Contents

Cours 1.

Les '==' dans le PDF sont à ignorer pour l'instant, ils permettent du surlignage en markdown.

Introduction

Objectif principal : Comment définir des structures de données et des algorithmes pour les manipuler efficacement ?

→ Une structure de données est un objet qui permet de stocker des informations et de faire des requêtes sur ces informations.

Exemple

Un éditeur de texte : manipuler une chaine de caractères.

Note : on ne parle pas du rendu visuel

Définition abstraite :

- Insérer une chaîne de caractères à un endroit
- Supprimer une chaîne de caractères à un endroit
- Rechercher une chaîne de caractères dans un texte
- Rechercher une chaîne de caractères / un sous mot dans un texte
- Mise en page visuelle
  - Chaque caractère a un certain nombre d'attributs (couleur, police, taille, etc.)

Lors de l'étape de la programmation il faut que cette définition abstraite devienne concrète dans la machine en utilisant les constructions du langage de programmation. Tout cela va constituer la structure de données de l'éditeur de texte : c'est **l'implémentation**.

Détails Une manière de faire :

- structure en `C` pour représenter un caractère et ses attributs
- l'objet pour stocker les caractères est un tableau de caractères

Structure de données concrète pour éditeur de texte ici sera : ce tableau ainsi que les fonctions qui permettent de manipuler ce tableau (`lire`, `ecrire`, `inserer`, `supprimer`, `rechercher`, etc.).

En java, le constructeur classe encapsule les différentes fonctions manipulant le tableau de caractères.

Remarque

- Une modification de caractère revient à supprimer un caractère et à insérer un autre caractère. Donc la question de la complétude de la structure de données des opérations proposées se pose pour chaque chaque structure de données.

Complétude : Est-ce que tout les comportements voulus sont définis par les opérations proposées ?

- Pour être sûr que la fonction implémentée est correcte, on décrit un "bon comportement" en langage mathématique. On peut ensuite vérifier que la fonction implémentée correspond bien à ce comportement à l'aide d'une preuve (exemple : récurrence). Ensuite, pour chaque implémentation, on montre que l'implémentation respecte la spécification.

Retour sur les détails Est-ce que c'est une bonne implémentation ?

- Bonne par rapport à quoi ? On définit des objetifs (par exemple une fonction associant à chaque implémentation une valeuret notre objectif sera une certaine valeur).
  - Exemple complexité en temps (grosso modo temps d'execution)
  - Complexité en mémoire (espace en mémoire auxiliaire utilisée)

Temps d'execution	Mémoire utilisée
`ajouter` / `supprimer` (on va devoir décaler les caractères suivants)	Mémoire ? Comment on gère ? Si on utilise la structure de données avec tableau dynamique, c'est un peux mieux

Temps d'execution	Mémoire utilisée
La `recherche` est lente : on ne peut pas utiliser d'algorithme complexe (le texte n'est pas forcément trié)	

Langage de description

- Langage usuel pour décrire des structure de données avec de temps en temps des notations mathématiques ou constructions mathématiques (que vous avez déjà vues)
- Pour expliquer une implémentation d'une structure de données :
  - La description d'étapes élémentaires (basée sur le cours d'Alhorithmique)
  - De temps en temps du pseudo-code est équivalent au mélange de `C` et de pseudo-code vu en L1.

Chapitre 1 : Algorithmique, Types, Valeurs

- **Algorithme** : Suite d'opérations en vue de résoudre un prolbème.
  - Un algorithme a des entrées, et produit une sortie (réponse à la question)
  - Un algorithme manipule des données
  - Chaque opération est définie par un nom et une description
    - Ces **données** on les classifie par rapport aux opérations possibles : c'est ce que l'on appelle le typage
    - L'encodage des données produit un ensemble de **valeurs**

Cours 2.

On va classifier les types : chaque type est identifié par un ensemble de valeurs, et l'ensemble des opérations possibles sur les valeurs. Une classe sera un type.

- Une constante est une valeur particulière pour ce type.
- Une variable va représenter des valeurs possibles pour ce type.
  - Faire la distinction entre la variable et le contenu (la variable est l'ensemble des valeurs possibles pour ce type, le contenu est une valeur de ce type)

Les types

Il y a deux types de types :

- Les ==types primitifs==
- Les ==types composés==

Types primitifs

Ils sont non décomposables et fournis par défaut. Dans ce cas les type primitifs sont entiers, réels, carctères, booléens, chaînes de caractères.

Types composés

Ils sont construits à partir d'autres types : 3 constructeurs de types :

- *Produit* : $T_1, T_2, ..., T_m$ sont des types, alors $T_1 \times T_2 \times ... \times T_m$ est un type avec comme valeurs les couples $(v_1, v_2, ..., v_m)$ où v_i est une valeur de T_i.
- *Somme* : $T_1, T_2, ..., T_m$ sont des types, alors $T_1 \oplus T_2 \oplus ... \oplus T_m$ est un type avec comme valeurs les couples $(v_1, v_2, ..., v_m)$ où v_i est une valeur de T_i.
  - **Exemple** : les constructeurs `union` en C
- *Enregistrements* : chaque valeur d'un enregistrement est composé de plusieurs entités, appelées **champs**, chacun ayant un identifiant, un type et une valeur.
  - **Exemple** : les constructeurs `struct` en `C`, ou encore `class` en `Java`
- *Constructeur de tableau* : Un tableau est un ensemble contigu de valeurs où on a accès en temps constant à chaque valeur. Dans la plupart des implémentations de tableau toute les valeurs ont le même type.
  - Pour réserver de la place mémoire pour un tableau de taille n, et la taille d'une valeur c'est p, on réserve $n \times p$ octets bits. Si adresse de début c'est adr_0, alors pour accéder à l'indice $0 \leq i \leq n-1$:

$$i \times p + adr_0 \text{ (opération constante)}$$

Chapitre 2 : Types de données abstrait

Définition 2.1 : Types de données abstrait

Un ==type de données abstrait (TDA)== est un type de données composé, dont on définit :

- Une **signature** : un identifiant, et type de retour / paramètres de chaque opération et d'un ensemble de types prédéfinis à utiliser.
- Une **liste d'axiomes** qui vont définir le comportement des opérations sur des valeurs du TDA.

Exemple

On choisit les entiers. - Opérations, addition, soustraction prend deux entiers en paramètres et renvoie un entier.

Pour avoir des ==types de données concrètes (TDC)==, on propose une façon de représenter dans la machine une TDA, en proposant la façon de manipuler

cette représentation à travers les opérations du TDA. C'est ce qu'on appelle une ==implémentation d'un TDA==.

Exemple

- Un TDC : on choisit de représenter les entiers par des nombres binaires en représentation complément à deux. On va donc implémenter les opérations de base sur les entiers.
- La définition d'un `String` en `C` : on choisit de représenter les chaînes de caractères par des tableaux de caractères. On va donc implémenter les opérations de base sur les chaînes de caractères. C'est un TDA.

2.1 : TDA Pile

Le ==TDA Pile== est apparu lorsque l'on a eu besoin de structures, les programmes en blocs que l'on peut référencer (bloc pouvant se référencer).

Principale question : comment fournir des données aux blocs référencés et ensuite retourner au bloc appelant ? C'est représenter les références aux blocs dans un SDD (Structure de Données Dynamique) ayant le *même comportement qu'une pile d'assiette.*

Définition 2.2 : Pile

Les ==différentes opérations== sur une pile sont :

- `creerPile`: $() \rightarrow$ `Pile_T` : Créer une pile vide
- `estVidePile`: `Pile_T` $\rightarrow$ `Booléen` : Vérifie si la pile est vide
- `empiler`: `Pile_T` $\times$ `T` $\rightarrow$ `Pile_T` : Empile un élément sur la pile
- `depiler`: `Pile_T` $\rightarrow$ `Pile_T` : Dépile un élément de la pile
- `sommetPile`: `Pile_T` $\rightarrow$ `T` : Renvoie le sommet de la pile

==Axiomes== :

- `estVidePile(creerPile())= True`
- `estVidePile(empiler(p, x))= False`
- `sommetPile(empiler(p, x))= x`
- `depiler(empiler(p, x))= p`

Le TDA `T` est utilisé pour dire que la pile on l'utilise pour stocker que des objets du même type. On ne fait aucune supposition sur `T`, à part les valeurs de T existent.

Implémentation possible On utilise un tableau pour stocker les valeurs et un entier qui va pointer sur l'indice du sommet de pile.

En `C` :

```c
struct pile {
    int T[MAX];
```

```
    int sommetPile;
};
```

Ici, `sommetPile` permet de savoir où l'on doit poser le sommet de la pile. On peut donc empiler et dépiler en temps constant (au lieu de chercher l'élément le plus en bas de la pile).

Le TDA File

Le besoin : gestion des accès à des ressources limitées ou pour éviter les accès concurrents.

==**Paradigme**== : Premier arrivé, premier servi.

Description du TDA Le TDA `File_T` est un TDA composé de :

- `creerFile`: () → `File_T` : Créer une file vide
- `estVideFile`: `File_T` → `Booléen` : Vérifie si la file est vide
- `enfiler`: `File_T` × `T` → `File_T` : Enfile un élément dans la file
- `defiler`: `File_T` → `File_T` : Défile un élément de la file
- `teteFile`: `File_T` → `T` : Renvoie le premier élément de la file

==Axiomes== :

- `estVideFile(creerFile())= True`
- `estVideFile(enfiler(f, x))= False`
- `teteFile(enfiler(f, x))= x`
- `defiler(enfiler(f, x))= f`
- `defiler(enfiler(enfiler(f, x), y))= enfiler(f, y)`

Implémentation possible On peut représenter une file par un tableau et deux entiers :

- Tête qui point sur l'indice de la tête de file
- Queue qui pointe sur l'indice de la queue de la file

En `C` :

```
struct file {
    int T[MAX];
    int tete;
    int queue;
};
```

Pour un fil, comme c'est un ensemble ordonné : on doit toujours connaître l'ordre de la file, ce qui va permettre de défiler et d'enfiler en temps constant.

```
digraph G {
    rankdir=LR;
    node [shape=record];
    a [label="{ <data> 1 : Tête | <ref>  }", width=1.2]
    b [label="{ <data> 2 | <ref>  }", width=1.2]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    a:ref -> b
    b:ref -> c
    c:ref -> d

}
```

On défile la tête de la file, on décale la tête de la file vers la droite.

```
digraph G {
    rankdir=LR;
    node [shape=record];
    a [label="{ <data> 1 : Tête | <ref>  }", width=1.2, color="red",
        fontcolor="red"]
    b [label="{ <data> 2 | <ref>  }", width=1.2]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    a:ref -> b
    b:ref -> c
    c:ref -> d

}
```

On obtient donc :

```
digraph G {
    rankdir=LR;
    node [shape=record];
    b [label="{ <data> 2 : Nouvelle tête | <ref>  }", width=1.2,
        color="darkgreen", fontcolor="darkgreen"]
    c [label="{ <data> 3 | <ref>  }", width=1.2]
    d [label="{ <data> 4 : Queue | <ref>  }", width=1.2]

    b:ref -> c
    c:ref -> d

}
```

La file est stockée dans un intervalle de tableau, on peut donc décaler la tête de la file en temps constant.

Par contre on n'arrive pas à distinguer File vide de tableau Plein.

→ Ce n'est pas insurmontable :

- Compter le nombre d'éléments
- Garde une case vide entre queue et tête
- …