جامعة ابن طفيل
UNIVERSITÉ IBN TOFAÏL

Faculté des Sciences
Département d'Informatique

Structures de Données

SMI/SMA (S4)

Pr. Khalid HOUSNI

https://khalidhousni.shost.ca/

Objectifs de cours

Ce cours a pour objectifs :

- Introduire la problématique de la **structuration des données** et les types abstraits .
- Étudier les structures les plus classiques rencontrées en informatique pour organiser des données (files, piles, arbres, …). permettant la mise au point de programmes nécessitant la représentation de données complexes.

Le cours est structuré en 4 parties

- Structures de données et types abstraits.
- Structures linéaires: listes, files et piles.
- Structures arborescentes: arbres binaires, arbres binaire de recherche, tas, hachage, arbre équilibré.
- Graphes: terminologie, représentation, algorithmes de parcours.

Introduction : algorithme

Qu'est-ce qu'un algorithme ?

- Enchaînement d'opérations destiné à résoudre un problème
- Spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé

L'élaboration d'un algorithme exige :

- Description des données
- Description des méthodes
- Preuve de bon fonctionnement

La complexité d'un algorithme :

- Temps de calcul,
- Espace nécessaire.

Définitions et Terminologie …

Type

Un **type de données**, ou simplement **type**, définit le genre de contenu d'une donnée et les opérations pouvant être effectuées sur la variable correspondante.

=> Le typage est l'association à un objet :

- un ensemble de valeurs possibles,
- et un ensemble d'opérations admissibles sur ces valeurs

Structure de données

Une **structure de données** est une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir être utilisées efficacement. On distingue :

- Structures de données du langage (types primitifs : Tableaux,…)
- Structures de données abstraites (piles, listes…)

… Définitions et Terminologie

Structure de données abstraite

Un **type abstrait** ou une **structure de données abstraite** est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer. On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter.

Structure dynamique

Une **structure dynamique** est une structure dont la taille peut varier en fonction des besoins.

Ce cours exige la maîtrise des notions de **pointeurs** et de **gestion dynamique de la mémoire**.

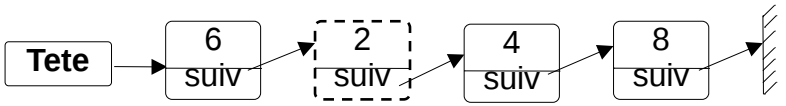
Classification des structures de données

On peut classer les structures de données en deux grandes catégories

- **Les structures de données linéaires**, qui permettent de relier des données en séquence (on peut numéroter les éléments)

√ Tableaux

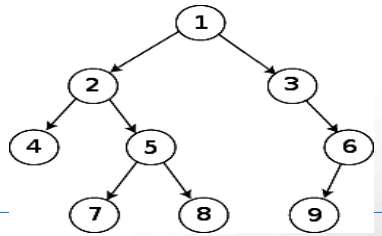
√ Listes
- √ Simplement chaînées
  - √ Piles
  - √ Files
- √ Doublement chaînées



- **Les structures de données non linéaires**, qui permettent de relier un élément à plusieurs autres éléments

√ Arbres
- √Binaire (arbre binaire, arbre binaire de recherche, arbre équilibré, tas)
- √ n-aire

√ Graphes



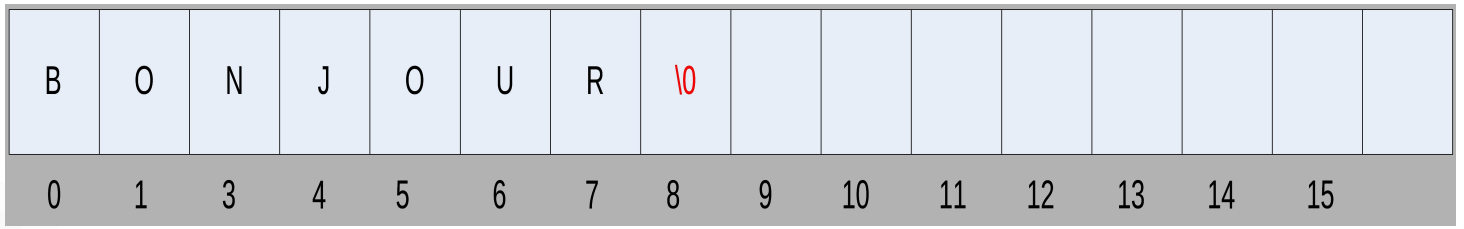
Chapitre 1 : Rappel

- Tableaux
- Enregistrement (structure)
- Pointeurs
- Allocation dynamique
- Fonctions & procédures

1.1) Tableaux

Définition :

- Un tableau est une structure de données composée d'un nombre fixe d'éléments d'un même type. Les éléments d'un tableau sont indicés (numérotés).
- Un tableau occupe dans la mémoire des zones contiguës ayant même longueur



1.1) Tableaux – Déclaration

- Le nombre d'indices correspond à la dimension du tableau :
  - Pour un tableau simple on utilisera un seul indice
  - Pour un tableau de dimension 2 on utilisera deux indices.

VAR nomTableau1D : ***Tableau***[1..1000] de **Type**

VAR nomTableau2D : **Tableau**[1..50] [1..10] de **Type**

VAR Notes : **Tableau**[1..253] de **Réel**

1.1) Tableaux - Accès aux éléments

- L'accès aux éléments d'un tableau simple se fait à l'aide des indices
- Un élément est référencé par la donné du nom du tableau suivie par une expression, renvoyant la valeur de l'indice, entre crochés :

NomDuTableau[expression]

Exemple :

```
A[8] ← 25;

M ← Note[5*2+3]

Pour i depuis 0 JSQ 5 faire
        Note[i] ← 0
FinPour
```

1.2) Enregistrement

Problématique

Besoin d'un mécanisme permettant de grouper des variables de types différents au sein d'une même entité.
Exemple : on veut regrouper une variable de type chaîne de caractères pour le nom, une variable de type entier pour le numéro d'employé, etc.

1.2) Enregistrement

Solutions possibles ?

→ On peut penser à créer une variable pour chaque champ.

Problème : ces variables ne permettent de contenir que des informations d'une seule personne.

→ On peut penser à créer un tableau pour chaque champ.

Problème : les informations concernant une personne sont éparpillées sur plusieurs tableaux on ne peut pas donc parler de l'entité personne.

1.2) Enregistrement

- La réponse à ce besoin est le concept **d'*enregistrement*** (en anglais record) :
- Un enregistrement (appelé aussi **structure** dans certains langages) est un type, ou encore un **méta-type**, complexe construit à partir de types plus simples.
- Un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des ***champs***.

1.2) Enregistrement – Déclaration

Syntaxe de déclaration d'un enregistrement

```
Type identifiant_du_type = structure
        champ1 : type1
        champ2 : type2
          ...
Fin identifiant_du_type
```

Exemple

```
Type Etudiant = structure
    nom, prenom : Chaîne de caractères
    sexe : Caractère
    moyenne : Réel
    CNE : Entier long
Fin Etudiant
```

1.2) Enregistrement - Manipulation

- Déclaration d'une variable de type enregistrement

  VAR identificateur : NomEnregistrement

  Exemple

  VAR Et1 : Etudiant

- La manipulation d'un enregistrement se fait au travers de ses champs.

  nom_enregistrement . nom_champ

  Et1.nom ← "Hanine"
  Et1.prénom ← "Azedine"
  Et1.sexe ← "M"
  Et1.moyenne ← 13.25
  Et1.CNE ← 9997070890

Gestion dynamique de la mémoire

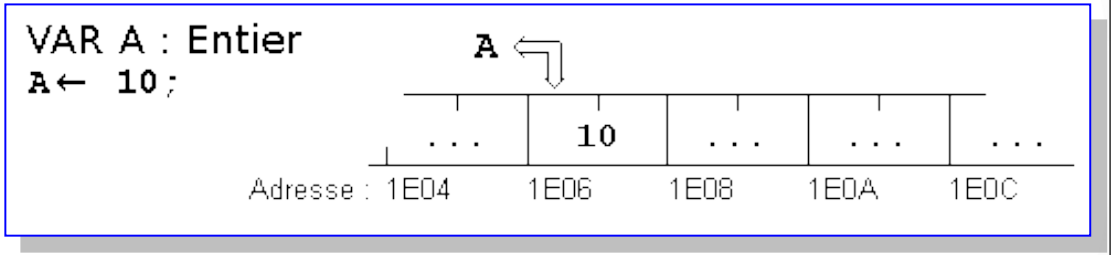
- Pointeurs
- Variables dynamiques
- Tableaux dynamiques : dimension 1

I.3) Pointeurs

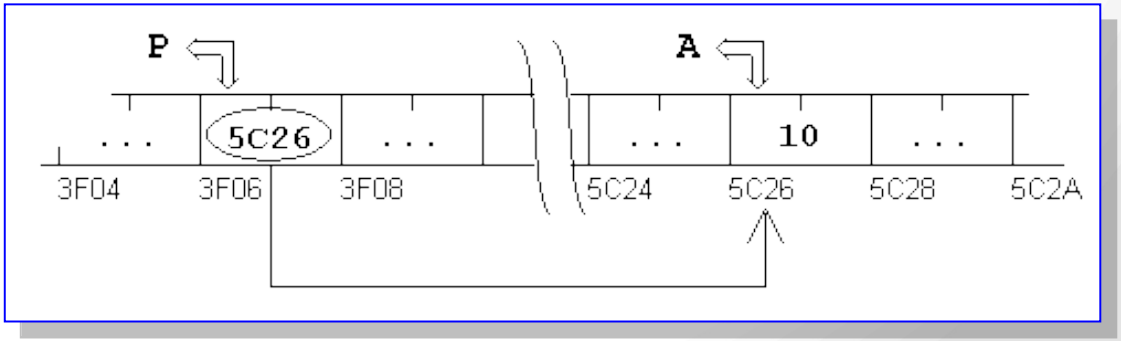
a. Adressage des variables

- Il existe deux modes d'adressage de zone en mémoire :

- ***Adressage direct:*** Accès au contenu d'une variable par le nom de la variable.



- ***Adressage indirect:*** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.



I. Pointeurs

b. Définition

- Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable ou zone mémoire.
- Si un pointeur P contient l'adresse d'une variable A, on dit que : *P pointe sur A*

- Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire. Il faut faire la différence:
  - Un **pointeur** est une variable qui peut 'pointer' vers différentes adresses durant son existence.
  - Le **nom d'une variable** reste toujours lié à la même adresse.

I. Pointeurs

c. Déclaration

La syntaxe de déclaration d'un pointeur :

VAR identificateur :↑ type;

Exemple : VAR np :↑ Entier ;
VAR xp :↑ Réel;

- Un pointeur peut être initialisé à 0, à NULL, ou à une adresse mémoire. Un pointeur de valeur 0 ou NULL ne pointe vers rien.
  - Exemple : np ← 0 ;
    xp ←NULL;

On peut pointer sur n'importe quel type : prédéfini, structure, tableau, chaîne, pointeur, fonction, …

I. Pointeurs

c. Opérateurs

- Opérateur 'adresse de' : **&** pour obtenir l'adresse d'une variable.
  Exemple : np←&A

  *L'opérateur **&** ne peut pas être appliqué à des constantes ou des expressions.*

Opérateur 'contenu de' : ↑ (* en langage c) pour accéder au contenu d'une adresse.
Exemple : np↑←345;

- les pointeurs sont aussi des variables et peuvent être utilisés comme telles:

  p1 ← p2; *p1 pointe vers la même zone que p2*

  i←p1-p2; ***nombre d'éléments entre les deux adresses***

I. Pointeurs

d. Exemple

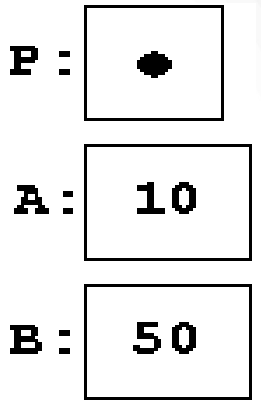
- Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et np un pointeur non initialisé:
- Après les instructions :

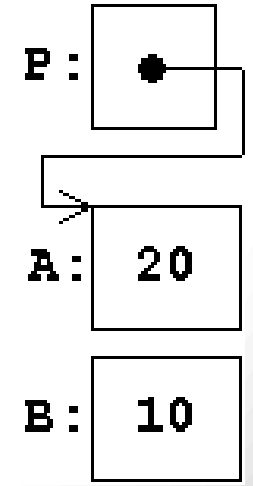
P← &A ; B ← P↑ ; P↑ ← 20 ;

- np pointe sur A ,

- le contenu de A (référencé par P↑) est affecté à B,

- le contenu de A (référencé par P↑) est mis à 20.





I. Pointeurs

e. pointeurs et structures

- Un pointeur peut être utilisé sur des structures:

type *complexe=structure*

reel : Réel

img : Réel

Fin complexe

VAR pc:↑complexe

VAR z:complexe

pc←&z;

- A l'aide d'un pointeur on peut accéder aux champs d'une zone structure par l'opérateur ↑ :

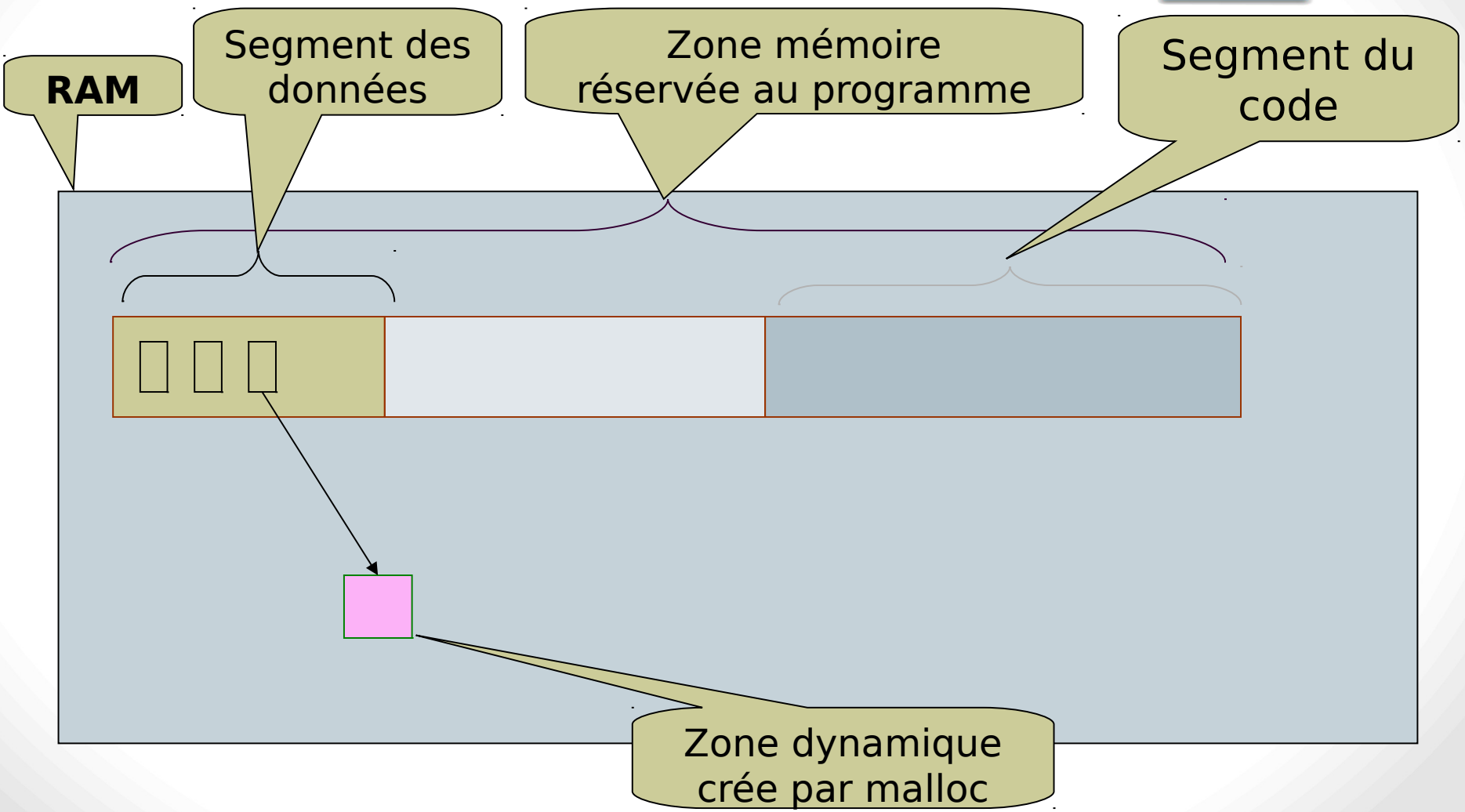
pc↑.reel←0;

pc ↑.mg←0;

Gestion dynamique de la mémoire

- L'utilisation dynamique de la mémoire est basée sur la manipulation des pointeurs par application des fonctions :
  - *Allouer* (en langage c : malloc); allocation d'une zone mémoire ,
  - *Libérer* (en langage c : free); libère une zone mémoire.
- Et les opérateurs :
  - ↑ ; opérateur d'indirection,
  - & ; opérateur d'adressage

Gestion dynamique de la mémoire



Variables dynamiques

- L'accès à une variable dynamique est réalisé à l'aide de pointeurs
- Pour allouer l'espace mémoire pour une seule valeur on écrie Allouer(p)
- Pour allouer l'espace mémoire pour un tableau de nelems on écrie : Allouer(p , nelems)
  - exemple : VAR px, pt: ↑Réel
    Allouer(px)
    Allouer(pt,5)
- Pour libérer une zone allouée, on utilise la fonction libérer : libérer(Nom_pointeur)

Fonctions et procédures

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :
  - permettent de "**factoriser**" les programmes, càd de mettre en commun les parties qui se répètent
  - permettent une **structuration** et une **meilleure lisibilité** des programmes
  - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
  - ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes

Fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat à partir des valeurs des paramètres**

- Une fonction s'écrit en dehors du programme principal sous la forme :

Fonction nom_fonction (paramètres et leurs types) : type_fonction
    Déclaration des variables locales
Début
    Instructions constituant le corps de la fonction
    retourne …
FinFonction

- Pour le choix d'un nom de fonction il faut respecter les mêmes règles que celles pour les noms de variables
- type_fonction est le type du résultat retourné (calculé par la fonction)
- L'instruction **retourne** sert à retourner la valeur du résultat

Fonctions : exemples

- La fonction SommeCarre suivante calcule la somme des carrées de deux réels x et y :

```
Fonction SommeCarre (x : réel, y: réel ) : réel
        variable z : réel
    Début
        z ←x*x+y*y
        retourne (z)
FinFonction
```

- La fonction Pair suivante détermine si un nombre est pair :

```
Fonction  Pair (n : entier ) : booléen
   Début
        retourne (n%2=0)
FinFonction
```

Utilisation des fonctions

- L'utilisation d'une fonction se fera par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, devra être affecté ou être utilisé dans une expression, une écriture, ...

- **Exepmle : Algorithme exepmleAppelFonction**

```
variables     z : réel,  b : booléen
Début
    b ←Pair(3)
    z ←5*SommeCarre(7,2)+1
    écrire("SommeCarre(3,5)= ", SommeCarre(3,5))
Fin
```

- Lors de l'appel Pair(3) le **paramètre formel** n est remplacé par le **paramètre effectif** 3

Procédures

- Dans certains cas, on peut avoir besoin de répéter une tache dans plusieurs endroits du programme, mais que dans cette tache on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois
- Dans ces cas on ne peut pas utiliser une fonction, on utilise une **procédure**
- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**
- Une procédure s'écrit en dehors du programme principal sous la forme :

Procédure nom_procédure (paramètres et leurs types)

Déclarations des variables locales

Début

Instructions constituant le corps de la procédure

FinProcédure

- **Remarque** : une procédure peut ne pas avoir de paramètres

Appel d'une procédure

- L'appel d'une procédure, se fait dans le programme principale ou dans une autre procédure par une instruction indiquant le nom de la procédure :

```
Procédure exemple_proc (…)
        …
FinProcédure

Algorithme exepmleAppelProcédure
Début
        exemple_proc (…)
        …
Fin
```

- **Remarque** : contrairement à l'appel d'une fonction, on ne peut pas affecter la procédure appelée ou l'utiliser dans une expression. L'appel d'une procédure est une instruction **autonome**

Paramètres d'une procédure

- Les paramètres servent à échanger des données entre le programme principale (ou la procédure appelante) et la procédure appelée

- Les paramètres placés dans la déclaration d'une procédure sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)

- Les paramètres placés dans l'appel d'une procédure sont appelés **paramètres effectifs**. ils contiennent les valeurs pour effectuer le traitement

- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre

Transmission des paramètres

Il existe deux modes de transmission de paramètres dans les langages de programmation :

- **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification
- **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la procédure
  - **Remarque :** le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse
- En pseudo-code, on va préciser explicitement le mode de transmission dans la déclaration de la procédure

Transmission des paramètres : exemples

```
Procédure incrementer1 (x : entier par valeur,  y : entier par adresse)
        x ← x+1
        y ← y+1
FinProcédure

Algorithme Test_incrementer1
variables     n, m : entier
Début
    n ← 3
    m ← 3
    incrementer1(n, m)
    écrire (" n= ", n, " et m= ", m)
Fin
```

résultat :

n=3 et m=4

Remarque : l'instruction **x ← x+1** n'a pas de sens avec un passage par valeur

Transmission des paramètres : exemples

```
Procédure incrementer1 (VAR x : entier, y : entier )
        x ← x+1
        y ← y+1
FinProcédure

Algorithme Test_incrementer1
variables     n, m : entier
Début
    n ← 3
    m ← 3
    incrementer1(n, m)
    écrire (" n= ", n, " et m= ", m)
Fin
```

résultat :

n=3 et m=4

Variables locales et globales (1)

- On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie").

- Une **variable locale** n'est connue qu'à l'intérieur du module ou elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution.

- Une **variable globale** est connue par l'ensemble des modules et le programme principale. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme.

Variables locales et globales (2)

- La manière de distinguer la déclaration des variables locales et globales diffère selon le langage
  - En général, les variables déclarées à l'intérieur d'une fonction ou procédure sont considérées comme variables locales
- En pseudo-code, on va adopter cette règle pour les variables locales et on déclarera les variables globales dans le programme principale

- **Conseil :** Il faut utiliser autant que possible des variables locales plutôt que des variables globales. Ceci permet **d'économiser** la mémoire et d'assurer **l'indépendance** de la procédure ou de la fonction

Récursivité

Raisonnement par récurrence (Math)

- Exemple : Montrer que : $$S(n)=\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

  Preuve :

  Cas de base :

  - *n = 0 : (Vrai)*
  - *n = 1 : (Vrai)*

  $$S(0)=\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2} = 0$$

  $$S(1)=\sum_{i=0}^{1} i = 0+1 = \frac{1(1+1)}{2} = 1$$

  Récurrence :

  Hypothèse : S(n) est vrai, n>1

  - *montrer que S(n+1) est vrai*

Récursivité

$$S(n+1) = \sum_{i=0}^{n+1} i = (\sum_{i=0}^{n} i) + (n+1) = S(n) + (n+1)$$

hypothèse de récurrence

⇩

$$= \frac{n(n+1)}{2} + (n+1)$$

$$= (n+1)(\frac{n}{2}+1) = \frac{(n+1)(n+2)}{2} \quad (vrai\ )$$

Récursivité

Équations de récurrence

- Base : S(0) = 0
- Récurrence : S(n+1) = S(n)+(n+1)

☝ La récursivité est un mécanisme de calcul puissant

↳ Résoudre un *problème de taille n* peut se ramener à la résolution *d'un (plusieurs) sous problème(s) de taille plus réduite* ... ⇨

Récursivité

- **Action récursive**

  une action *A* est exprimée de façon *récursive* si la décomposition de A *fait appel* à A (une procédure/fonction qui s'appelle elle-même est dite ***récursive***)

- Action calculer S(n) :

  Appliquer les deux règles suivantes:

  1) S(0) = 0 (règle de base)
  2) S(n) = S(n-1) + n , n>0 (règle de récurrence)

➢ S(3) ?

= S(2) +3 = (S(1)+2) +3

= ((S(0)+1)+2)+3 = ((0+1)+2)+3 = 6

Récursivité

```
procedure recursive(paramètres)
      // déclaration des variables locales
debut
     siTEST_D'ARRET alors
          instructions du point d'arrêt
     sinon  //----- exécution
       instructions
        recursive(paramètres changés)  // appel récursif
        instructions
     FinSi
Fin
```

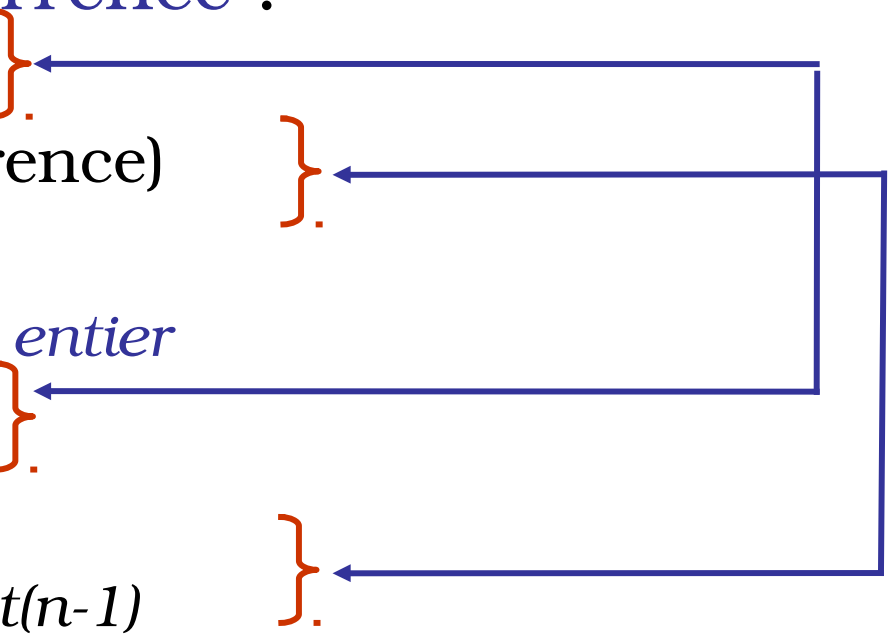
Récursivité : applications

- **Exemple 1 : calcul de la factorielle**

  Équations de récurrence :

  - 0! = 1 (base)
  - n! = n*(n-1)! (récurrence)

```
fonction fact(d n:entier): entier
debut
   si n = 0 alors
      retourne 1
   sinon retourne n * fact(n-1)
   fin si
fin fonction
```

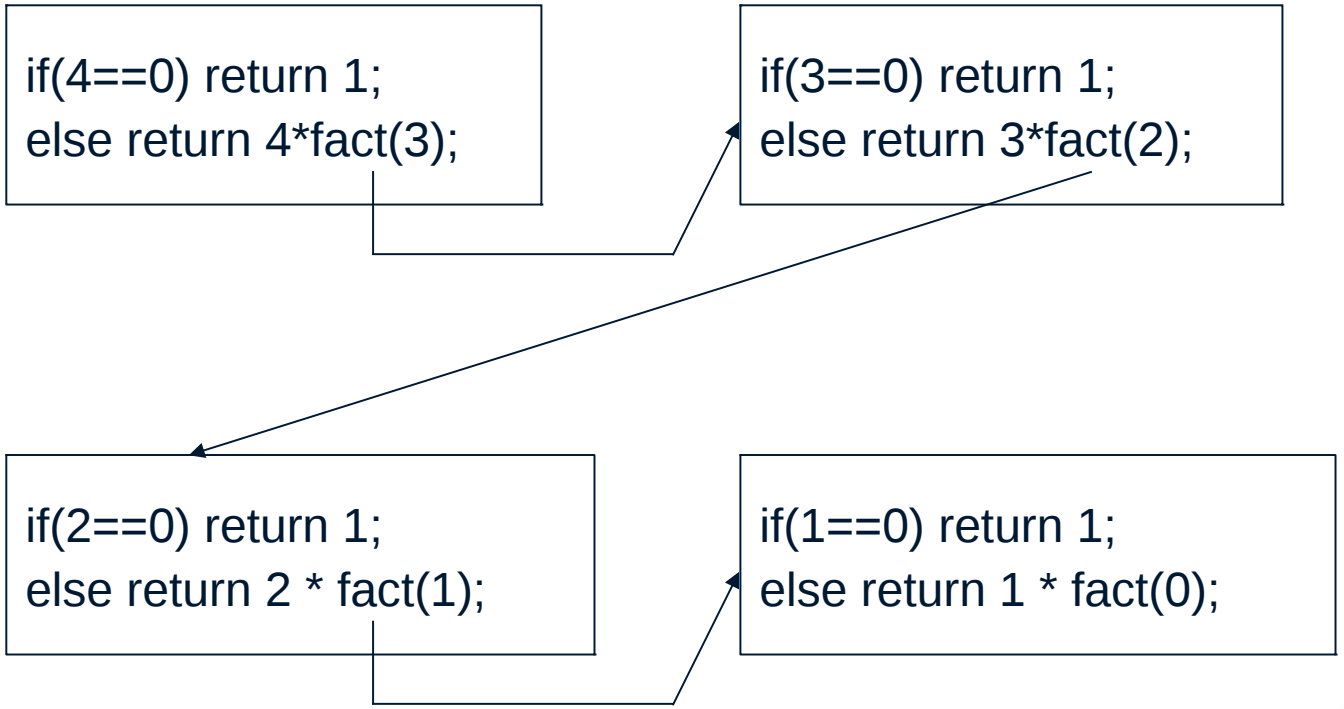


- *Que se passe-t-il si n <0?* ⇨ *récursivité infinie*

Récursivité : applications

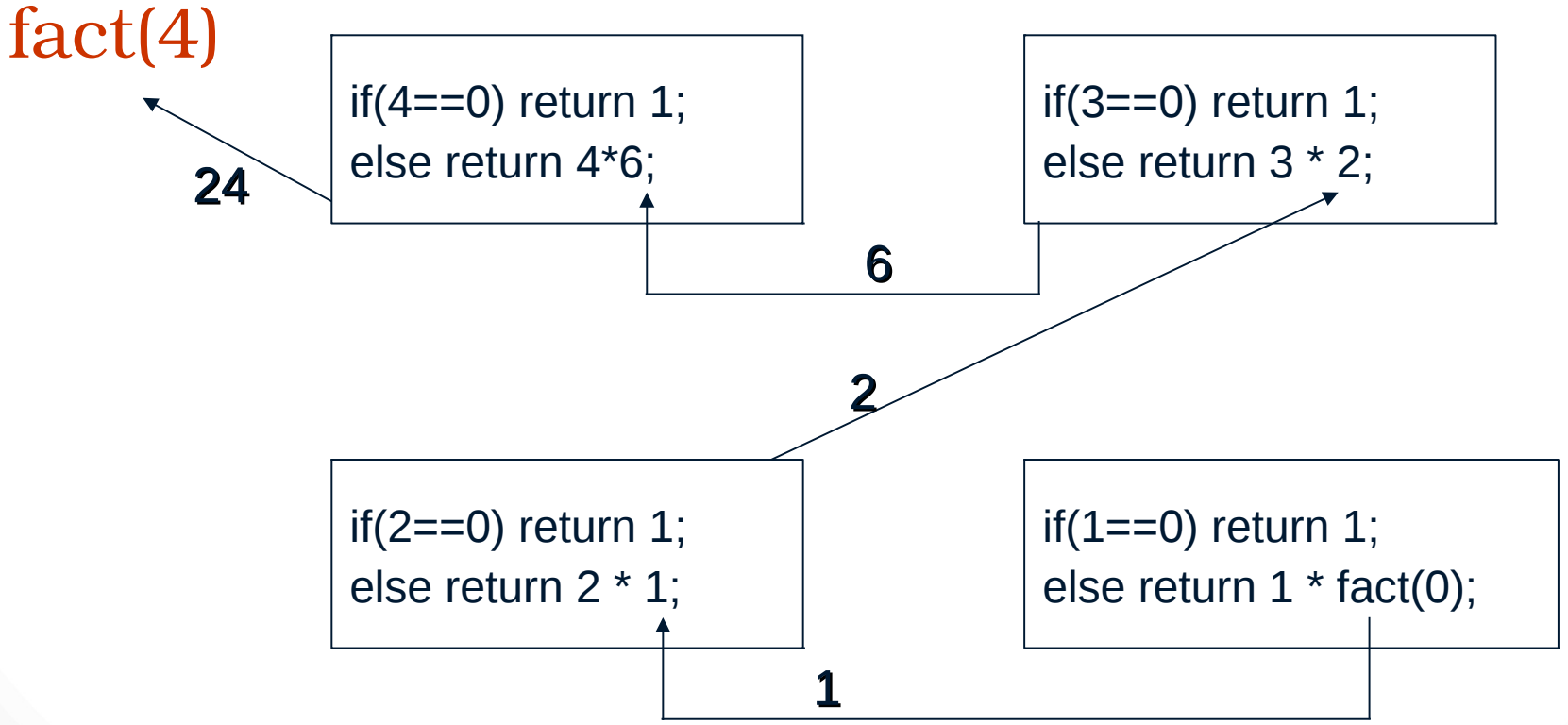
La fonction factorielle : exécution

fact(4)



Récursivité : applications

La fonction factorielle : exécution (suite)



Récursivité : applications

- **Exemple 2 : suite de fibonacci**

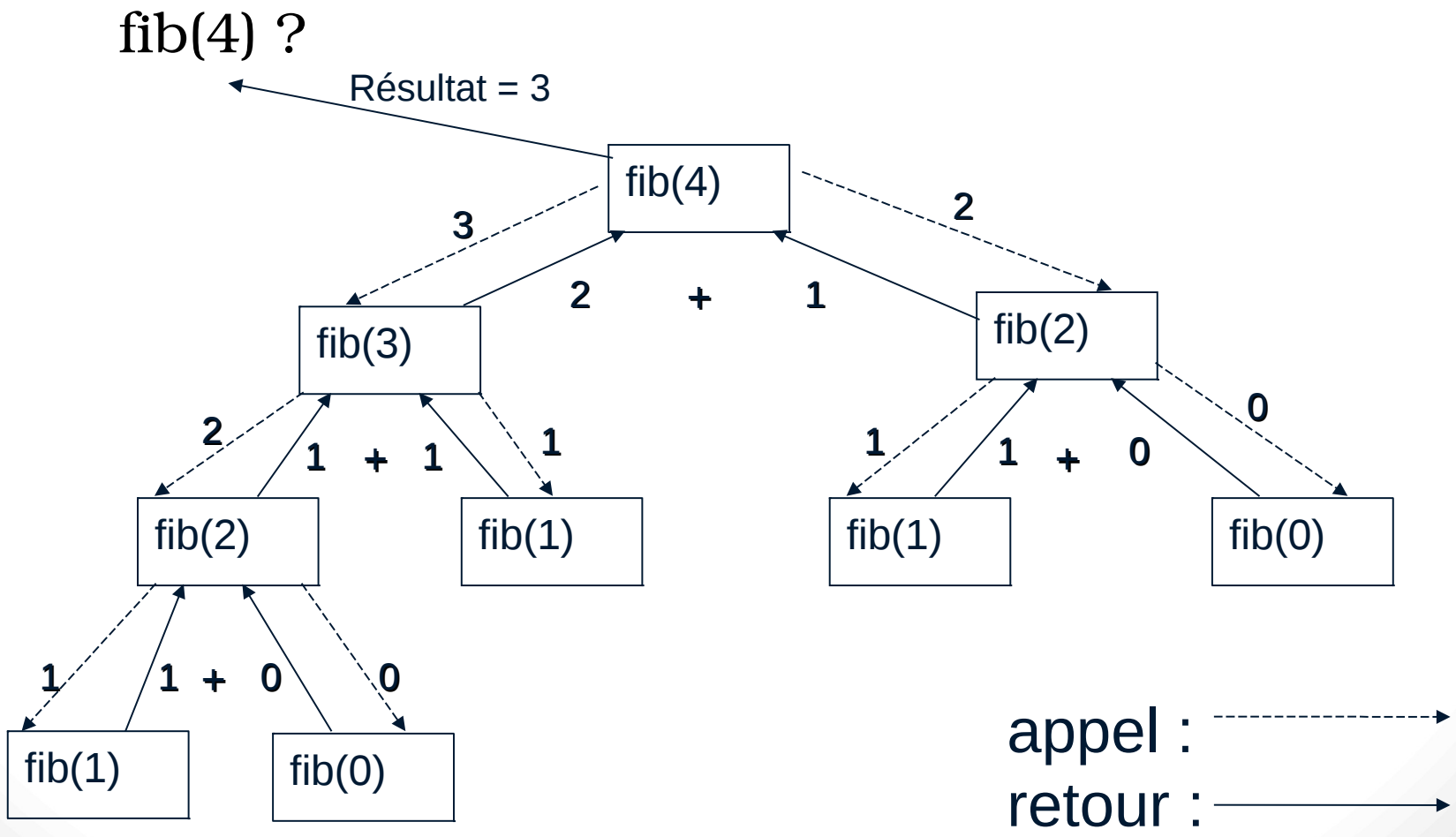
Équations de récurrence :

- u(0) = 0, u(1) = 1 (Base)
- u(n) = u(n-1)+u(n-2), n>1 (récurrence)

```
fonction u( n:entier) : entier
      si (n=0) ou (n=1)
       alors retourne n
      sinon retourne u(n-1) + u(n-2)
      fin si
fin action
```

Récursivité : applications

suite de Fibonacci : exécution



Récursivité : applications

- **Suite de Fibonacci (suite)**

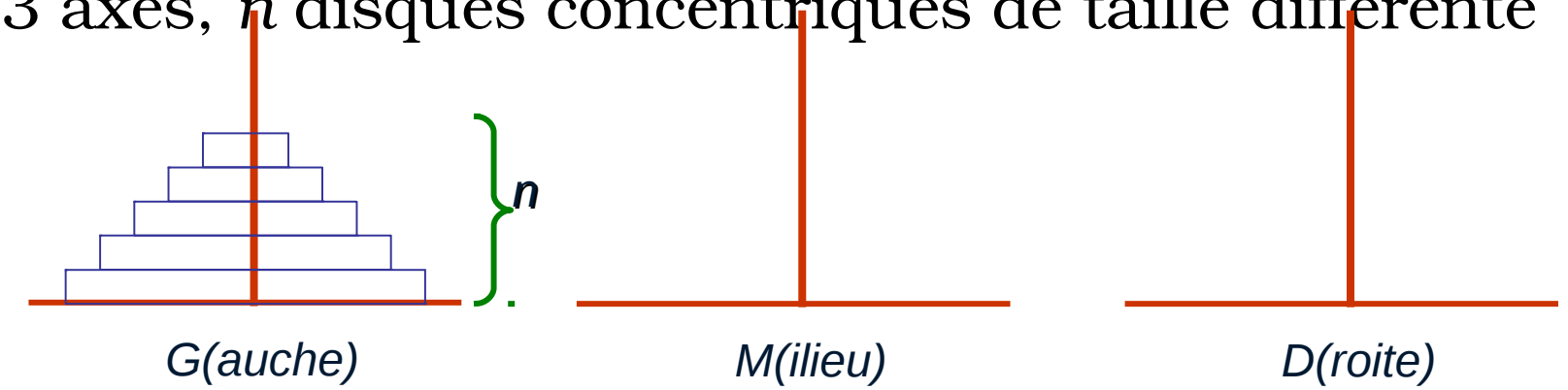
  $\approx O(2^{n-1})$ *appels à la fonction fib*

  pour $n = 100$, $O(2^{99}) \approx O(10^{30})$ *!!!*

- *Une solution récursive n'est pas toujours viable* $\Rightarrow$ solution itérative

Récursivité : applications

- **Les Tours de Hanoï (TH)**
- 3 axes, *n* disques concentriques de taille différente



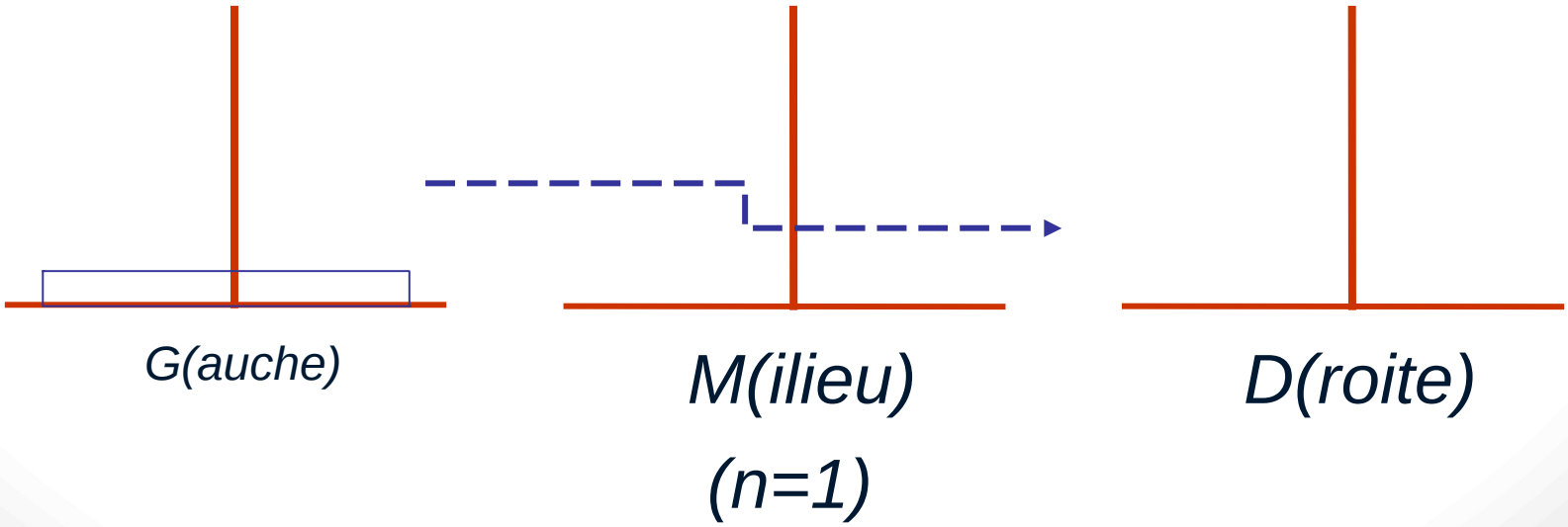
- *But* : déplacer tous les disques de *G* à *D*
- *Règles* :
  - 1 seul disque peut être déplacé à la fois
  - 1 disque ne peut jamais être déposé sur un plus petit
  - 1 disque ne peut être déposé que sur G, M ou D

Récursivité : applications

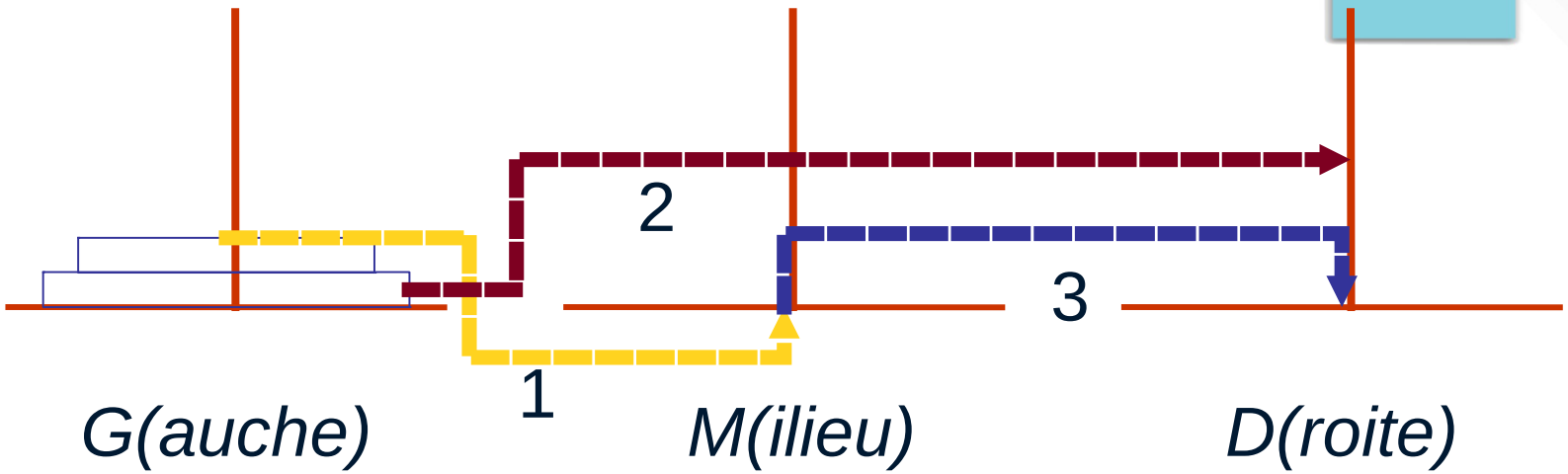
TH : idée de la solution récursive

- *exprimer le problème des n disques à l'aide de la même solution sur n-1 disques*
- *(base) savoir résoudre le problème pour 1 disque*

Équations de récurrence



Récursivité : applications



(n=2)

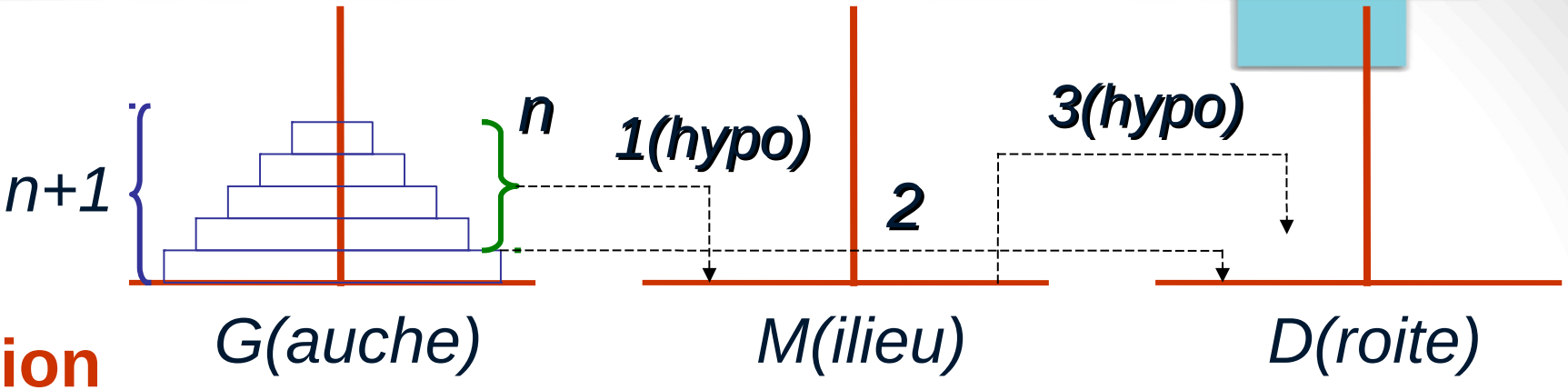
(n=3) …, bonne chance

Récurrence :

Hypothèse: on connaît la solution pour ***n*** disques

Trouver la solution pour ***n+1*** disques

Récursivité: applications



Solution

procédure déplacer(n, G, M, D)

" déplacer n disques de G vers D en utilisant l'axe M "

Début

Si n=1 alors

déplacer un disque de G vers D

sinon

déplacer(n-1, G, D, M)

déplacer un disque de G vers D

- *déplacer(n-1, M, G, D)*

Finsi

Fin

Exercice …

Un compte en banque concerne une personne spécifiée par son nom, un numéro de compte (un entier), et un montant (réel).

1) Déclarez un enregistrement pour cette structure "**Compte**"
2) Écrire un programme qui permet de créer un tableau de taille N et de type compte (la valeur de N doit être saisie au clavier par l'utilisateur). Remplir le tableau par des valeurs entrées au clavier.
3) Écrire une procédure qui permet de trier le tableau par ordre croissant des montants.
4) Écrire une procédure qui permet d'afficher les informations d'un compte dont le numéro est passé en paramètres.

…Exercice

1)

Type Compte=Enregistrement

nom : chaîne de caractères

num : Entier

montant : Réel

Fin Compte

…Exercice

```
Algorithme Compte_bancaire
    VAR TCB :↑ Compte
            i,N : Entier
Début
    Ecrire("donner la taille du tableau")
    lire(N)
    Allouer(TCB,N)
    Pour i depuis 1 JSQ N faire
        Ecrire("Compte N° ",i,"\n")
        Ecrire("donner le Nom")
        lire(TCB[i].nom)
        Ecrire("donner le numéro du compte")
        lire(TCB[i].num)
        Ecrire("donner le montant du compte")
        lire(TCB[i].montant)
    FinPour
    libérer(TCB)
Fin
```

…Exercice

```
Fonction indiceMax(t:Tableau de Compte, d:Entier, f : Entier) : Entier
    VAR i,indice : Entier
Début
    indice←d
    Pour i depuis d+1 JSQ f faire
        Si t[i].montant>t[indice].montant alors
            indice ←i
        FinSi
    FinPour
    retourne indice
finFonction
Procédure permute( VAR a : Compte, VAR b : Compte)
    VAR aux : Compte
Début
    aux←a
    a←b
    b←aux
Fin
```

…Exercice

```
Procédure TrieTC(VAR t : tableau de Compte, N:Entier)
    VAR i,ind_max:Entier
Début
    Pour i depuis 1 JSQ N-1 faire
        ind_max←indiceMax(t,i,N)
        Si ind_max <> i alors
            Permute(t[i],t[ind_max])
        FinSi
    FinPour
Fin
```

…Exercice

```
Procédure Affiche(t:Tableau de Compte, N:Entier, num:Entier)
    VAR i : Entier
Début
    i←1
    Tant que(i<=N et t[i].num<>num)
        i←i+1
    FinTantque
    Si i<=N alors
        Ecrire("Nom :",t[i].nom)
        Ecrire("Numéro du compte :", num)
        Ecrier("Montant du compte : ", t[i].montant)
    FinSi
Fin
```

Chapitre 2 :**Structures de données abstraites**

1) Notion d'abstraction en informatique

2)Spécification abstraite (TAD)

- Signature
- Propriétés

2) Spécification opérationnelle (Structures de données concrètes)

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’implémentation de cet objet.

Notion de Type Abstrait de Données

- Un **type abstrait** est une spécification de données décrivant l'ensemble des opérations associées à ces données et l'ensemble des **propriétés** de ces **opérations**. Cette spécification ne précise pas la représentation interne.
- De ce fait, les algorithmes sont indépendants de la représentation interne des données.

Notion de Type Abstrait de Données

Nous pouvons distinguer deux sortes de types abstraits :

- Les **types abstraits de base** qui correspondent aux types disponibles dans la plupart des langages de programmation (entier, réel, caractère…).
- Les **types abstraits construits** aux moyens de types abstraits de base ou bien d'autres types abstraits construits prédéfinis.

Exemples :

- Dans un algorithme qui manipule des entiers , on s'intéresse, non pas à la représentation des entiers, mais aux opérations définies sur les entiers : + , - , * , /
- Type booléen , ensemble de deux valeurs (faux , vrai) muni des opérations : non , et , ou

Spécification d'une structure de données

Une structure de données peut être **spécifiée** 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'implémentation informatique choisie pour représenter et décrire la structure de donnée.

Spécification abstraite

Un type abstrait (spécification abstraite) est la donnée

- de sa **signature** décrivant la syntaxe du type avec le nom des opérations,
- des **propriétés** des opérations du type qui seront données sous forme d'un ensemble **d'axiomes** (formules logiques). On introduit alors une **sémantique** (une signification) aux objets de la signature.

Signature d'un TAD

- Comporte :
  - Le nom du TAD ;
  - Les noms des types des objets utilisés par le TAD ;
  - Pour chaque opération, l'énoncé des types des objets qu'elle reçoit et qu'elle renvoie.
- Décrite par les paragraphes :
  - Sorte
  - Paramètre
  - Utilise
  - Opérations

Syntaxe d'écriture de la signature d'un type abstrait de données

Sorte : ..... *//les noms de types définis par le TAD, ce sont les types au //sens des* langages de programmation.

Paramètre: ……*//Pour les TAD collections, le nom générique du type //des éléments stockés*

Utilise : ..... *// Types déjà définis (types de bases ou types //construits) utilisés par le TAD*

Opérations : *//Cette partie décrit la syntaxe du TAD*

nom1: $Type_1$ x $Type_2$ x ... → $Type'_1$ x $Type'_2$ x $Type'_3$ x …..
nom2: $Type_1$ x $Type_2$ x . . . → $Type'_1$ x $Type'_2$ x $Type'_3$ x …..

Signature d'un TAD
Exemple : TAD Booléen

Sorte : Booléen

Le nom de TAD

Opérations :

vrai : → Booléen

faux : → Booléen

Les constantes sont représentées sous forme des opérations qui n'ont pas d'arguments

non : Booléen → Booléen

et : Booléen x Booléen → Booléen

ou : Booléen x Booléen → Booléen

Type valeur de retour

Les noms des opérations

Deux arguments de type Booléen

Signature d'un TAD
Exemple : TAD Booléen

Sorte : Booléens

Opérations :

vrai : → Booléens

faux : → Booléens

¬_ : Booléens → Booléens

_ ∧ _: Booléens x Booléens → Booléens

_∨_ : Booléens x Booléens → Booléens

- Les constantes sont représentées sous forme des opérations qui n'ont pas d'arguments
  Exemple : **Vrai :** → **Booléen**

Exemple

Sorte séquence-réelle
Utilise entier, réel
Opérations

séquence-réelle : entier → séquence-réelle
ième : séquence-réelle × entier → réel
changer-ième : séquence-réelle × entier × réel
→ séquence-réelle
taille : séquence-réelle → entier

Exemple

Sorte séquence-entier
Utilise entier
Opérations

séquence-entier : entier → séquence-entier
ième : séquence-entier × entier → entier
changer-ième : séquence-entier × entier × entier
→ séquence-entier
taille : séquence-entier → entier

Remarque

La définition d'une séquence doit être **générique**, c'est-à-dire permettre de varier les types d'éléments qu'elle manipule.

Exemple TAD séquence générique

Sorte séquence
Utilise entier
Paramètre élément
Opérations

seq: entier → séquence
ième : séquence × entier → élément
changer-ième : séquence × entier × élément
→ séquence
taille : séquence → entier

Propriétés d'un type abstrait (sémantique)

- Donne une sémantique (signification) aux sortes et aux opérations de la signature au moyen d'axiomes agissant sur les variables.
- Précise :
  - Les domaines de définition (ou d'application) des opérations;
  - Les propriétés des opérations.
- Décrite par les paragraphes :
  - Préconditions
  - Axiomes
- On parle d'un Type Abstrait Algébrique (TAA) lorsque la sémantique est définie par un système d'équations

Syntaxe d'écriture des propriétés d'un type abstrait de données

Axiomes / Sémantique: *// décrivant sans ambiguïté ses opérations*

- axiome$_1$ *// Axiome : décrit logiquement ce*
//que fait une composition d'opérations

. . . *// Sémantique : explique ce que fait chaque opération*

- axiome $_n$

Préconditions : *// Le domaine d'une opération définie*
// partiellement est défini par une précondition

..... **est définie ssi**.....

Exemple (suite d'exemple de TAD séquence)

Axiomes:

∀ s :séquence, ∀ i : entier, ∀r : réel

ième(changer-ième(s, i, r), i) = r

ième(changer-ième(s, i, r), j) = ième(s, j) si i # j

taille(changer-ième(s, i, r)) = taille(s)

Préconditions:

ième(s, i) **est définie ssi** borne-inf(s) ≤ i ≤ borne-sup (s)

changer-ième(s, i, r) **est définie ssi** borne-inf(s) ≤ i ≤ borne-sup (s)

Opérateurs

Les opérations sont divisées en plusieurs types :

- Les **constructeurs** : ils permettent de créer un nouvel "objet" du type que l'on est en train de définir.
- les **modificateurs** : ils permettent de modifier les objets et leur contenu.
- les **accesseurs:** ils donnent des informations sur l'état de l'objet (valeurs, affichages, …)

Constante : opérateur sans argument

Important
On ne manipule un type abstrait qu'avec les fonctions qui lui sont associées !!

Formalisme des types abstraits

Sorte : ……..

Paramètre: …….//*Pour les TAD collections, le nom générique du type des éléments stockés*

Utilise : ..... *// Types déjà définis (types de bases ou types construits) utilisés par le TAD*

Opérations : //*Cette partie décrit la syntaxe du TAD*

nom1: $Type_1$ x $Type_2$ x . . . → $Type'_1$ x $Type'_2$ x $Type'_3$ x …..

nom2: $Type_1$ x $Type_2$ x . . . → $Type'_1$ x $Type'_2$ x $Type'_3$ x …..

. . .

Axiomes : //

- $axiome_1$ *// Axiome : décrit logiquement ce que fait une composition d'opérations*

. . . *// axiomes définis les propriétés des opérations*

- $axiome_n$ *// Sémantique : explique ce que fait chaque opération*

Préconditions : *// Le domaine d'une opération définie partiellement est défini*
// par une précondition

..... **est définie ssi**.....

FinTDA …..

Règles lors de l'analyse

A la définition d'un TAD, il faut faire attention à:

- la complétude : A-t-on donné un nombre suffisant d'opérations pour décrire toutes les propriétés du type abstrait ?
- la consistance (non contradictoire) : N'y a-t-il pas des compositions d'axiomes contradictoires ?

Comment bien définir les opérations d'un TAD ?

- Choisir des identifiants significatifs
- Bien préciser les conditions de bon fonctionnement (préconditions)

TAD Booléen

Sorte : Booléen

Opérations :

vrai: → Booléen

Faux : → Booléen

Non Booléen Booléen

Et : Booléen × Booléen → Booléen

Ou : Booléen × Booléen → Booléen

Préconditions : Aucune précondition

Axiomes :

∀ a, b : Booléen

non(vrai) = faux

non(non(a)) = a

vrai et a = a

faux et a = faux

a ou b = non(non(a) et non(b))

FinTAD Booléen

TAD Séquence

Sorte Séquence
Paramètre élément
Utilise entier, booléen *//propriétés utilisées*
Opération
seq : entier x élément → séquence *// **constructeur***
changer-ième : séquence x entier x élément → séquence *// **modificateur***
ième : séquence x entier → élément
borne-sup : séquence → entier
borne-inf : séquence → entier
défini : séquence x entier → booléen
Axiomes
∀ s: Séquence, ∀i: entire, ∀k: entire, ∀e: élément
ième(changer-ième(s,i,e),i) = e
ième(changer-ième(s,i,e), j) = e **si** i=j **sinon** ième(s,j) **fsi**
Préconditions
ième(t,k) **est définie ssi** défini(t,k)
changer-ième(s,k,e) **est définie ssi** borne-inf(s) ≤ k ≤ borne-sup(t)
FinTAD Séquence

TAD Réel

Sorte : Réel
Utilise : Booléen
Opérations :

[0-9]$^{+}$*.[0-9]** : → Réel
_+_ : Réel × Réel → Réel
_-_ : Réel × Réel → Réel
_*_ : Réel × Réel → Réel
_/_ : Réel × Réel → Réel
_<_ : Réel × Réel → Booléen

Préconditions :

∀ a,b : Réel

a / b **est définie ssi** b ≠ 0

FinTAD Réel

… TAD Réel

Remarque

[0-9]$^{+}$ (en italique) est une expression rationnelle (appelée aussi expression régulière)

[0-9]$^{+}$**,***[0-9]** représente ici tous les identifiants d'opération composés uniquement de chiffres (rôle du *[0-9]*) avec au moins un chiffre (rôle du +) suivi d'un virgule et optionnellement d'autant de chiffres que l'on veut (rôle du *).

TAD Caractère

Sorte : Caractere
Utilise: Naturel
Opérations:

'[.]': → Caractere
car: Naturel → Caractere
pred: Caractere → Caractere
succ: Caractere → Caractere
ord: Caractere → Naturel

TAD Tableau

Sorte tableau
Paramètre élément
Utilise entier
Opérations

table : entier × element → tableau
ième : tableau × entier →élément
changer-ième : tableau × entier × élément → tableau
taille : tableau → entier

FinTAD tableau

TAD Structure

Sorte structure
Paramètres E1, E2, . . ., En
Opérations

structure : Idn × E1 × E2... × En → structure
e1 : structure → E1
e2 : structure → E2
. . .
en : structure → En
changer-e1 : E1 × structure → structure
changer-e2 : E2 × structure → structure
. . .
changer-en : En × structure → structure

Axiomes

∀ s : structure, ∀ v : E_i

e_i(changer-e_i(v,s)) = v avec i=1 , 2, …n

FinTAD Structure

Réutilisation des TADs

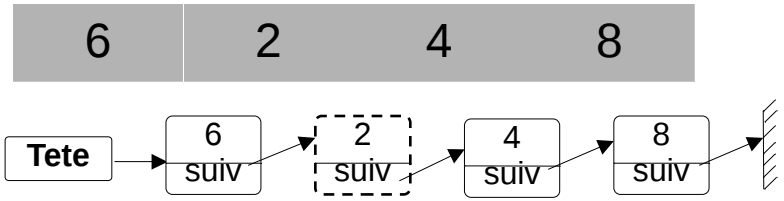
- Quand on définit un type, on peut réutiliser des types déjà définis
- La signature du type défini est l'union des signatures des types utilisés enrichie des nouvelles opérations
- Le type hérite des propriétés des types qui le constituent
- Exemples :
  - Types ***Entier*** et ***Elément*** utilisés par le TAD ***Séquence***

Spécification opérationnelle

- La spécification opérationnelle correspond à l'implémentation d'un TAD.
- Composée d'un algorithme pour chaque opération, plus éventuellement des données spécifiques à la structure pour sa gestion
- Un même TAD peut donner lieu à plusieurs structures de données, avec des performances différentes

Spécification opérationnelle

- Il existe deux façons de représenter (d'implémenter) les structures de données :
  - par tableau
  - chaînage dynamique.



- Lors du passage à la programmation
  - Les TAD sont implémentés par des types de données c.-à-d. par des structures appelée aussi des enregistrements (classes dans le cas de Programmation Orientée Objets) ;
  - Les opérations sont implémentées par des sous-programmes (méthode en POO).
- Remarque : Les constantes (Exemple vrai et faux pour le TAD booléen) sont représentées par l'énumération.

Spécification opérationnelle Implémentation d'un TAD en C

- **Utiliser la programmation modulaire :**
  - Programme découpé en plusieurs fichiers, même de petites tailles (réutilisabilité, lisibilité, etc.)
  - Chaque composante logique (un module) regroupe les fonctions et types autour d'un même thème.
- **Pour chaque module *truc* , créer deux fichiers :**
  - fichier ***truc.h*** : l'interface (la partie publique) ; contient la spécification de la structure ;
  - fichier ***truc.c*** : la définition (la partie privée) ; contient la réalisation des opérations fournies par la structure. Il contient au début l'inclusion du fichier ***truc.h***
- **Tout module ou programme principal qui a besoin d'utiliser les fonctions du module *truc* , devra juste inclure le *truc.h***
- **Un module C implémente un TAD :**
  - ***L'encapsulation*** : détails d'implémentation cachés ; l'interface est la partie visible à un utilisateur
  - ***La réutilisation :*** placer les deux fichiers du module dans le répertoire où l'on développe l'application.

Exemple – structure de données booléen
Rappel sur les énumérations

- Un type énuméré permet de définir des valeurs n'existant pas dans les types fournis par le langage.
- Un type énuméré s'utilise comme n'importe quel type pour typer des variables ou des paramètres.
- En algorithmique, pour les types énumérés, on adoptera la syntaxe la plus courante qui est similaire à celle des enregistrement.
- Syntaxe

```
type IdentificateurTypeEnum = enum
    Liste des valeurs
fin Enum
```

- Exemple

```
Type  ArcEnCiel=enum
    rouge, orange, jaune, vert, bleu, indigo, magenta, violet
Fin Enum
```

Exemple – structure de données booléen

```
Type  Booléen=enum
    vrai, faux
Fin Enum
Fonction et(x : Booleen ,y :  Booleen ) : Booleen
Debut
    si (x = faux) alors
        retourne  faux
    sinon
         retourne y
    FinSi
Fin et
Fonction  ou(x : Booleen ,y :  Booleen ): Booleen
debut
    si (x = vrai) alors
        retourne vrai
    Sinon
         retourne y;
    Finsi
Fin ou
```

```
Fonction non(x : Booleen ) : Booleen
Debut
    si (x = vrai) alors
        retourne faux;
    Sinon
        retourne vrai;
    Finsi
Fin
```

Structures de données dynamiques

- Il existe deux types d’allocation de mémoire :
  - statique : Allocation de mémoire effectuée lors de l'exécution mais prévue lors de la compilation.
  - Dynamique : Allocation de mémoire effectuée lors de l'exécution mais non prévue lors de la compilation.
    - Structures dynamiques contiguës
    - Structures dynamiques chaînées

Synthèse

Une structure de données peut être **spécifiée** 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'implémentation informatique choisie pour représenter et décrire la structure de donnée.

C/c

1) Spécification abstraite

Sorte : .....
Paramètre: …….
Utilise : .....
Opérations :
. . .

Signature

Axiomes :
- $axiome_1$
. . .
- $axiome_n$

Préconditions :
..... **est définie ssi**.....
FinTDA …..

Propriétés

Formalisme d'un type abstrait

2) Une spécification opérationnelle : implémentation

Chapitre 3

Structures de Données Linéaires
Listes, Piles & Files