions de hachage doivent associer à n'importe quelle clé un nombre dans $[0, \ldots, m-1]$.

- Méthode de la division : $h(k) = k \mod m$.
- Méthode de la multiplication : $h(k) = \lfloor m(\lceil kA \rceil - kA) \rfloor$ avec $A = \frac{\sqrt{5}-1}{2}$.

Propriétés fondamentales : uniformité, surjectivité.

Fonction de hachage

On note m le nombre de cases de la table de hachage. Les fonctions de hachage doivent associer à n'importe quelle clé un nombre dans $[0, \ldots, m-1]$.

- Méthode de la division : $h(k) = k \mod m$.
- Méthode de la multiplication : $h(k) = \lfloor m(\lceil kA \rceil - kA) \rfloor$ avec $A = \frac{\sqrt{5}-1}{2}$.

Propriétés fondamentales : uniformité, surjectivité.

Proposer une autre fonction pour hacher des suites de bits ou des chaînes de caractères.
Voir http ://www.sinfocol.org/herramientas/hashes.php

Fonction de hachage

On note m le nombre de cases de la table de hachage. Les fonctions de hachage doivent associer à n'importe quelle clé un nombre dans $[0, \ldots, m-1]$.

- Méthode de la division : $h(k) = k \mod m$.
- Méthode de la multiplication : $h(k) = \lfloor m(\lceil kA \rceil - kA) \rfloor$ avec $A = \frac{\sqrt{5}-1}{2}$.

Propriétés fondamentales : uniformité, surjectivité.

Proposer une autre fonction pour hacher des suites de bits ou des chaînes de caractères.
Voir http ://www.sinfocol.org/herramientas/hashes.php

Exemple d'insertion

On a un tableau à $m = 9$ éléments, et on veut insérer les éléments de clés : $5, 28, 19, 15, 20, 33$. On utilise une fonction de hachage qui utilise la méthode de la division.
Au départ, la table est vide.

- On calcule $h(5) = 5$. On insère donc l'élément de clé 5 dans la liste $T[5]$.
- On calcule ensuite $h(28) = 1$. On insère donc l'élément de clé 28 dans la liste $T[1]$.
- On calcule ensuite $h(19) = 1$. On insère donc l'élément de clé 19 dans la liste $T[1]$.
- On a inséré deux éléments au même endroit, c'est une collision.
- On continue comme cela avec tous les éléments.

Exemple au tableau.

Collisions dans une table de hachage

En première approche, quand il y a collision, on perd un élément. On voudrait donc éviter les collisions.

On peut prendre m, la taille de la table très grande. On s'approche de l'adressage direct. Mauvaise solution car de l'espace est gâché.

Collisions dans une table de hachage

En première approche, quand il y a collision, on perd un élément. On voudrait donc éviter les collisions.

On peut prendre m, la taille de la table très grande. On s'approche de l'adressage direct. Mauvaise solution car de l'espace est gâché.

On appelle facteur de charge le nombre d'éléments stockés dans la table divisé par la taille de la table. Idéalement on voudrait qu'il soit proche de 1.
Pourquoi ?

Collisions dans une table de hachage

En première approche, quand il y a collision, on perd un élément. On voudrait donc éviter les collisions.

On peut prendre m, la taille de la table très grande. On s'approche de l'adressage direct. Mauvaise solution car de l'espace est gâché.

On appelle facteur de charge le nombre d'éléments stockés dans la table divisé par la taille de la table. Idéalement on voudrait qu'il soit proche de 1.
Pourquoi ?

De la quantité de collisions

On ne peut pas éviter les collisions efficacement même avec m grand à cause du paradoxe des anniversaires : si on insère environ $\sqrt{m}$ éléments dans la table on va avoir plus d'une chance sur deux de collision.

Les tables de hachages peuvent donner lieu à des attaques par déni de service. Un *adversaire* qui connaîtrait la fonction de hachage peut insérer des éléments qui ont le même hachage ce qui créé plein de collisions et donc de mauvaises performances. Attaque connue de *SHA-1*.

De la quantité de collisions

On ne peut pas éviter les collisions efficacement même avec m grand à cause du paradoxe des anniversaires : si on insère environ $\sqrt{m}$ éléments dans la table on va avoir plus d'une chance sur deux de collision.

Les tables de hachages peuvent donner lieu à des attaques par déni de service. Un *adversaire* qui connaîtrait la fonction de hachage peut insérer des éléments qui ont le même hachage ce qui créé plein de collisions et donc de mauvaises performances. Attaque connue de *SHA-1*.

Résolution des collisions par chaînage

On utilise les listes chaînées pour gérer ce problème : à un indice du tableau correspond la liste des éléments hachés à cet indice. Le type de ce genre de table de hachage est donc un tableau de listes d'entiers.

On peut ainsi stocker un nombre quelconque d'éléments.

Exemple d'une suite d'opérations sur une table de hachage au tableau.

Résolution des collisions par chaînage

On utilise les listes chaînées pour gérer ce problème : à un indice du tableau correspond la liste des éléments hachés à cet indice. Le type de ce genre de table de hachage est donc un tableau de listes d'entiers.

On peut ainsi stocker un nombre quelconque d'éléments.

Exemple d'une suite d'opérations sur une table de hachage au tableau.

Résolution des collisions par chaînage (2)

Algorithme 6 : recherche

```
Data : T : TableHachage, cle : entier
L : ↑ Elément ;
L = T[h(cle)];
while L ≠ NIL do
    if L.val == cle then
        return True ;
    L = L.suivant ;
return False;
```

Complexité dans le pire des cas des trois opérations importantes *recherche*, *insertion*, *suppression* ?

Résolution des collisions par chaînage (2)

Algorithme 7 : recherche

Data : T : TableHachage, cle : entier

```
L : ↑ Elément ;
L = T[h(cle)];
while L ≠ NIL do
    if L.val == cle then
        return True ;
    L = L.suivant ;
return False;
```

Complexité dans le pire des cas des trois opérations importantes *recherche*, *insertion*, *suppression* ? Pour n éléments, $O(n)$.

Résolution des collisions par chaînage (2)

Algorithme 8 : recherche

Data : T : TableHachage, cle : entier

```
L : ↑ Elément ;
L = T[h(cle)];
while L ≠ NIL do
    if L.val == cle then
        return True ;
    L = L.suivant ;
return False;
```

Complexité en moyenne si le facteur de charge est α pour les trois opérations : $1 + \alpha$

Calcul de la complexité moyenne

1. L'insertion est toujours rapide, si on insère au début de la liste, complexité en $O(1)$.
2. Pour l'analyse en moyenne, on utilise l'uniformité : les éléments ont la même probabilité d'être hachés vers n'importe laquelle des m cases.
3. Soit n_i la taille de la liste dans la case i, la complexité de recherche ou de suppression dans la case i est n_i
4. La complexité moyenne de recherche ou de suppression est donc la taille moyenne d'une liste dans une case. Par définition, cette taille est le facteur de charge α.

Résolution des collisions par adressage ouvert

On veut que tous les éléments soient stockés dans la table de hachage elle-même. Pour résoudre une collision, on inspecte une suite de cases dans la table jusqu'à en trouver une vide.

On utilise une fonction de hachage qui aura deux arguments : la clé et le nombre de cases déjà sondées. Pour une clé c, on inspectera les cases $h(c, 0)$, $h(c, 1)$,$h(c, 2) \dots h(c, m-1)$ jusqu'à trouver une case vide pour insérer l'élément.

Résolution des collisions par adressage ouvert

On veut que tous les éléments soient stockés dans la table de hachage elle-même. Pour résoudre une collision, on inspecte une suite de cases dans la table jusqu'à en trouver une vide.

On utilise une fonction de hachage qui aura deux arguments : la clé et le nombre de cases déjà sondées. Pour une clé c, on inspectera les cases $h(c, 0)$, $h(c, 1)$,$h(c, 2) \ldots h(c, m-1)$ jusqu'à trouver une case vide pour insérer l'élément.

L'inconvénient de cette méthode est que le nombre d'éléments que l'on peut stocker est inférieur ou égal à m.

Résolution des collisions par adressage ouvert

On veut que tous les éléments soient stockés dans la table de hachage elle-même. Pour résoudre une collision, on inspecte une suite de cases dans la table jusqu'à en trouver une vide.

On utilise une fonction de hachage qui aura deux arguments : la clé et le nombre de cases déjà sondées. Pour une clé c, on inspectera les cases $h(c, 0)$, $h(c, 1)$,$h(c, 2) \ldots h(c, m-1)$ jusqu'à trouver une case vide pour insérer l'élément.

L'inconvénient de cette méthode est que le nombre d'éléments que l'on peut stocker est inférieur ou égal à m.

Résolution des collisions par adressage ouvert (2)

Il existe 3 types de fonctions de hachage usuelles :

Le sondage linéaire $h(k,i) = (h'(k) + i) \mod m$ avec i le nombre de sondages déjà effectués et h' une fonction de hachage classique vue dans le cas précédent.

Le sondage quadratique $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ avec c_1 et c_2 des constantes quelconques.

Le double hachage $h(k,i) = (h_1(k) + i h_2(k))$

Quelle est la difficulté avec la suppression ?

Résolution des collisions par adressage ouvert (2)

Il existe 3 types de fonctions de hachage usuelles :

Le sondage linéaire $h(k,i) = (h'(k) + i) \mod m$ avec i le nombre de sondages déjà effectués et h' une fonction de hachage classique vue dans le cas précédent.

Le sondage quadratique $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ avec c_1 et c_2 des constantes quelconques.

Le double hachage $h(k,i) = (h_1(k) + i h_2(k))$

Quelle est la difficulté avec la suppression ?

Résolution des collisions par adressage ouvert (3)

Exemple d'une suite d'opérations sur une table de hachage au tableau.

Le facteur de charge α est toujours inférieur ou égal à 1. La complexité en moyenne de l'insertion et de la recherche est $O(\frac{1}{1-\alpha})$.

Résolution des collisions par adressage ouvert (3)

Exemple d'une suite d'opérations sur une table de hachage au tableau.

Le facteur de charge α est toujours inférieur ou égal à 1. La complexité en moyenne de l'insertion et de la recherche est $O(\frac{1}{1-\alpha})$.