Structures de données séquentielles

1 Introduction

1.1 Modèle vs. implémentation

Définition (*structure de données abstraite*) :

> Une structure de données abstraite est la description d'un ensemble de données et des opérations que l'on peut y appliquer.
>
> Elle est donc caractérisée par :
> - son type/format, qui répond à la question : "quel genre d'informations enregistre-t-on ?"
> - ses opérations, qui répondent à la question : "que peut on faire de ces informations ?"

Exemple : Un tableau d'éléments de type `t` est une structure de données abstraite. Elle prend la forme d'une suite finie d'éléments de type `t` et possède les opérations suivantes :
- créer un tableau d'une certaine taille
- accéder à la valeur de l'élément à la position i dans le tableau
- modifier la valeur de l'élément à la position i
- obtenir la taille du tableau

Remarquons bien que caractériser un tableau par un ensemble ne suffirait pas, puisqu'il manquerait les notions d'ordre et de multiplicité. On peut néanmoins munir les ensembles de la multiplicité, on obtient alors ce que l'on appelle des multi-ensembles.

Définition (*structure de données concrète*) :

> Une strcuture de données concrète est une implémentation d'une structure de données abstraite (...).

Exemple : L'implémentation suivante des tableaux en C est une structure concrète :

```
struct tableau{
    int taille;
    int* tab;
};
```

Remarque : La complexité des opérations d'une structure n'est donc fixée que pour une structure concrète, et dépendra pour les structures abstraites de leur implémentation.

Dans la suite, on dira tout simplement structure de données pour parler de structures de données abstraites.

1.2 Opérations

La description des opérations d'une structure de données ne se veut pas exhaustive ; au contraire, on ne décrit que les opérations élémentaires, qui elles, devront être implémentées par des fonctions

de faible complexité.

Exemple : Faire la somme des entiers d'un tableau d'éléments de type `int` n'est pas une opération élémentaire pour les tableaux statiques, on ne la précise donc pas.

S'il arrive qu'on fasse toutefois apparaître une opération qui aurait pu se décomposer en opérations plus simples déjà disponible, ce sera toujours pour un gain de complexité.

Exemple : Une opération d'initialisation (ou d'ajout, de concaténation...) d'une structure peut être plus efficace si l'on connaît à l'avance n valeurs à ajouter que si l'on ajoute n fois une valeur, en faisant les opérations d'ajout une par une.

Définitions (*types d'opérations dans une structure*) :

Pour une structure de données, on distingue trois sortes d'opérations :
- les constructeurs, qui permettent de créer et/ou d'initialiser la structure
- les accesseurs, qui permettent de récupérer des informations de la structure
- les transformateurs, qui permettent de modifier la structure (en changeant une valeur, ou même en modifiant la forme de la structure, par exemple pour ajouter ou supprimer une valeur).

Remarque : On ne considèrera pas de d'opération qui supprime la structure (destructeur).

Dans la suite de ce chapitre, on s'intéressera surtout à des structures séquentielles d'éléments homogènes. On fixe donc `t` un type pour les éléments.

2 Pile

2.1 Idée

La structure de pile fonctionne selon le principe "Last In, First Out" (ou "LIFO") : le premier dernier à rentrer dans la structure doit donc toujours être le premier à en sortir, et inversement le premier à rentrer sera le dernier à sortir.

Illustration : On peut se représenter cette structure avec l'image des piles d'assiettes :

2.2 Structure de données abstraite

Une pile représente une suite finie d'éléments de type `t` dont la taille peut varier (au cours du programme, si elle est implémentée). On présente ci-dessous ses opérations élémentaires :

pile
- pile_vide : () → pile[t]
- empiler : t, pile[t] → ∅
- dépiler : pile[t] → ∅
- sommet$^{(*)}$: pile[t] → t
- est_vide : pile[t] → booléen

Remarque : On ne peut pas demander la taille de la pile (c'est-à-dire on n'y a pas accès).

Remarque : $^{(*)}$Dans certaines définitions du type abstrait de pile, on peut trouver une fonction (souvent nommée pop) qui dépile renvoie de sommet *en le dépilant*, plutôt que d'uniquement le renvoyer : il s'agit donc à la fois d'un modificateur et d'un accesseur.

2.3 Implémentation par liste chaînée

Dans cette implémentation :

- la pile vide sera représentée par un pointeur de valeur `NULL`
- pour empiler, on effectuera un ajout en début de liste
- pour dépiler, on enregistrera `p->next` (la nouvelle première cellule) avant de faire `free(p)`.

cf. TP n°6 pour plus de détails.

2.4 Implémentation avec un tableau

On fait à présent l'hypothèse qu'il existe une taille limite $N_{\max} \in \mathbb{N}^*$ que la pile ne peut pas dépasser. Alors, il est possible d'effectuer une implémentation des piles par un tableau de taille $N_{\max}$:

Définition de la structure

```
struct s_pile{
    int nb_elem;
    // curseur donnant le sommet de la pile,
    // comme une sorte d'indice de fin
    t* tab;
};

typedef struct s_pile pile;
```

Fonction pile_vide

```
pile pile_vide (){
    pile p;
    p.nb_elem = 0;
    p.tab = (t*)malloc(sizeof(t)*Nmax);
    return p;
}
```

Fonction est_vide

```
bool est_vide (pile p){
    return p.nb_elem == 0;
}
```

Fonction empiler

```
void empiler (t val, pile* p){
    // hyp : (*p).nb_elem < Nmax
    if (p->nb_elem == Nmax){
        // message d'erreur
    }
    else{
        p->tab[p->nb_elem] = val;
        p->nb_elem++;
    }
}
```

Fonction dépiler

```
void depiler (pile* p){
    // hyp : !(est_vide(*p))
    p->nb_elem--;
}
```

Fonction sommet

```
t sommet(pile p){
    // hyp : !est_vide(p)
    return p.tab[p.nb_elem-1]
```

3 File

3.1 Idée

La structure de file utilise le principe “First In First Out” (“FIFO”), ou encore “Last In Last Out” (“LILO”) : le premier élément rentré est le premier à sortir, et le dernier rentré est le aussi le dernier à sortir.

Illustration : On peut imaginer une queue (une file) de personnes qui entrent dans une pièce par une porte et en ressortent par une autre :

3.2 Type abstrait

Voici les opérations du type abstrait de file :

file	• file_vide : () → file[t] • enfiler : t, file[t] → ∅ • défiler : file[t] → ∅ • tête$^{(*)}$: file[t] → t • est_vide : file[t] → booléen

3.3 Implémentation par liste chaînée

Il est possible d'utiliser des listes doublement chaînées pour implémenter le type abstrait file, mais on peut également se contenter d'une liste simplement chaînée avec :

- un pointeur vers la première cellule, pour défiler (toujours inclus dans une liste chaînée)
- un pointeur vers la dernière cellule, pour enfiler.

cf. TP n°6 pour les détails de l'implémentation.

3.4 Implémentation par tableau

Afin de réaliser une implémentation des files avec des tableaux, il nous faut encore une fois nous placer sous certaines hypothèses : le nombre d'insertions est au plus $N_{\max}$ où $N_{\max} \in \mathbb{N}^*$.

Définition de la structure

```
struct file{
    int debut;
    int fin;
    t* tab;
};
```

cf. TP n°6 pour le reste (définitions des fonctions réalisant les opérations élémentaires).

3.5 Implémentation avec deux piles

Le principe de l'implémentation des files avec deux piles est le suivant :

- on se donne deux piles, p_1 et p_2
- pour enfiler un élément, on l'empile sur p_1
- pour défiler, on dépile l'élément au sommet de p_2 lorsque celle-ci est non vide, et dans le cas contraire, on renverse p_1 puis on transvase le contenu renversé dans p_2 afin de pouvoir y dépiler l'élément à défiler.

On illustre ci-dessous cette implémentation :

On donne maintenant le pseudo-code des opérations élémentaires :

Pseudo-code – file_vide () → file

```
F.p1 = pile_vide()
F.p2 = pile_vide()
Renvoyer F
```

Pseudo-code – *F*.est_vide () → booléen

```
Retourner (F.p1.est_vide et F.p2.est_vide)
```

Pseudo-code – *F*.tête () → `elem`

```
hyp : F est non vide
Si F.p2.est_vide() alors
    Tant que (non F.p1.est_vide)
        F.p2.empile(F.p1.sommet)
        F.p1.dépile()
Renvoyer F.p2.sommet()
```

Pseudo-code – *F*.enfile (`elem` e) → ()

```
F.p1.empile(e)
```

Pseudo-code – *F*.défile () → ()

```
hyp : F est non vide
Si F.p2.est_vide() alors
    Tant que (non F.p1.est_vide)
        F.p2.empile(F.p1.sommet)
        F.p1.dépiler()
F.p2.dépile()
```

Remarque : On a ici utilisé ce qu'on appelle la notation objet pour les fonctions.

4 Listes

4.1 Définition

Définition (*liste*) :

> Une liste est une structure de données abstraite permettant de stocker une suite finie d'éléments (munis d'un ordre et d'une multiplicité), dans laquelle on peut réaliser des insertions et suppressions en début, milieu ou fin de la séquence, et que l'on peut parcourir.

La liste doit être munie des opérations suivantes :

liste	constructeur
	• liste_vide : () → liste
	accesseurs
	• L.est_vide : () → booléen
	• L.début : () → place$^{(*)}$
	• L.suivant : place p → place
	• L.contenu : place p → `elem`

- L.est_dernier : place $p \to$ booléen

transformateurs
- L.ajoute_après : place p, `elem` $e \to \varnothing$
- L.ajoute_début : `elem` $e \to \varnothing$
- L.supprime : place $p \to \varnothing$

Remarque : (*)Les positions sont ici à voir comme des pointeurs, et non pas des numéros.

4.2 Implémentation par listes doublement chaînées

L'implémentation par tableau est possible mais elle est restrictive (la taille de la liste est bornée) et de plus, l'insertion et la suppression d'éléments serait en temps linéaire.

On préfèrera donc implémenter les listes à l'aide de listes doublement chaînées : chaque cellule possède, en plus d'un champ contenant un élément de type `elem` et du pointeur vers la cellule suivante, un pointeur vers la cellule précédente pour remonter en arrière.

cf. TP n°6 pour l'implémentation.

4.3 Fonctions non élémentaires

On propose ci-dessous le pseudo-code de deux fonctions non élémentaires pour la structure, mais néanmoins très utiles lorsque l'on manipule des listes :
- L.est_présent, qui teste si un élément e est présent dans L
- L.première_place, qui retourne la première place (occurence) de e dans L en supposant qu'elle existe

Pseudo-code – *L*.est_présent (`elem` e) $\to$ booléen

```
Si L.est_vide() alors
    renvoyer Faux
Sinon
    place p = L.début
    Tant que (non L.est_dernier(p))
        Si L.contenu(p) = e alors
            renvoyer Vrai
        Sinon
            p ← L.suivant(p)
    Renvoyer (L.contenu(p) = e)
```

Pseudo-code – *L*.première_place (`elem` e) $\to$ place

```
hyp : L.est_présent(e)
place p = L.début
Tant que (non L.est_dernier(p))
    Si L.contenu(p) = e alors
        renvoyer p
    Sinon
        p ← L.suivant(p)
Renvoyer p
```

5 Complexité amortie

Pour une structure de données donnée et pour une opération donnée sur cette structure, considérer la complexité pire cas de cette opération n'est pas toujours pertinent puisque le coût du pire cas induit celui des opérations qui améliorent l'état de la structure.

Pour pallier ce manque de représentativité, nous étudierons la complexité amortie, c'est-à-dire la complexité globale de plusieurs opérations successives.

> **Remarque :** Il faut distinguer la complexité amortie, qui est une moyenne sur des appels successifs, de la complexité dite moyenne, qui correspond plutôt à une moyenne sur les différentes instances possibles pour un seul appel.

5.1 Table dynamique

La table dynamique est un exemple illustrant la nécessité de considérer la complexité amortie pour estimer fidèlement le coût d'une opération. Dans cette section, nous définissons cette structure et faisons une étude précise de complexité amortie pour l'une de ses opérations.

La table dynamique, contrairement aux tableaux statiques, est une structure dont la taille peut varier : on peut ajouter des éléments tant qu'il reste de la place, et une fois que la table est remplie, la prochaine insertion fera doubler sa taille en vue d'accueillir des nouveaux éléments par la suite.

table dynamique

constructeur
- crée_case : $() \to$ table dynamique

accesseurs
- T.élément : `int` $i \to$ `elem`
- T.nb_elem : $() \to$ `int`

transformateurs
- T.modifie : `int` i, `elem` $e \to \varnothing$
- T.ajoutefin : `elem` $e \to \varnothing$
- T.retirefin : $() \to \varnothing$

Soit $N \in \mathbb{N}^*$. On considère la suite d'appels ci-dessous :

```
T = crée_case()
Pour i allant de 1 à N
    T.ajoutefin(i)
```

Pour $j \in [1..N]$, on note σ_j le coût du j-ième appel à ajoutefin.
Alors, on peut représenter comme ci-dessous l'état (taille, cases vides et pleines) de la table T à chaque appel, c'est-à-dire à chaque tour de la boucle "Pour" :

Propriété :

On en déduit alors : $\forall\, j \in [1..N],\ \sigma_j = \begin{cases} 1 \text{ si } j-1 \notin \{2^k \mid k \in \mathbb{N}\} \\ 2+2^k \text{ s'il existe } k \in \mathbb{N} \text{ tel que } j-1 = 2^k \end{cases}$

soit encore, en changeant d'indice : $\forall\, j' \in [0..N-1],\ \sigma_{j'+1} = \begin{cases} 2+2^k \text{ si } j' = 2^k \text{ pour un } k \in \mathbb{N} \\ 1 \text{ sinon} \end{cases}$.

Le coût total vaut alors : $S_N = \sum_{j=1}^{N} \sigma_j = \sum_{j=0}^{N-1} \sigma_{j+1}$.

Notons $D = 2^{\lfloor \log_2(N-1) \rfloor}$ alors, D est la plus grande puissance de 2 inférieure ou égale à $N-1$. Ainsi, $\forall\, j \in [D+1, N]$, $\sigma^{j+1} = 1$ car j n'est pas une puissance de 2. En notant $p = \log_2(D)$, on a :

$$\begin{aligned}
S_N &= \sum_{j=0}^{D-1} \sigma_{j+1} + \sigma_{D+1} + \sum_{j=D+1}^{N-1} \sigma_{j+1} \\
&= \left(\underbrace{\sigma_1}_{=1} + \sum_{k=0}^{p-1} \sum_{j=2^k}^{2^{k+1}-1} \sigma_{j+1} \right) + \underbrace{\sigma_{D+1}}_{=2+2^p} + (N - 1 - (D+1) + 1) \\
&= (3 + D) + (N - D - 1) + \sum_{k=0}^{p-1} \left(\sigma_{2^k+1} + \sum_{j=2^k+1}^{2^{k+1}-1} \sigma_{j+1} \right) \\
&= N + 2 + \sum_{k=0}^{p-1} (2^k + 2) + (2^{k+1} - 2^k - 1) \\
&= N + 2 + \sum_{k=0}^{p-1} 2^{k+1} + 1 \\
&= N + p + 2 \sum_{k=0}^{p-1} 2^k \\
&= N + 2 + p + 2(2^p - 1) \\
&= N + p + 2D \\
&= N + \lfloor \log_2(N-1) \rfloor + 2^{\lfloor \log_2(N-1) \rfloor + 1}
\end{aligned}$$

Or, $\lfloor \log_2(N-1) \rfloor \in \Theta(\ln(N))$ et $2^{\lfloor \log_2(N-1) \rfloor + 1} \in \Theta(N)$.

Propriété :

On a donc finalement $S_N \in \Theta(N)$: la complexité amortie d'une succession opérations d'ajout dans une table dynamique est linéaire en le nombre d'ajouts N.