Cours 2 : Structures de données

1. Tableaux

1.1. Définition et exemple.

Définition 1.1. *Un **tableau** (en anglais,* array*) est une structure de donnée représentant un ensemble d'éléments accessibles par un **indice** numérique.*

Structure de bas niveau, elle est très courante dans tous les langages de programmation. Concrètement, les éléments sont stockés dans des **espaces mémoires consécutifs** ce qui permet un accès indicé rapide.

Quelques exemples : les tableaux C, les `Vector` en C++, les `list` en python, les `ArrayList` en Java...

Exemple 1.2.
En C, un tableau est simlement un *pointeur* sur la première case du tableau.

```c
int main(void) {
    int T[10] = {1,2,3,4,5,6,7,8,9,10};
    int i;
    int *p = T;

    for(i=0; i<10; i++) printf("%d ",T[i]);
    printf("\n");

    for(i=0; i<10; i++) printf("%d ",*(p+i));
    printf("\n");

}
```

1.2. Complexité.

- Accès au premier élément : $O(1)$
- Accès à l'élément i : $O(1)$
- Accès au dernier élément : $O(1)$
- Ajout d'un élément en début de tableau : $O(n)$
- Ajout d'un élément en position i : $O(n-i)$
- Ajout d'un élément en fin de tableau : $O(1)$
- Suppression du premier élément : $O(n)$
- Suppression de l'élément i : $O(n-i)$
- Suppression du dernier élément : $O(1)$

1.3. Le problème du dépassement de capacité. .

Problème : on ne peut réserver qu'un espace de mémoire donnée. Combien faut-il réserver d'espace ? Que se passe-t-il quand le tableau est plein ?

```
L = []
i = 0
while i < 500:
    L.append(i)
    i+=1
```

A chaque fois que je dépasse la capacité initiale, il faut **réallouer** un espace plus important et **recopier** l'ensemble des valeurs.

On va étudier les différentes stratégie en se basant sur l'algorithme suivant

```
Input : n un entier
Processus :
    T un tableau vide avec une certaine allocation initiale
    Pour i allant de 0 à n :
        Si T est plein :
            Reallouer(T)
        T[i] <- i
```

Exemple 1.3.
Une première solution.

— On alloue initialement une certaine taille K.

— Quand le tableau est plein, on augmente la taille de K.

Par exemple, si on choisis $n = 100$ et $K = 10$. Le tableau aura initialement une taille de 10. Lorsqu'on ajoutera le 11ème élément, on devra allouer une taille de 20 et recopier les 10 premiers éléments. Lorsqu'on ajoutera le 21ème élément, on allouera 30 et on recopiera les 20 premiers, etc. Au final, si on compte le nombre d'affectations dans le tableau, on trouve :

$$100 + 10 + 20 + 30 + 40 + 50 + \cdots + 90. \tag{1.1}$$

De façon générale, si $m = \frac{n}{K}$, la complexité est de

$$n + \sum_{i=1}^{m} ik = n + K \sum_{i=0}^{m} i \tag{1.2}$$

$$= n + K \frac{(m+1)m}{2} \tag{1.3}$$

$$= \frac{1}{2K} n^2 + \frac{3}{2} n. \tag{1.4}$$

c'est-à-dire $O(n^2)$.

Exemple 1.4.
Une deuxième solution.

— On alloue initialement une taille 1.

— Lors d'un dépassement, on multiplie la taille par 2.

Pour le cas où $n = 100$, le nombre d'affectations sera donc :

$$100 + 1 + 2 + 4 + 8 + 16 + 32 + 64. \tag{1.5}$$

Lors de la dernière réallocation, la taille est de 128 et peut contenir l'ensemble du tableau. De façon générale, soit $m = \lfloor \log(n) \rfloor$, c'est-à-dire

$$2^m \leq n < 2^{m+1}, \tag{1.6}$$

le nombre d'affectations est de

$$n + \sum_{i=1}^{m} 2^i = n + \frac{2^{m+1} - 1}{2 - 1} \tag{1.7}$$

$$= n + 2 \cdot 2^m - 1 \tag{1.8}$$

$$\leq 3n. \tag{1.9}$$

En conclusion, la complexité est de $O(n)$.

Exemple 1.5.
Un exemple concret, la doc python sur la réallocation des listes :

```
/* This over-allocates proportional to the list size, making room
* for additional growth. The over-allocation is mild, but is
* enough to give linear-time amortized behavior over a long
* sequence of appends() in the presence of a poorly-performing
* system realloc().
* The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
*/
```

2. Listes chaînées

2.1. Définition.

Définition 2.1. *Une **liste chaînée** est un ensemble de* cellules *telles que chaque cellule contient une* valeur *ainsi que* l'adresse *en mémoire de la cellule suivante. La liste en tant que telle est donnée par* l'adresse de la première cellule.

Exemple 2.2.
Exemple d'implantation en `C`.

```
typedef struct c{
    int valeur;
    struct c *suivante;
}Cellule;

typedef Cellule * Liste;

Cellule * creeCellule(int valeur) {
    Cellule * c = malloc(sizeof(Cellule));
    if(c == NULL) {
        exit(0);
    }
    c-> valeur = valeur;
    c-> suivante = NULL;
    return c;
}
void afficheListe(Liste L) {
    Cellule *c = L;
    while(c!=NULL) {
        printf("%d\n", c->valeur);
        c = c->suivante;
    }

}
void ajouteEnTete(Liste *L, Cellule * c) {
    c->suivante = *L;
    *L = c;
}
```

En pseudo-code, on écrira la structure de cette façon :

```
Structure Cellule :
    Entier valeur
    Cellule suivante

Structure Liste :
    Cellule premiere
```

On supposera qu'une structure n'est manipulée que *par référence* (adresse). Si on a une variable `c` de type `Cellule`, on accède à ses paramètres par la syntaxe `c.valeur`.

Exemple de listes chaînées implantées nativement : `LinkedList` en Java.

2.2. **Complexité.**

Exercice 1.
Écrivez les algorithmes et calculez les complexités pour les opérations suivantes :

- Accès au premier élément : $O(1)$
- Accès à l'élément i : $O(i)$
- Accès au dernier élément : $O(n)$
- Ajout d'un élément en début de liste : $O(1)$
- Ajout d'un élément en position i : $O(i)$
- Ajout d'un élément en fin de liste : $O(n)$
- Suppression du premier élément : $O(1)$
- Suppression de l'élément i : $O(i)$
- Suppression du dernier élément : $O(n)$

2.3. **En pratique.** .

Inconvénients :

- Complexité élevée d'accès aux valeurs
- En pratique, coût élevé de l'allocation mémoire

Avantages :

- S'il y a de fortes contraintes sur la mémoire
- Facilité d'accès et d'ajout en début de liste

Remarque : on peut modifier légèrement la structure pour un accès rapide aux dernières valeurs en ajoutant un **pointeur sur la dernière cellule** dans la structure de liste.

Quelques structures similaires : listes chaînées circulaires, listes doublement chaînées.

3. Ensembles : tables de hachage

3.1. **Quel problème ?** On souhaite définir une structure E tel que les opérations suivantes soient rapide :

- Test si un élément e est dans E
- Ajout d'un élément
- Suppression d'un élément

Exemple 3.1.
Soit E un ensemble de nombres entiers entre 1 et 100 (chaque nombre peut apparaître au plus une seule fois). **Quelle structure peut-on adopter pour que les questions posées puissent être résolue en O(1) ?**

On utilise un tableau booléen de de taille 100 : si `t[i] = False` alors `i` n'est pas dans E, et si `t[i] = True` alors `i` est dans E.

3.2. **Structure générale, fonction de *hachage*.**

Définition 3.2. *Une table de hachage est une structure qui permet de stocker des éléments sur un modèle* ***d'association clé − valeur***

Concrètement : c'est un "grand" tableau dont on définit
— l'ensemble des indices (les clés),
— une fonction qui à chaque objet associe une clé.
Pour retrouver un objet, on calcule sa clé et on "regarde" dans le tableau à l'indice en question.

Définition 3.3. *La fonction qui associe la clé à l'objet s'appelle* ***fonction de hachage*** *ou* ***hash function*** *en anglais.*

Exemple 3.4.
Reprenons l'exemple précédent mais cette fois E est un ensemble d'entier dont on ne connaît pas l'intervalle. **Quelles clés ? Quelle fonction de hachage ?**
On peut prendre les clés entre 1 et un entier k fixé à l'avance et choisir `%k` comme fonction de hachage.

Problème : comment faire quand deux objets ont la même clé ? Cela s'appelle une collision.
Solution : les cases de la table ne contiennent pas un simple objet mais la liste des objets associés à cette clé (stockée sous forme de tableau ou de liste chaînée).

3.3. **Complexité.** La complexité qui nous intéresse est celle de l'accès à un élément donné. Pour l'optimiser, il faut **éviter les collisions**.
— Premièrement, il faut que que la table soit grande par rapport au nombre d'entrées. On calcule le **facteur de compression** k/n où k est la taille de la table et n le nombre d'entrée. Si lorsque n augmente, k ne varie pas, on peut obtenir qu'une complexité linéaire pour l'accès aux éléments : la valeur de k ne fera varier que la constante. Pour optimiser réellement la complexité, il faut que la table soit *réallouée* lorsque le facteur augmente de la même façon que pour les tableaux.
— Par ailleurs, l'efficacité d'une table de hachage dépend grandement de sa fonction de hachage. Il faut que celle-ci distribue **uniformément** les éléments sur les clés pour éviter au maximum les collisions. Exemple pour les entiers : le modulo.

Lorsque l'implantation d'une table de hachage suit ces principes, on peut obtenir une complexité $O(1)$ **en moyenne**. Attention, la complexité du pire des cas sera toujours $O(n)$ (si tous les éléments ont la même valeur de hachage).

3.4. **En pratique.** Les tables de hachage sont une structure très largement utilisées dans de nombreux langages : `set` et `dict` python, `HashMap` Java. Il est très important d'en connaître le principe pour pouvoir utiliser au mieux ces objets.
Remarque : problème de la mutabilité des objets. cf demo.