La notion de Type Abstrait de Données (TAD)

1. Définition « formelle »

- Pour concevoir un algorithme complexe, on adopte une **démarche descendante** : l'analyse procède par affinements successifs.
- Au départ, on reste **loin de toute implémentation** ; en particulier, la représentation concrète des données n'est pas fixée. On parle alors de **type abstrait de données**.
- On se donne une ***notation*** qui **décrit** les données, des **opérations applicables** à ces données (***primitives***), et les **propriétés** de ces opérations (***sémantique***).

Exemple du type entier

- ***Notation :*** suite de chiffres décimaux, éventuellement précédée d'un signe – ou +
- ***Primitives :*** op. arithm. (+, *, /, -, mod, div)
- ***Sémantique :*** sens habituel

⇒ On ne se soucie pas de la représentation d'un entier : binaire par complément à 2, …

Exemple du type booléen

- ***Notation :*** Vrai, Faux (par exemple)
- ***Primitives :*** opérateurs OU, ET, NON, etc…
- ***Sémantique :*** sens habituel

- Représentation ? peu importe…
  1 octet en Pascal et C++, 1 bit en Java, rien de prévu en C, …

Remarques

- Les types de données et constructeurs de types déjà rencontrés sont en fait des TAD…

- L'implémentation des TAD sera abordée ultérieurement… important et délicat…

  contrainte : respecter les spécifications abstraites sans négliger l'efficacité

2. Définition « plus pratique »

- **Pourquoi** avoir recours à cette notion nouvelle de **type abstrait** ?
  - elle permet de définir des types de données non « primitifs », c'est-à-dire non disponibles (non déjà implémentés) dans les langages de programmation courants.
- *Exemple*

  type Date : notation jj/mm/aa

```
//base
Date()  // crée un objet Date non initialisé
Date(int j, int m, int a)

// consultation
int getJour() const
int getMois() const
int getAnnee() const
```

```
//modification
void setJour(int j)
etc...

//relations
bool operator<=(const Date & autreDate)
   const
bool operator>=(const Date & autreDate)
   const
bool operator==(const Date & autreDate)
   const

//services
void lendemain()
int intervalle(const Date & autreDate) const
```

Exercices

- Ecrire les opérateurs <=, >= et ==
- Ecrire une méthode qui modifie la date et la met au lendemain.

Abstraits en 4 types de structures :

- *les structures séquentielles* : les **listes**, et leurs cas particuliers que sont les **piles** et les **files**.
- *les structures arborescentes* : les **arbres** (binaires ou généraux).
- *les structures relationnelles* : les **graphes**.
- *structures à accès par clé* : les **tables**.

Structures séquentielles

Les **Listes** et leurs cas particuliers :
les **Piles** et les **Files**

Les Listes

- **Liste linéaire** : forme la plus commune d'organisation de données devant être traitées séquentiellement…
- **Liste linéaire** : suite d'éléments d'un même type, chacun possédant un rang.
- Parcours d'une liste : séquentiel.
- Une liste est évolutive : ajout/suppression de n'importe quel élément.
- **Pile** et **File** = Listes avec restriction sur l'extrémité où peuvent être réalisés les opérations ajout/suppression.
  Elles jouent un rôle important en informatique.

TAD Pile

Structure **LIFO**

(**L**ast **I**n **F**irst **O**ut)

Description

- Liste dans laquelle les ajouts et suppressions n'ont lieu que sur une même extrémité appelée ***sommet de pile***.
- *Exemple* : pile d'assiettes, piles de livres, …

- **Structure LIFO** : le dernier élément entré (Last In) est le premier sorti (First Out).

Primitives

TPile = **Pile** de Tinfo // n'importe quel type

Pile<Tinfo> ()
bool **pileVide** ()
TInfo **valeurSommet** ()
void **empiler** (Tinfo Elem)
void **depiler** ()

Axiomatique

axiomatique : les propriétés des primitives

- **TPile** : initialise une pile à vide doit être appelée avant toute utilisation d'une pile.
- **pileVide**, **valeurSommet** , **empiler** et **depiler** ne sont pas définies sur une pile dont la valeur est indéterminée.
- **Attention : valeurSommet** et **depiler** ne sont pas définies sur une pile vide.

Principales utilisations

- ***traitement des appels de fonctions***: gestion des adresses de retour, nécessaire dans le cas de fonctions récursives…

- ***évaluation d'expressions arithmétiques :*** les algorithmes utilisent la structure de Pile.

Exercice 1

- Récupérer les fichiers Pile.cxx; Pile.h et Makefile à l'adresse :

www.labri.fr/perso/bourqui/downloads/cours/AP2/TPs/_7/source/

- Ecrire une fonction qui affiche une pile d'entier

  void affiche(Pile<int> p)

- Écrire une fonction qui compte le nombre d’éléments d’une pile donnée.

  int nbElements(Pile<int> p)

Exercice 2

- Écrire une action qui crée un clone d’une pile donnée.

```
void clonerPile(Pile<int> p,
                Pile<int> & p_clone)
```

Exercice 3

- Écrire une fonction qui inverse une pile.

```
void inverserPile(Pile<int> &p)
```

Exercice 4

- Écrire une action qui supprime les entiers négatifs d'une pile (l'ordre des éléments positifs doit rester inchangé).

  void supprimeNegatif(Pile<int> &p)