Livret - 6

Types abstraits et données

TAD, liste, pile, file.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

6 : Type abstraits et structures de données

Plan du chapitre:

Introduction

Le mécanisme de l'abstraction est l'un des **plus** important avec celui de la méthode structurée fonctionnelle en vue de la réalisation des programmes.

Notion d'abstraction en informatique

L'abstraction consiste à penser à un objet en termes d'actions que l'on peut effectuer sur lui, et non pas en termes de représentation et d'implantation de cet objet.

C'est cette attitude qui a conduit notre société aux grandes réalisations modernes. C'est un art de l'ingénieur et l'informatique est une science de l'ingénieur.

Dès le début de la programmation nous avons vu apparaître dans les langages de programmation les notions de **subroutine** puis de **procédure** et de **fonction** qui sont une abstraction d'encapsulation pour les familles d'instructions structurées:

- Les paramètres formels sont une **abstraction** fonctionnelle (comme des variables muettes en mathématiques),
- Les paramètres effectifs au moment de l'appel sont des **instanciations** de ces paramètres formels (une implantation particulière).

L'idée de considérer les **types** de données comme une abstraction date des années 80. On s'est en effet aperçu qu'il était nécessaire de s'abstraire de la représentation ainsi que pour l'abstraction fonctionnelle. On a vu apparaître depuis une vingtaine d'année un domaine de recherche : celui des **spécifications algébriques**. Cette recherche a donné naissance au concept de **Type Abstrait Algébrique** (TAA).

Selon ce point de vue une structure de donnée devient:

Une collection d'informations structurées et reliées entre elles selon un graphe relationnel établi grâce aux opérations effectuées sur ces données.

Nous spécifions d'une façon simple ces structures de données selon deux niveaux d'abstraction, du plus abstrait au plus concret.

Une spécification abstraite :

Description des propriétés générales et des opérations qui décrivent la structure de données.

Une spécification opérationnelle :

Description d'une forme d'implantation informatique choisie pour représenter et décrire la structure de donnée. Nous la divisons en deux étapes : la spécification opérationnelle concrète (choix d'une structure informatique classique) et la spécification opérationnelle d'implantation (choix de la description dans le langage de programmation).

Remarque :

Pour une spécification abstraite fixée nous pouvons définir plusieurs spécifications opérationnelles différentes.

1. La notion de TAD (Type Abstrait de Données)

Bien que nous situant au niveau débutant il nous est possible d'utiliser sans effort théorique et mental compliqué, une méthode de spécification semi-formalisée des données. Le " type abstrait de donnée " basé sur le type abstrait algébrique est une telle méthode.
Le lecteur ne souhaitant pas aborder le formalisme mathématique peut sans encombre pour la suite, sauter le paragraphe qui suit et ne retenir que le point de vue pratique de la syntaxe d'un TAA.

1.1 Le Type Abstrait Algébrique (TAA)

Dans ce paragraphe nous donnons quelques indications théoriques sur le support formel algébrique de la notion de TAA. *(notations de F.H.Raymond cf.Biblio)*

Notion d'algèbre formelle informatique

Soit (Fn) $n \in \mathbf{N}$ **,** une famille d'ensembles tels que :

$\mathbf{(}\exists i_0 \in \mathbf{N}\ (1 \leq i_0 \leq n) / Fi_0 \neq \varnothing \mathbf{)} \wedge \mathbf{(}\forall i, \forall j,\ i \neq j \Rightarrow F_i \cap F_j = \varnothing \mathbf{)}$
posons : $\mathbf{I} = \{ n \in \mathbf{N} / \mathbf{F}_n \neq \varnothing \}$

Vocabulaire :

Les symboles ou éléments de F_0 sont appelés symboles de constantes ou symboles fonctionnels 0-aire.

Les symboles de F_n (où $n \geq 1$ et $n \in \mathbf{I}$) sont appelés symboles fonctionnels n-aires.

Notation :

$$F = \bigcup_{n \in N} Fn = \bigcup_{n \in \mathbf{I}} Fn$$

soit F* l'ensemble des expressions sur F, **le couple (F^*,F) est appelé une algèbre abstraite**.

On définit pour tout symbole fonctionnel n-aire f, une application de F^*_n dans F^* notée "$\hat{f}$" de la façon suivante :

$\forall$ e, ($f \in F_n \rightarrow \hat{f}$)

et { $\hat{f}$: $F^{*n}_{\bullet} \rightarrow F^*$ telle que $(a_1,..., a_n) \rightarrow \hat{f}(a1,...,an) = f(a1,...,an)$ }

On dénote :

$F_n = \{ \hat{f} / f \in Fn \}$ et $\hat{F} = \bigcup_{n \in I} Fn$

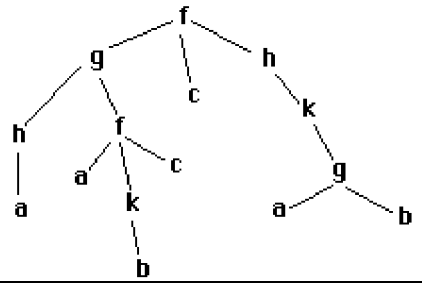
le couple $(F^*, \hat{F})$ est appelé une algèbre formelle informatique (AFI).

Les expressions de F^* construites à partir des fonctions $\hat{f}$ sur des symboles fonctionnels n-aires s'appellent des schémas fonctionnels.

Par définition, les schémas fonctionnels de F_0 sont appelés les constantes.

Exemple :
F0 = {a,b,c} // *les constantes*
F1 = {h,k} // *les symboles unaires*
F2 = {g} // *les symboles binaires*
F3 = {f} // *les symboles ternaires*

Soit le schéma fonctionnel : **fghafakbcchkgab**
(chemin aborescent abstrait non parenthésé)

Ce schéma peut aussi s'écrire avec un parenthèsage :	ou encore avec une représentation arborescente:
f [g(h(a),f(a,k(b),c)), c, h(k(g(a,b)))]	

Interprétation d'une algèbre formelle informatique

Soit une algèbre formelle informatique (AFI) : **(** $F^*, \hat{F}$ **)**

- on se donne un ensemble **X** tel que **X** $\neq \varnothing$,
- **X** est muni d'un ensemble d'opérations sur X noté Ω,

L'on construit une fonction ψ telle que :

$\psi : (F^*, \hat{F}) \rightarrow \mathbf{X}$ ayant les propriétés d'un homomorphisme

ψ est appelée fonction d'interprétation de l'AFI.
X est appelée l'univers de l'AFI.

Une AFI est alors un modèle abstrait pour toute une famille d'éléments fonctionnels, il suffit de changer le modèle d'interprétation pour implanter une structure de données spécifique.

Exemple :

$F_0 = \{x, y\}$ une AFI
$F_2 = \{f, g\}$
$F = F_0 \cup F_2$

l'Univers : **X** = **R** *(*les nombres réels)

les opérateurs Ω = {+,*} (addition et multiplication)

l'interprétation $y : (F^*, \hat{F}) \rightarrow (\mathbf{R}, \Omega)$
définie comme suit :

$\psi(f) : \mathbf{R}^2 \rightarrow \mathbf{R}$
$\psi(f) : (a,b) \rightarrow \psi(f)\ [a,b] = a+b$

$\psi(g) : \mathbf{R}^2 \rightarrow \mathbf{R}$
$\psi(g) : (a,b) \rightarrow \psi(g)\ [a,b] = a*b$

$\psi(x) = a_0$ (avec $a_0 \in$ **R** fixé interprétant la constante x)
$\psi(y) = a_1$ (avec $a_1 \in$ **R** fixé interprétant la constante y)

Soit le schéma fonctionnel **fxgyx,** son interprétation dans ce cas est la suivante :

$\psi(fxgyx) = \psi(f\ (x,g(y,x))) = \psi(f)\ (\psi(x),\psi(g)[\psi(y),\psi(x)])$
$= \psi(x) + \psi(g)[\psi(y),\psi(x)]$ ⇐ *propriété de ψ(f)*
$= \psi(x) + \psi(y)*\psi(x)$ ⇐ *propriété de ψ(g)*

Ce qui donne comme résultat : ψ (fxgyx) = $a_0 + a_1 * a_0$

A partir de la même AFI, il est possible de définir une autre fonction d'interprétation ψ'et un autre Univers **X**'.

par exemple :
l'Univers : **X = N** *(les entiers naturels)*
les opérateurs : Ω = **{ reste** , ≥ **}** *(le reste de la division euclidienne et la relation d'ordre)*

La fonction ψ'est définie comme suit :

ψ'(g) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(g) : (a,b) → ψ'(g)[a,b] = **reste**(a,b)

ψ'(f) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(f) : (a,b) → ψ'(f)[a,b] = a ≥ b ψ'(x) = n_0 (avec $n_0 \in$ **N** fixé)
ψ'(y) = n_1 (avec $n_1 \in$ **N fixé**)

On interprète alors le même schéma fonctionnel dans ce nouveau cas **fxgyx** :
ψ'(**fxgyx**) = $n_0 \geq$ **reste**(n_1, n_0)

Ceci n'est qu'une interprétation. cette interprétation reste encore une abstraction de plus bas niveau, le sens (sémantique d'exécution), s'il y en a un, sera donné lors de l'implantation de ces éléments. Nous allons définir un outil informatique se servant de ces notions d'AFI et d'interprétation, il s'agit du type abstrait algébrique.

Un TAA (type abstrait algébrique) est alors la donnée du triplet :

- une AFI
- un univers **X** et Ω
- une fonction d'interprétation ψ

la syntaxe du TAA est définie par l'**AFI** et l'ensemble **X**
la sémantique du TAA est définie par ψ et l'ensemble Ω

Notre objectif étant de rester pratique, nous arrêterons ici la description théorique des TAA (compléments cités dans la bibliographie pour le lecteur intéressé).

1.2 Disposition pratique d'un TAA

on écrira (exemple fictif):

Sorte : A, B, C ..... *les noms de types définis par le TAA, ce sont les types au sens des langages de programmation.*

Opérations :

f : A x B → B
g : A → C
x : → B *(notation pour les symboles de constantes de F0)*
y : → B *(notation pour les symboles de constantes de F0)*

Cette partie qui décrit la syntaxe du TAA s'appelle aussi **la signature du TAA** .

La sémantique est donnée par ψ , Ω sous la forme d'axiomes et de préconditions.

Le domaine d'une opération définie partiellement est défini par une précondition.

Un TAA réutilise des TAA déjà définis, sous forme de **hiérarchie**. Dans ce cas, la signature totale est la réunion des signatures de tous les TAA.

Si des opérateurs utilisent le même symbole, le problème de **surcharge** peut être résolu sans difficulté, parce que les opérateurs sont définis par leur ensembles de définitions.

SYNTAXE DE L'ECRITURE D'UN TYPE ABSTRAIT ALGEBRIQUE :

sorte : .....
utilise : .....
opérations :
préconditions :
..... **def_ssi** .....
axiomes :

FinTAA

Exemple d'écriture d'un TAA (les booléens) :

sorte : Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
axiomes :

¬ (V) = F ; ¬ (F) = V
a ∧ V = a ; a ∧ F = F
a ∨ V = V ; a ∨ F = a

FinBooléens

1.3 Le Type Abstrait de Donnée (TAD)

Dans la suite du document les TAA ne seront pas utilisés entièrement, la partie axiomes étant occultée. Seules les parties opérations et préconditions sont étudiées en vue de leur implantation.
C'est cette restriction d'un TAA que nous appellerons un type abstrait de données (TAD).
Nous allons fournir dans les paragraphes suivants quelques Types Abstrait de Données différents.

Nous écrirons ainsi par la suite un TAD selon la syntaxe suivante :

TAD Truc

utilise : .....
Champs : .....
opérations : .....
préconditions : .....

FinTAD Truc

Le TAD Booléens s'écrit à partir du TAA Booléens :

TAD Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
FinTAD Booléen

Nous remarquons que cet outil permet de spécifier des structures de données d'une manière générale sans avoir la nécessité d'en connaître l'implantation, ce qui est une caractéristique de la notion d'abstraction.

1.4 Classification hiérarchique

Nous situons, dans notre approche pédagogique de la notion d'abstraction, les TAD au sommet de la hiérarchie informationnelle :

HIERARCHIE INFORMATIONNELLE

- **1°** TYPES ABSTRAITS (les TAA,...)
- **2°** CLASSES / OBJETS
- **3°** MODULES
- **4°** FAMILLES de PROCEDURES et FONCTIONS
- **5°** ROUTINES (procédures ou fonctions)
- **6°** INSTRUCTIONS STRUCTUREES ou COMPOSEES
- **7°** INSTRUCTIONS SIMPLES (langage évolué)
- **8°** MACRO-ASSEMBLEUR
- **9°** ASSEMBLEUR (langage symbolique)
- **10°** INSTRUCTIONS MACHINE (binaire)

Nous allons étudier dans la suite 3 exemples complets de TAD classiques : la **liste linéaire**, la pile **LIFO**1, la file **FIFO**2. Pour chacun de ces exemples, il sera fourni une spécification opérationnelle en pascal, puis plus loin en Delphi.

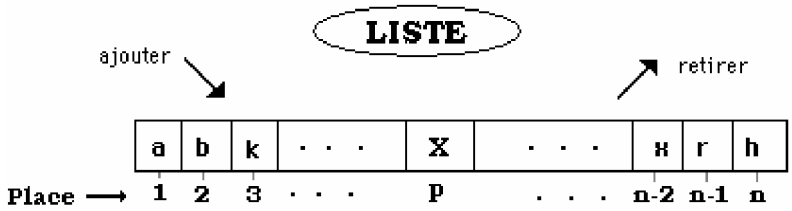
Exemples de types abstraits de données

1.5 Le TAD liste linéaire (spécifications abstraite et concrète)

Spécification abstraite

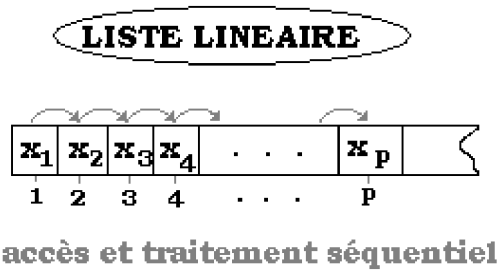
Répertorions les fonctionnalités d'une liste en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments en n'importe quel point de la liste.
- **L'ordre** des éléments est primordial. Cet ordre est construit, non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 : $(a_i)_{i \in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- Chaque place a un **contenu** de type T_0.
- Le nombre d'éléments d'une liste λ est appelé **longueur** de la liste. Si la liste est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer au minimum (non exhaustif) les actions suivantes sur les éléments d'une liste λ : **accéder** à un élément de place fixée, **supprimer** un élément de place fixée, **insérer** un nouvel élément à une place fixée, etc ....



si Place = p **alors** Contenu (Place) = X **fsi**

- C'est une structure de donnée séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres :



accès et traitement séquentiel

De cette description nous extrayons une spécification sous forme de TAD.

Ecriture syntaxique du TAD liste linéaire

TAD Liste
utilise : **N**, T_0, Place
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
liste_vide : → Liste
acces : Liste x N → Place
contenu : Place → T_0
kème : Liste x **N** → T_0
long : Liste → **N**
supprimer : Liste x **N** → Liste
inserer : Liste x **N** x → Liste
succ : Place → Place

préconditions :
acces(L,k) **def_ssi** $1 \leq k \leq long(L)$
supprimer(L,k) **def_ssi** $1 \leq k \leq long(L)$
inserer(L,k,e) **def_ssi** $1 \leq k \leq long(L) +1$
kème(L,k) **def_ssi** $1 \leq k \leq long(L)$

Fin-Liste

signification des opérations : (spécification abstraite)

acces(L,k) : opération générale d'accès à la position d'un élément de rang k de la liste L.
supprimer(L,k) : suppression de l'élément de rang k de la liste L.
inserer(L,k,e) : insérer l'élément e de T_0 , à la place de l'élément de rang k dans la liste L.
kième(L,k) : fournit l'élément de rang k de la liste.

spécification opérationnelle concrète

- La liste est représentée en mémoire par un ***tableau*** et un attribut de ***longueur***.
- Le kème élément de la liste est le kème élément du tableau.
- Le tableau est plus grand que la liste (il y a donc dans cette interprétation une contrainte sur la longueur. Notons Longmax cette valeur maximale de longueur de liste).

Il faut donc, afin de conserver la cohérence, ajouter deux préconditions au **TAD Liste** :

long(**L**) **def_ssi** long(**L**) ≤ Longmax
inserer(**L,k,e**) **def_ssi** $(1 \leq k \leq long(L) +1) \wedge long(\mathbf{L}) \leq Longmax$**)**

D'autre part la structure de tableau choisie permet un traitement itératif de l'opération kème (une autre spécification récursive de cet opérateur est possible dans une autre spécification opérationnelle de type dynamique).

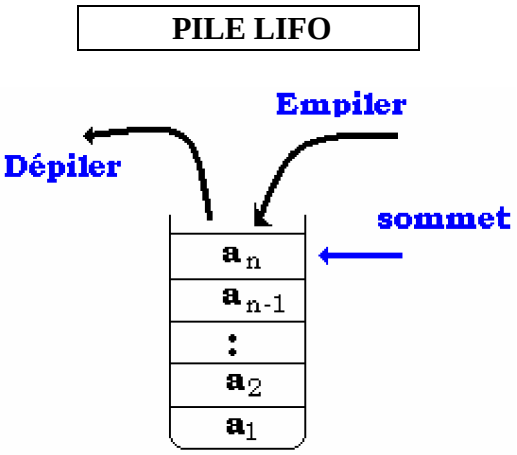
kème(L,k) = contenu(acces(L,k))

1.6 Le TAD pile LIFO (spécification abstraite et concrète)

Spécification abstraite

Répertorions les fonctionnalités d'une pile LIFO (Last In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit de n'effectuer un traitement que sur le dernier élément entré. *Exemples* : pile de dossiers sur un bureau, pile d'assiettes, etc...
- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments uniquement au début de la pile.
- **L'ordre** des éléments est imposé par la pile. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i\in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La pile possède une place spéciale dénommée **sommet** qui identifie son premier élément et contient toujours le dernier élément entré.
- Le nombre d'éléments d'une pile LIFO P est appelé **profondeur** de la pile. Si la pile est vide nous dirons que sa profondeur est nulle (profondeur = 0).
- On doit pouvoir effectuer sur une pile LIFO au minimum (non exhaustif) les actions suivantes : voir si la pile **est vide**, **dépiler** un élément, **empiler** un élément, observer le **premier** élément sans le prendre, etc...



- C'est une structure de données séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres à partir du sommet

Ecriture syntaxique du TAD Pile LIFO

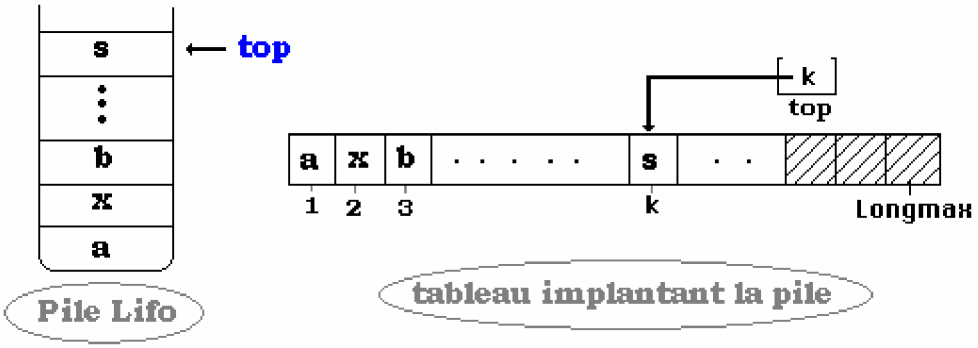
TAD PILIFO

utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
sommet : → PILIFO
Est_vide : PILIFO → Booléens
empiler : PILIFO x T_0 x sommet → PILIFO x sommet
dépiler : PILIFO x sommet → PILIFO x sommet x T_0
premier : PILIFO → T_0
préconditions :
dépiler(P) **def_ssi** est_vide(P) = **Faux**
premier(P) **def_ssi** est_vide(P) = **Faux**

FinTAD-PILIFO

spécification opérationnelle concrète

- La Pile est représentée en mémoire dans un *tableau*.
- Le *sommet* (noté **top**) de la pile est un *pointeur* sur la case du tableau contenant le début de pile. Les variations du contenu k de top se font au gré des empilements et dépilements.
- Le tableau est plus grand que la pile (il y a donc dans cette interprétation une contrainte sur la longueur, notons *Longmax* cette valeur maximale de profondeur de la pile).
- L'opérateur *empiler* : rajoute dans le tableau dans la case pointée par top un élément et top augmente d'une unité.
- L'opérateur *depiler* : renvoie l'élément pointé par **top** et diminue top d'une unité.
- L'opérateur *premier* fournit une copie du sommet pointé par top (la pile reste intacte).
- L'opérateur *Est_vide* teste si la pile est vide (vrai si elle est vide, faux sinon).

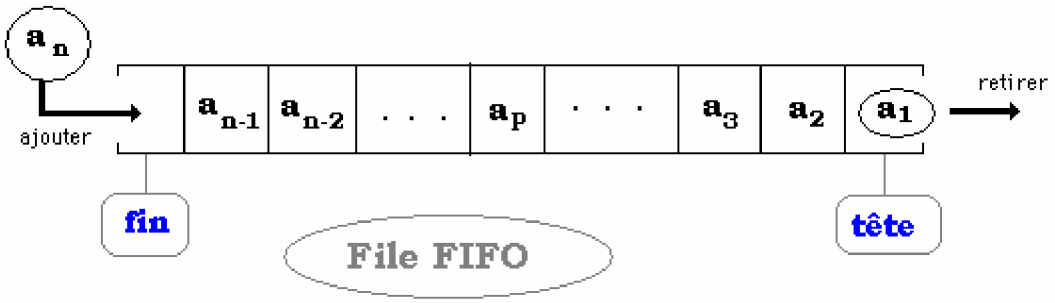


1.7 Le TAD file FIFO (spécification abstraite seule)

Spécification abstraite

Répertorions les fonctionnalités d'une file FIFO (First In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit d'effectuer un traitement selon l'ordre d'arrivée des éléments, comme dans une file d'attente.
- *Exemples* :toutes les files d'attente, supermarchés, cantines , distributeurs de pièces, etc...
- Il est possible dans une telle structure **d'ajouter** des éléments à la fin de la file, ou de **retirer** des éléments uniquement au début de la file.
- **L'ordre** des éléments est imposé par la file. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i\in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La file possède deux places spéciales dénommées **tête** et **fin** qui identifient l'une son premier élément, l'autre le dernier élément entré.
- Le nombre d'éléments d'une file FIFO " F " est appelé **longueur** de la file ; si la file est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer sur une file FIFO au minimum (non exhaustif) les actions suivantes : voir si la file **est vide**, **ajouter** un élément, **retirer** un élément, observer le premier élément sans le prendre, etc...



Ecriture syntaxique du TAD file FIFO

TAD FIFO
utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
tête : → FIFO
fin : → FIFO
Est_vide : FIFO → Booléens
ajouter : FIFO x T_0 x fin → PILIFO x fin
retirer : FIFO x tête → FIFO x tête x T_0
premier : FIFO → T_0

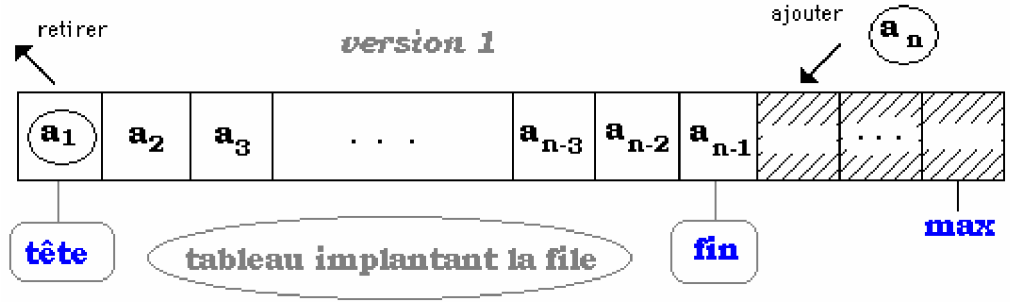
préconditions :
retirer(F) **def_ssi** est_vide(F) = **Faux**
premier(F) **def_ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Spécification opérationnelle concrète

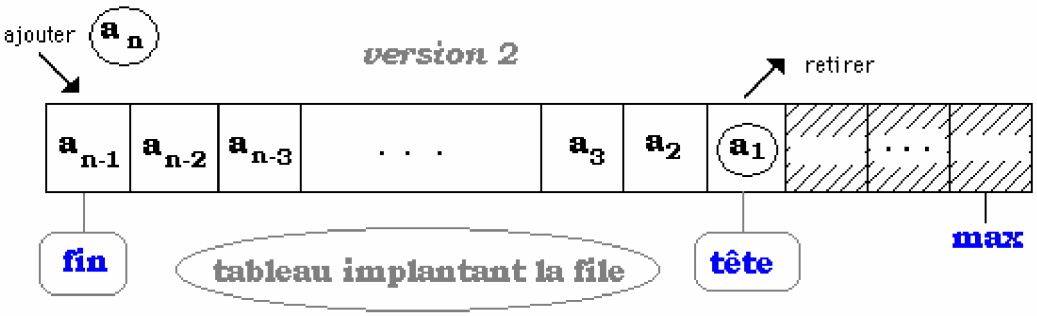
- La file est représentée en mémoire dans un *tableau*.
- La *tête* de la file est un pointeur sur la case du tableau contenant le début de la file. Les variations de la valeur de la tête se font au gré des ajouts et des retraits.
- La *fin* ne bouge pas , c'est le point d'entrée de la file.
- Le tableau est plus grand que la file (il y a donc dans cette interprétation une contrainte sur la longueur ; notons *max* cette valeur maximale de longueur de la file).
- L'opérateur *ajouter* : ajoute dans le tableau dans la case pointée par fin un élément et tête augmente d'une unité.
- L'opérateur *retirer* : renvoie l'élément pointé par tête et diminue tête d'une unité.
- L'opérateur *premier* fournit une copie de l'élément pointé par tête (la file reste intacte).
- L'opérateur *Est_vide* teste si la file est vide (vrai si elle est vide, faux sinon).

On peut ajouter après la dernière cellule pointée par l'élément fin comme le montre la figure ci-dessous :



dans ce cas retirer un élément en tête impliquera un décalage des données vers la gauche.

On peut aussi choisir d'ajouter à partir de la première cellule comme le montre la figure ci-dessous :



dans ce cas ajouter un élément en fin impliquera un décalage des données vers la droite.