Introduction aux Types Abstraits de Données: Le Type Liste

Cours Premier Cycle (IBIS), INSA de Rouen
Habib Abdulrab

Plan

- (I) Types Abstraits de Données (TAD): pourquoi ?
- (II) Le TAD: liste.
- (III) Représentations des listes dans un pseudo langage algorithmique.
- (IV) Implémentation en C.
- (V) Travaux Dirigés (TDs)

(I) Type Abstrait de Données: pourquoi (1)

- Un Type abstrait de données (TAD) est un ensemble de données organisé, et d'opérations sur ces données. Il est défini d'une manière indépendante de la représentation des données en mémoire.
- Exemple 1): *Les nombres réels*, munis des opérations: +, *, /, …
- Exemple 2): *Les chaînes de caractères*, munies des opérations: insertion, concaténation, …

(I) Type Abstrait de Données: pourquoi (2)

- Cette notion est centrale en Informatique. Elle met au même niveau les *données* et les *opérations*.
- Elle permet de *modéliser* les applications informatiques autour des collections imbriquées de TADs.
- Elle permet l'élaboration des algorithmes d'une manière *abstraite*:

  1) en faisant appel aux données et aux opérations abstraites du TAD (couche supérieure),

  2) suivi d'un choix de représentation du TAD en mémoire (couche inférieure).

Types Abstraits de Données: pourquoi (3)

- Le codage de la couche supérieure donne des programmes abstraits, compréhensibles, et réutilisables.
- Le codage de la couche inférieure implante un choix de la représentation du TAD en mémoire.
- La couche supérieure reste inchangée si on change cette couche inférieure.

(II) Le TAD: Liste (1)

- Une **liste** S est une suite finie d'éléments s_1, s_2,…, s_n; $n \geq 0$.
- On appelle:
  - **tête** de S: premier élément s_1 de S,
  - **reste** de S: la *liste*: s_2,...,s_n,
  - **longueur** de S : nombre d'éléments de S,
  - **liste vide**: liste sans élément (de longueur 0), notée NIL, ou ().
  - **successeur** de l'élément s_i: l'élément s_{i+1} (i<n),

Le TAD: Liste (2)

- **Quelques Opérations de base sur les liste:**
  - accès au premier élément de S,
  - accès au reste de la liste,
  - vérification si une liste est vide ou non,
  - accès au $i^{ème}$ élément de S,
  - insertion d'un élément en tête d'une liste,
  - insertion d'un élément après l'élément courant,
  - suppression de l'élément courant,
  - ……

Le TAD: Liste (3)

- **Quelques fonctions liées à ces opérations de base**:
  - **premier(S):** retourne le premier élément de S.
  - **reste(S):** retourne la liste $s_2,\ldots, s_n$.
  - **vide?(S):** retourne vrai si S = (), faux sinon.
  - **notVide?(S):** retourne faux si S = (), vrai sinon.
  - **i^{ème}(S, k):** retourne l'élément s_k.
  - **ajouterDebut(x, S):** retourne la liste $x, s_1, s_2,\ldots, s_n$.
  - ……

Le TAD: Liste (4)

- **D'autres Opérations utiles sur les listes :**
  - Détermination de la longueur de la liste: la liste est entièrement parcourue en incrémentant un compteur à chaque déplacement.
  - Parcours de la liste à la recherche d'une donnée x: on se déplacée du début à la fin de la liste, afin de chercher si un de ses éléments est égal à x.
  - Concaténation de deux listes: on obtient une liste contenant les éléments de L_1 suivis par les éléments de L_2.
  - Tri: les éléments sont réordonnés en fonction de leur relation d'ordre.
  - ……

Exemple d'un algorithme sur le TAD liste (1)

/* Fonction qui retourne l'inverse d'une liste */

Entrée: une liste l = $l_1, l_2, \ldots, l_n$; n ≥ 0.

Sortie: la liste $l_n, l_{n-1}, \ldots, l_1$; n ≥ 0.

inverser(l)

1. Resultat ← ().
2. Tant que notVide?(l) faire:
   - 2.1 Resultat ← ajouterDebut(premier(l), Resultat);
   - 2.2 l ← reste(l);
3. Retourner Resultat;

Exemple d'un algorithme sur le TAD liste (2)

- Le codage de cet algorithme permet d'obtenir une couche abstraite, indépendante des choix de l'implantation des listes en mémoire.
- Le codage séparé des opérations primitives: notVide?(liste), ajouterDebut(donnée, liste), reste(liste) fixe un certain choix d'implantation.
- La couche abstraite reste intacte si on modifie ce choix d'implantaion.

(III) Représentation des listes en mémoire

- On cherche à représenter les listes en mémoire dans un pseudo langage algorithmique indépendant d'un langage informatique.
- Rappelons auparavant les concepts de: *mémoire, variable, type, tableau, structure, pointeur*; et les notations que nous utilisons en pseudo langage algorithmique.

1) La mémoire

- La mémoire peut être vue comme une grille uni-dimensionnelle constituée de cases.
- Chaque case de la grille a:
  - un contenu: sa *valeur*;
  - un numéro: son *adresse* (comprise entre 0 et le nombre de cases disponibles).

2) Les variables, les Types

- Une variable à un nom.
- A chaque variable est associée une zone de l'espace mémoire: un ensemble de cases contiguës (1 ou plusieurs) contenant la *valeur* de la variable.
- Une variable a un *type* qui permet d'indiquer la manière d'interpréter l'ensemble de cases associées à cette variable.
- Une variable a une *adresse*.

3) Enregistrement (1)

- En plus des **types élémentaires** : Entiers, Réels, Caractères…, on peut définir des nouveaux types (des types **composites**) grâce à la notion d'**enregistrement**.
- Remarque: La notion de **Classe** (beaucoup plus riche) dans les langages à Objets remplace avantageusement la notion d'enregistrement.

3) Enregistrement (2)

- Un **enregistrement** est défini par n *attributs* munis de leurs **types, n ≥ 0.**
- Un attribut peut être de type élémentaire ou de type enregistrement.
- **Syntaxe (en pseudo langage algorithmique) :**

Type

<NomEnregistrement> **=** Enregistrement

<$Attribut_1$> :<type de $Attribut_1$>;

.....

<$Attribut_n$> :<type de $Attribut_n$>;

end;

3) Enregistrement (3)

Exemple:

On peut créer le type Personne, défini par un nom, un prénom, et un age.

```
Type Personne = Enregistrement
        nom : ChaîneDeCharactères;
        prénom : ChaîneDeCharactères;
        age : Entier;
end;
```

3) Enregistrements imbriqués (1)

- Il est possible d'imbriquer sans limitation des enregistrements les uns dans les autres.
- Exemple:

  On peut définir l'adresse (par un numéro, une rue, une ville et un code postal) comme un nouvel enregistrement, et l'ajouter comme un attribut supplémentaire à l'enregistrement Personne.

3) Enregistrements imbriqués (2)

```
Type Adresse = Enregistrement
    numero: Entier;
    codePostal: Entier;
    rue,ville: ChaîneDeCharactères;
end;
Type Personne = Enregistrement
    nom,prenom: ChaîneDeCharactères;
    age : Entier ;
    adresse : Adresse;
end;
```

3) Accès aux attributs d'un enregistrement

- L'accès aux attributs d'un enregistrement se fait, attribut par attribut, à l'aide la notation "**.**"

  Exemple:

  Var p , p1: personne;

  p**.**nom ← 'Dupont';

  p**.**prenom ← 'Jean';

  p**.**adresse**.**rue ← 'Place Colbert';

4) Pointeurs

- Une variable de type pointeur est une variable qui contient une *adresse*.
- **Syntaxe (en pseudo langage algorithmique):**
  - **Var nom : ^<type>;**
    - Exemple :

    Var x : ^Entier;
  - **nom^** désigne la valeur (de type <type>) sur laquelle pointe la variable nom.
  - **NIL** est le pointeur vide: une variable x de type pointeur initialisée à **NIL** signifie que x ne pointe sur rien.
  - **allouer(nom, <type>)** permet d'allouer dynamiquement de l'espace mémoire (de type <type>) et fait pointer *nom* sur elle.
  - **liberer(nom)** permet de libérer l'espace mémoire alloué ci-dessus.

5) Structures et pointeurs

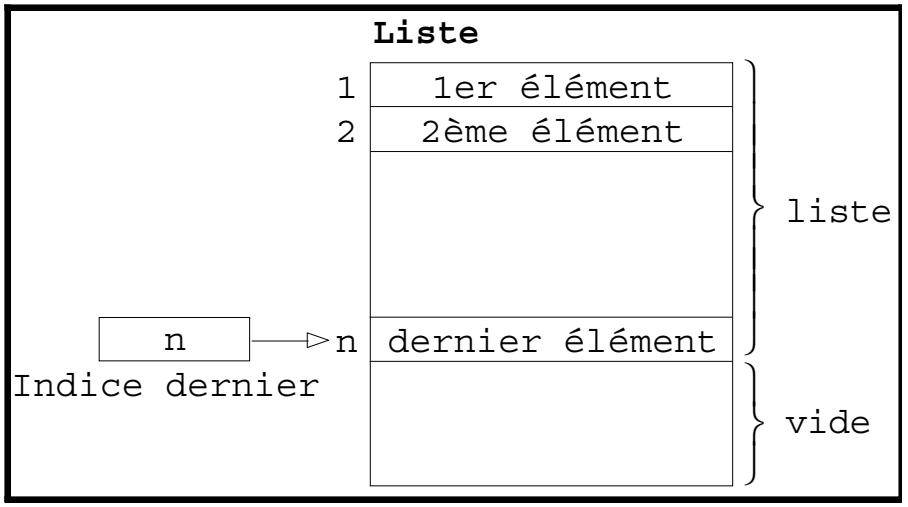
- On peut également définir le type: Pointeur sur des Structures :
  - Exemple :

    Var x : ^Personne;
  - L'accès à l'attribut *age* de la personne X se fait comme suit:

    x^.age;

A) Représentation des listes par des tableaux (1)

- Chaque élément s_i de la liste est placé dans la cellule i du tableau.
- L'indice du dernier élément est nécessaire pour délimiter la zone occupée par la liste de la zone encore inoccupée.



A) Représentation des listes par des tableaux (2)

- Avantages:
  - Parcours et accès faciles au $i^{ème}$ élément (accès direct).
  - Possibilité de recherche efficace si la liste est triée (par exemple, recherche dichotomique).
- Inconvénients:
  - Réservation, lors de la compilation, de la Taille maximale imposée (limitée), ce qui manque de souplesse et d'économie.
  - Inefficacité de la suppression et de l'insertion à l'intérieur de la liste: obligation de décaler tous les éléments entre l'élément inséré ou supprimé et le dernier élément.

B) Représentation des listes par des pointeurs (1)

Rappel :

- On peut redéfinir une *liste d'éléments L de type T* comme suit :

  L est NIL (i.e. La liste vide), ou

  elle est un élément du type ListeNonVide, avec :

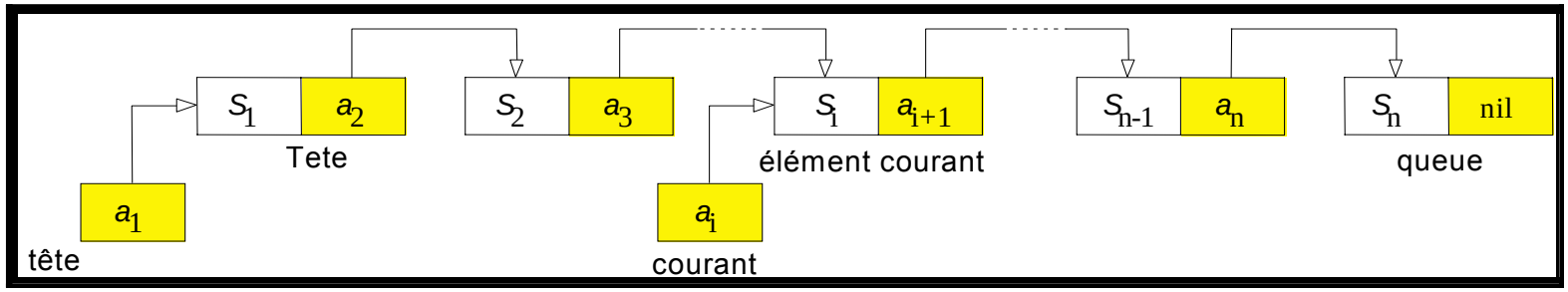
  Type ListeNonVide = **ENREGISTREMENT**

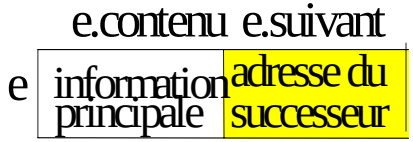
  debut : T ;

  suivant : Liste ;

  Fin

B) Représentation des listes par des pointeurs (1)





Déclaration de la liste non vide

```
TYPE

 cellule = Enregistrement
    contenu : typeElement;
    suivant : pointeurCellule

 pointeurCellule = ^cellule;

END ;
VAR  tete, courant : pointeurCellule
```

B) Représentation des listes par des pointeurs (2)

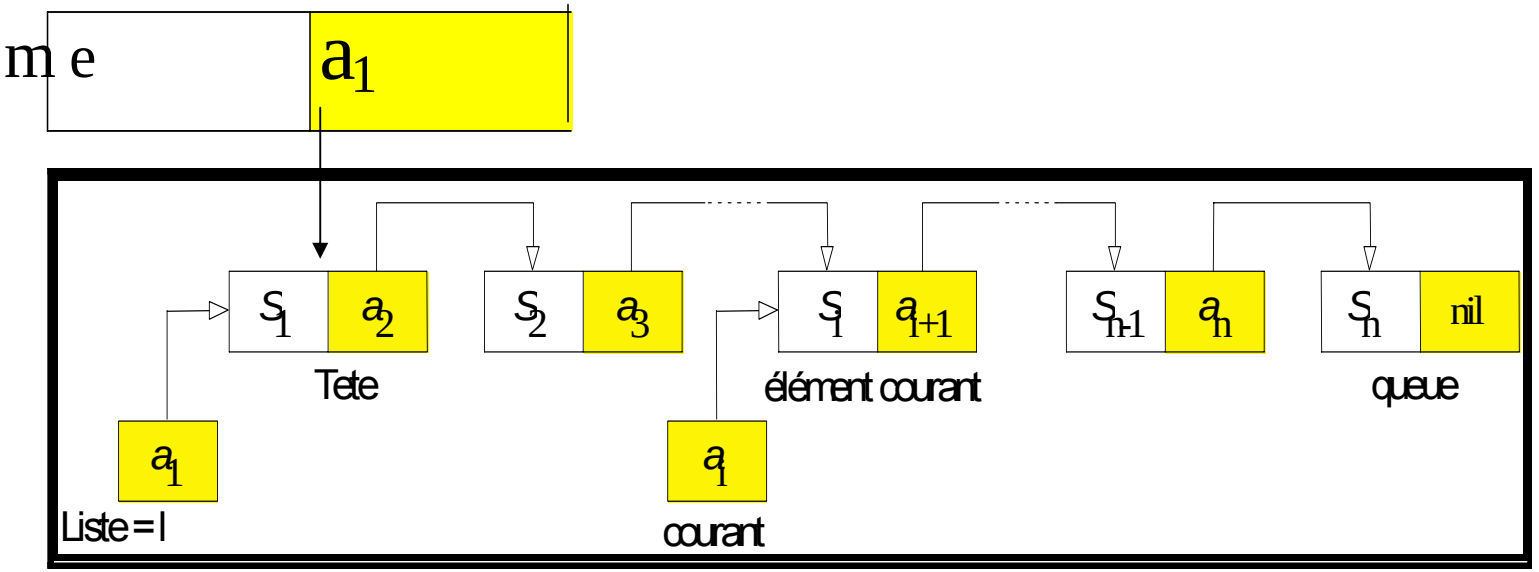
- Avantages:
  - La quantité de mémoire utilisée est exactement ajustée aux besoins.
  - Insertion et suppression plus aisées et efficaces que dans le cas de la représentation par des tableaux.
- Inconvénients:
  - Accès séquentiel uniquement.

Opérations sur les listes (Représentées par des pointeurs)

- Dans les listes chaînées simples, seul le premier élément est directement accessible.
- L'accès aux autres éléments est réalisé en parcourant la liste par déplacements successifs d'un élément à son successeur. Cela revient à déplacer un pointeur qui parcourt la liste.
- L'élément sur lequel est positionné ce pointeur à un instant donné est l'élément **courant**.
- A l’initialisation l'élément courant = tete.

1) Ajout d'un élément au début d'une liste

- 1- Allocation d'un nouvel emplacement mémoire *m* de type cellule.
- 2- Remplissage de *m* par l'élément *e*, et le pointeur vers l'adresse de la première cellule de la liste.



Algorithme: ajouterDebut

/* Algorithme qui retourne une nouvelle liste (sans effet de bord) construite en ajoutant une cellule de contenu = e au début d'une liste l */

Entrée: une élément e, une liste $l = l_1, l_2, \ldots, l_n$; $n \geq 0$.

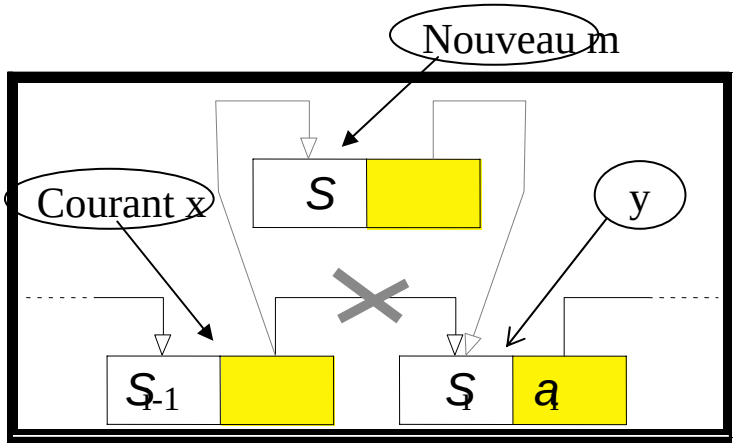
Sortie: une nouvelle liste $n = e, l_1, l_2, \ldots, l_n$.

ajouterDebut (e, l)

1. allouer(m, cellule) ;
2. m**^**.contenu ← e ;
3. m**^**.suivant ← l ;
4. Retourner m ;

2) Insertion d'un élément

- On cherche à insérer un élément S à la place i (i = 1, … , n+1) de la liste L (de longueur n).
- Remarque 1) Si i = 1, on utilise l'opération précédente: ajouterAuDébut(S , L)
- Remarque 2) Si i = n+1, y = NIL.

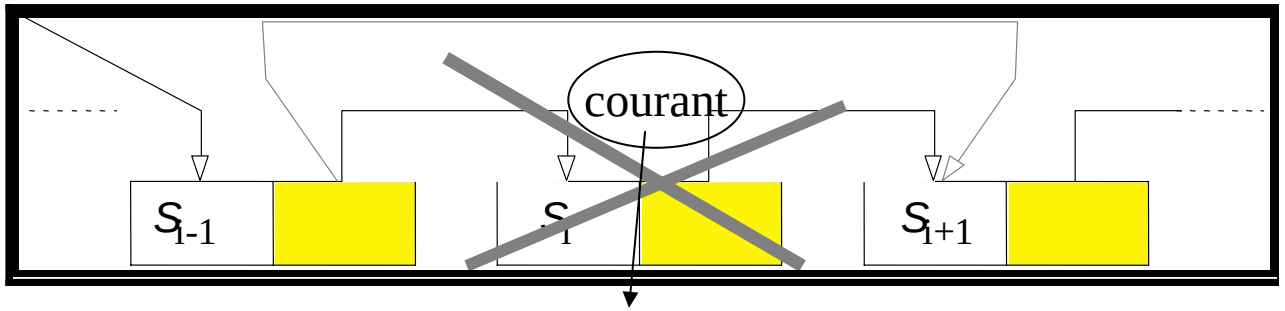


2) Insertion d’un élément (suite)

- 1- Positionnement du pointeur de parcours sur l’adresse de la cellule x avant lequel doit s'effectuer l'insertion.
- 2- Allocation d'un nouvel emplacement mémoire m de type cellule;
- 3- Remplissage de m par l’élément inséré S et l’adresse de y.
- 4- Modification de la cellule x: elle sera suivie par m.

3) Suppression d'un élément

- On cherche à supprimer l'élément S de la liste L (de longueur n) qui se trouve à la place à la place i (i = 1, … , n)
- Si i = 1, on retourne reste(L).



3) Suppression d'un élément (suite)

- 1- Positionnement du pointeur sur l'adresse de la cellule avant lequel doit s'effectuer l'insertion.

- 2- Modification des valeurs des pointeurs, (la cellule de s_{i-1} sera suivie par la cellule de s_{i+1}).

- 3- Libération de l'emplacement mémoire réservé auparavant à l'élément supprimé.

4) Exemples d'algorithmes sur la représentation par pointeurs: concatener

/* Algorithme qui modifie (avec effet de bord) une liste *tete1* en insérant physiquement une deuxième liste *tete2* à sa fin*/

Entrée: 2 listes: liste1 = $u_1, u_2,\ldots, u_n$; $n \geq 0$, liste2 = $v_1, v_2,\ldots, v_m$; $m \geq 0$.

Sortie: liste1 = $u_1, u_2,\ldots, u_n, v_1, v_2,\ldots, v_m$

concatener(liste1, liste2)

/* si tete = () on retourne la deuxième liste */

1. Si vide?(liste1) alors Retourner liste2;
2. sinon

4) Exemples d'algorithmes sur la représentation par pointeurs: concatener (suite)

2. sinon

/* Sinon on parcourt la première liste */

2.1 courant ← liste1;

2.2 Tant que notVide?(courant ^.suivant)

courant ← courant^.suivant;

/* insertion de liste2 à la fin de liste1 */

2.3 courant ^.suivant ← liste2;

3. Retourner liste1;

(IV) Implantation en C
Rappel: 1) *Struct*

- Un *enregistrement* est implanté en C par l'opérateur *struct*, comme suit:

```
struct NombreComplexe
    {
      double reel;
      double imaginaire;
}
```

- Le mot NombreComplexe n'est pas un type mais un nom de structure. Il doit toujours être accompagné du mot-clé struct.

2) *typedef*

- *typedef* permet de créer des types synonymes aux types existants :
- Exemple:

  typedef <TypeExistant> <TypeSynonyme>;

  /* Declaration du type 'montant' synonyme de 'double‘ */

  typedef double montant;

typedef (2)

- L'exemple suivant définit un type TypeNombreComplexe synonyme du type struct NombreComplexe, ainsi qu'une variable c de ce nouveau type

```
typedef struct NombreComplexe
{
        double reel;
        double imaginaire;
} TypeNombreComplexe;
TypeNombreComplexe c;
```

typedef (3)

- Il est utile pour créer des nom de TAD, afin d'écrire des programmes abstraits et portables: si on change de représentation du TAD, il suffit d'affecter au typedef du TDA un nouveau type.
- Exemple :

  Ancien type:

  typedef ListeParTableau Liste

  Nouveau type:

  typedef ListeParPointeur Liste

Pointeurs (1)

- Syntaxe:

  <type> *nom;

  Définit un pointeur de type *pointeur sur type*.
- **Caractéristiques des pointeurs**
  - la valeur du pointeur est l'*adresse* (ou la *référence*) de l'objet pointé ;
  - l'opérateur `&', appliqué à un objet, retourne l'adresse de l'objet ;
  - dans un bloc de déclarations l'opérateur ‘*' sert à définir/déclarer un pointeur. Dans un bloc d'instructions il sert à l'*accès indirect* à l'objet pointé ;

Pointeurs (2)

- Exemples :

  int x = 1, y = 2 ;

  int *p ; /* Déclare p comme un pointeur sur un entier */

  p = &x /* p contient l'adresse de x */

  y = *p /* y vaut 1 */

  p = 3 / x vaut maintenant 3 */
- Remarque : *&x = x ; &*p = p
- Le pointeur NULL est le pointeur vide. Il ne pointe sur rien. On l'utilise pour initialiser les variables de type pointeur.

Structures et Pointeurs (1)

- On peut également définir des types synonymes de pointeurs sur des structures :

  typedef struct NombreComplexe * PointeurNombreComplexe;

- Pour définir une variable *p* dont le type est pointeur sur une structure NombreComplexe:

  PointeurNombreComplexe p;

Structures et pointeurs (2)

- L'opérateur **->** permet l'accès aux attributs d'une structure. Son opérande gauche doit être un pointeur p sur une structure. Son opérande droit doit être un attribut *a* de cette structure. On a alors l'équivalence suivante :

  p->a <=> (*p).a

- Avec p de type PointeurNombreComplexe on peut donc faire:

```
p->reel = 0;
p->imaginaire = 1;
printf("c = (%f, %f)\n", p->reel, p->imaginaire);
```

Exemple: le Type Liste

```
struct cellule
{
        int contenu;
        struct cellule * suivant;
};
typedef struct cellule * Liste;
```

Exemples: 1) Affichage des éléments d'une liste

```
/* procédure qui affiche une liste */
void afficherListe(Liste tete)
{
 Liste courant = tete;
 while (courant != NULL)
        {
        printf("%d ", courant->contenu);
        courant=courant->suivant;
        }
}
```

Allocation de mémoire (1)

- il est possible d'attribuer à un pointeur de la mémoire allouée dynamiquement.
- L'allocation dynamique de la mémoire est nécessaire car on ne connaît pas toujours à l'avance la quantité de mémoire nécessaire.
- La fonction *malloc()* permet d'allouer un bloc de mémoire (un ensemble de cases contiguës). Elle prend en paramètre le nombre de cases à allouer, et elle retourne un pointeur vers la première case de la zone allouée.

Allocation de mémoire (2)

- Quand on veut allouer un bloc pour un ensemble d'éléments, le nombre de cases à allouer correspond au nombre d'éléments multiplié par la taille du type des éléments.

- L'opérateur *sizeof* prend un type en paramètre et retourne sa taille en nombre de cases (i.e. en nombre d'octets).

Allocation de mémoire (3)

- Exemple: on veut allouer 10 éléments de type int :

```
int *p = NULL;
p = malloc(10 * sizeof(int));
```

- La fonction *free* complète *malloc*. Elle permet de libérer un bloc de mémoire. Elle prend en paramètre l'adresse de la zone de mémoire à libérer. Cette adresse doit avoir été fournie par *malloc* et elle ne doit pas avoir déjà été libérée.

2) Ajout au début d'une liste

```
/* fonction qui retourne une nouvelle liste (sans effet
de bord) construite en ajoutant une cellule de contenu
= donnee au début d'une liste tete */

Liste ajouterDebut(int donnee, Liste tete)
{
 Liste nouveau;
  nouveau=(Liste)malloc(sizeof(struct cellule));
  nouveau->contenu=donnee;
  nouveau->suivant=tete;
  return nouveau;
}
```

3) Concaténation de deux listes (avec effets de bords)

```
/* Fonction qui retourne une liste tete1 modifiée (avec effet de bord) en
    insérant physiquement une deuxième liste tete2 à sa fin*/
Liste concatener(Liste tete1, Liste tete2)
    {
     Liste courant; /* pointeur de parcourt de tete1 */
     /* si tete = () on retourne la deuxième liste */
    if (tete == NULL) {return tete2}
    else
     /* Sinon on parcourt la première liste jusqu'à la dernière cellule, puis on
    lui insère la nouvelle liste comme suivante */
     {courant=tete1;
     while (courant->suivant != NULL)
            courant=courant->suivant;
     courant->suivant=tete2; /* insertion de tete2 à la fin de tete1 */
    /* Et on retourne la première liste modifiée */
    return tete1; }}
```

4) Insertion à la fin d'une liste

```
/* Fonction qui modifie (avec effet de bord) une liste tete
   en insérant physiquement une cellule de contenu =
   donnee à sa fin*/
Liste insererFin(int donnee, Liste tete)
   {
    Liste nouveau, courant;
    /* Allocation de la nouvelle Cellule */
    nouveau=(Liste)malloc(sizeof(struct cellule));
    nouveau->contenu=donnee;
    nouveau->suivant=NULL;
```

4) Insertion à la fin d'une liste (suite)

```
/* si tete = () on retourne la nouvelle cellule */
if (tete == NULL) {return nouveau}
else
 /* Sinon on parcourt la liste jusqu'à la dernière cellule,
puis on lui insère la nouvelle cellule comme suivante */
 {courant=tete;
 while (courant->suivant != NULL)
       courant=courant->suivant;
 courant->suivant=nouveau;
/* Et on retourne la nouvelle liste modifiée */
return tete; }}
```

5) Insertion

/* fonction qui retourne une liste *tete* modifiée (avec *effet de bord*) après l' insertion d'une cellule de *contenu* = donnee avant y */

```
Liste inserer(int donnee, Liste tete, Liste y)
{
 Liste nouveau, courant;
 courant=tete;
 while (courant->suivant != NULL) && (courant != y)
                courant=courant->suivant;
```

5) Insertion

```
if (courant->suivant != NULL)
      {/* Allocation de la nouvelle Cellule */
        nouveau=(Liste)malloc(sizeof(struct cellule));
        nouveau->contenu=donnee;
        nouveau->suivant=y;
        courant->suivant=nouveau; /* Insertion */

      }
return tete;
}
```

Références

1) Structures de Données et Algorithmes, *A. Aho, J. Hopcroft, J. Ullman*. InterEditions.

2) Types de Données et Algorithmes, *C. Froidevaux, M-C. Gaudel, M. Soria*. Editions McGRAW-Hill.

3) Quelques cours de DEUG:
   - a) ENS de Lyon, O. Aumage
   - b) Depart. Info de l'Université de bourgogne.

TD1

- **On considère des polynômes de degré maximal *n* de la forme: P(x) = an xn + an-1 xn-1 + ... a1x1 + a0**
- **1. Spécifier un type abstrait Polynôme avec les opérations suivantes :**
  - **construire le polynôme constant P(x) = a0 ;**
  - **construire le polynôme P(x) = Q(x).x + a, Q étant un polynôme et a un réel ;**
  - **retourner le i-ème coefficient d'un polynôme (ai) ;**
  - **calculer la valeur d'un polynôme P(x) en un x donné.**
  - **calculer le degré du polynôme (d tel que ai = 0 pour tout i > d).**
- **2. Montrer comment construire le polynôme 8x5 + 9x4 + 5x2 + 3x + 4 avec ce type abstrait.**
- **3. Donner des algorithmes pour les 5 opérations de base données ci-dessus.**

TD1 (suite)

- **4. Donner des algorithmes pour les fonctions suivantes en utilisant seulement les opérations du type abstrait :**
  - **ajouter deux polynômes ;**
  - **calculer la dérivée d'un polynôme (qui est un polynôme) ;**
  - **calculer la valeur maximale d'un polynôme sur les valeurs entières d'un intervalle [a, b] donné, a et b étant entiers.**
  - **Spécifier et réaliser la fonction Puissance (x: real, n: integer) qui retourne x à la puissance n.**
- **5. En choisissant d'implanter les listes des coefficients par des *tableaux*, compléter la *couche supérieure* abstraite du type abstrait Polynôme par une *couche inférieure* qui définit les opérations de base sur ces listes.**

TD2

Rappel :

On définit une *liste linéaire d'éléments L de type T* comme suit :

L est nil (i.e. La Liste Linéaire VIDE)

OU L est un pointeur du type ListeLinéaireNonVide, avec :

Type ListeLinéaireNonVide = **ENREGISTREMENT**

contenu : T ;

suivant : ListeLinéaire ;

Fin

Exercices 1) :

Soient : l : ListeLinéaire de type T de *longueur* n ;

x un élément de type T ;

et *i:naturel > 0.*

Ecrire un algorithme qui :

TD2 (Suite)

1. Retourne le premier élément d'une liste: ***premier (l : ListeLinéaire) : T***
2. Retourne le reste d'une liste: ***reste (l : ListeLinéaire) : ListeLinéaire***
3. Teste si une liste est vide: ***estVide? (l : ListeLinéaire) : booléen***
   // renvoie vrai si la liste est vide, faux sinon.
4. Ajoute un élément au début de la liste: **insererAuDebut *(x : T; l : ListeLinéaire)***
5. Ajoute un élément à la position i: ***inserer (x : T; i : naturel; l : ListeLinéaire)*** *RQ : l peut être modifiée physiquement*
6. Supprime un élément à la position i: ***supprimer (i:naturel; l : ListeLinéaire)*** *RQ : l peut être modifiée physiquement*

TD2 (suite)

7. Cherche un élément dans une liste: ***chercher (x : T; l : ListeLinéaire) : ListeLinéaire //*** *Recherche dans une liste l'élément de valeur x. Le résultat de la recherche est la liste qui commence par x, nil sinon.*

8. Imprime une liste: ***Imprimer (l : ListeLinéaire)***

9. Calcule la longueur d'une liste: ***longueur (l : ListeLinéaire): Naturel***

TD2 (suite)

10. Concatène deux listes *liste1* et *liste2* en une liste *liste (sans effet de bord)*: ***concaténer(liste1, liste2 : ListeLinéaire) : ListeLinéaire***

11. Fusionne deux listes *liste1* et *liste2* en une liste *liste*: ***fusionner(liste1, liste2 : ListeLinéaire) : ListeLinéaire***

Exercices 2) : Traduire cette bibliothèque en C.

TD2 (suite)

1.

premier (l:ListeLinéaire):T ou nil
debut
    si estVide(l) **alors retourner** nil
                sinon retourner l^.contenu
fin

2.

reste(l:ListeLinéaire):ListeLinéaire
debut
    si estVide(l) **alors retourner** nil
                sinon retourner l^.suivant

TD2 (suite)

3.

estVide?(l:ListeLinéaire):booléen
debut
    si l=nil **alors retourner** vrai
        sinon retourner faux
fin

4.

insererAuDebut(x:T, l:ListeLinéaire)
declaration l1:ListeLinéaire // RQ : allocution d'une nouvelle cellule l1 (i.e. allouer(l1, cellule).
debut
    l1^.contenu ← x
    l1^.suivant ← l
    retourner l1
fin

TD2 (suite)

5.

```
inserer(x:T, i:naturel, l:ListeLinéaire) // i <= n+1
declaration          nouveau, courant : ListeLinéaire
                     k : naturel
debut
Si i = 1 retourner insererAuDebut(x, l), Sinon faire
courant ← l
    pour k allant de 1 à i-2 faire
            courant ← courant^.suivant
    finpour // ici courant pointe sur la la i-1 ieme cellule.
    nouveau^.contenu ← x
    nouveau^.suivant ← courant^.suivant
    courant^.suivant ← nouveau
    retourner l ;
FinSi
fin
```

TD2 (suite)

RQ :

Les étapes :

nouveau^.contenu ← x

nouveau^.suivant ← courant^.suivant

courant^.suivant ← nouveau

peuvent être replacées par :

courant^.suivant ← insérerAuDebut(x, courant) ;

TD2 (suite)

6.

supprimer(i:naturel, l:ListeLinéaire) // i <= n
declaration k : naturel
courant : ListeLinéaire
debut
Si i = 1 **retourner** reste(l), Sinon **faire**
courant ← l
pour k allant de 1 à i-2 **faire**
courant ← courant^.suivant
finpour// ici *courant* pointe sur la la i-1 ieme cellule.
courant^.suivant ←(courant^.suivant) ^.suivant
retourner l ;
FinSi
fin

TD2 (suite)

7.

```
chercher(x:T, l:ListeLinéaire):ListeLinéaire
debut
   tantque notVide?(l) faire
        si l^.contenu=x alors retourner l
        l ← l^.suivant
   fintantque
   retourner nil
fin
```

TD2 (suite)

8.

```
imprimer(l:ListeLinéaire)
debut
   tantque notVide?(l) faire
        ecrire(l^.contenu)
l ← l^.suivant
   fintantque
fin
```

TD2 (suite)

9.

longueur(l:ListeLinéaire) : Naturel
declaration n : Naturel
Debut
n ← 0;
    tantque notVide?(l) faire
        n ← n+1;
        l ← l^.suivant
    fintantque
Retourner n;
fin

TD2 (suite)

10.

concatener(liste1, liste2:ListeLinéaire):ListeLinéaire

/* *courant1* parcourt la première liste, *courant* parcourt sa copie, liste3 est le résultat final */

declaration courant1, courant, liste3 : ListeLinéaire

Debut

Si estVide?(liste1) **alors Retourner** liste2;

sinon

allouer(courant , cellule); liste3 ← courant;

/* On parcourt la première liste en la copiant */

courant1 ← liste1;

TD2 (suite)

```
tantque notVide?(courant1^.suivant) faire
        courant^.contenu ← courant1^.contenu;
        allouer(x , cellule);
        courant^.suivant ← x;
        courant1 ← courant1^.suivant;
        courant ← courant^.suivant;
fintanque
/* insertion de liste2 à la fin de la copie de liste1 */
courant ^.suivant ← liste2;
Retourner liste3;
```

TD2 (suite)

11.

fusionner(liste1, liste2:ListeLinéaire):ListeLinéaire
declaration courant1, courant2, courant, liste3 : ListeLinéaire
Debut
courant1 ← liste1; courant2 ← liste2;
allouer(courant , cellule); liste3 ← courant;
/* Sinon on parcourt liste1 et liste2, en copiant la cellule courante de liste1 suivie de la cellule courante liste2 dans la nouvelle liste: courant */
pour k allant de 1 à min(longueur(liste1) , longueur(liste2)) **faire**
courant^.contenu ← courant1^.contenu;
allouer(z , cellule);
courant^.suivant ← z;
courant ← courant^.suivant;
courant1 ← courant1^.suivant;

TD2 (suite)

```
        courant^.contenu ← courant2^.contenu;
        allouer(z , cellule);
        courant^.suivant ← z;
        courant ← courant^.suivant;
        courant2 ← courant2^.suivant;
    finpour
si longueur(liste1) < longueur(liste2)
        alors courant ← liste2;
        sinon courant ← liste1;
Retourner liste3;
```

TD3

L'objectif du TD est de modéliser par TADs une petite application de gestion d'une bibliothèque, et de réaliser cette application en C.

1) Donner un modèle algorithmique complet décrivant les types de base choisis:

- Livre (défini par les attributs: titre, auteur, année…; et par les opérations: Décrire,…)

- ListeDeLivres (défini, entre autres, par une liste de Livres, et dont les opérations sont: chercherLivre, ajouterLivre,…)

2) Implémenter le modèle en C.