Calendrier
M2 CCI – Algorithmique

- Semaines bloquées (21h)
  - de rentrée septembre : 7h30 de cours, 7h30 de TD
  - décembre : 3h de Cours, 3h de TD
- Septembre à début mars (16 semaines - 72h)
  - 3 séances hebdomadaires d'1h30
    - *1 Cours , 1 TD*
    - *et , en alternance, 1 Cours ou 1 TD*
  - 1 séance de « tutorat » de temps en temps
- Travail personnel régulier
  - Lien entre les travaux dirigés et le cours
  - Lien avec l'enseignement de programmation
  - Devoirs à la maison.

Organisation
M2 CCI – Algorithmique

Enseignants

- Cours
  Catherine.Vigouroux@univ-grenoble-alpes.fr
  Laboratoire VERIMAG
- Travaux dirigés
  Cristian.Ene@univ-grenoble-alpes.fr
  Laboratoire VERIMAG

Documents

- Polycopié
- Page AL sur Moodle :
  - présentation enseignement, planning général + cours/TD, diaporamas cours, poly TD, annales

Contrôle de connaissances

- Contrôle continu (CC)
  - Interrogations écrites (1h) : fin octobre, fin janvier
  - Devoir surveillé (2h) début décembre (sans documents)
- Examen écrit (3h) début mars (sans documents)
- Règlement d'examen : CC 1/3 , Examen 2/3

Lignes directrices
M2 CCI – Algorithmique

- Développer la capacité d'abstraction
  - Lecture d'énoncés
  - Analyse de problèmes
  - Lecture d'algorithmes, de programmes
- Installer de "bonnes habitudes", des réflexes
  - Pour aborder un problème, rechercher des solutions
  - Pour utiliser correctement les moyens d 'expression
  - Pour utiliser des techniques répertoriées
  - Pour examiner les propriétés des algorithmes

Plan du cours
M2 CCI – Algorithmique

1. Fonctions, expressions, valeurs
2. Raisonner sur les actions
3. Séquences et tableaux
4. Séquences et chaînage
5. Définitions récursives
6. Arbres
7. Actions réalisées récursivement
8. Arbres et forêts : représentation chaînée

1. Fonctions, expressions, valeurs

1.0. Exemples introductifs

Langage algorithmique : trois points de vue

Concepts, composants du langage, éléments d'analyse

1.0. Exemples introductifs

1.1. Types et valeurs, modélisation de l'information

1.2. Composition de fonctions, analyse descendante

1.3. Composition conditionnelle, analyse par cas

1.4. Types construits, informations complexes

⚠ S'approprier le langage des expressions
Savoir calculer le type d'une expression

Moyenne olympique

Spécification

Voir polycopié E1.1

Déterminer la moyenne olympique de quatre entiers

- Comprendre l'énoncé avec des exemples
  - **10, 8, 12, 24 → 11**
  - **24, 14, 14, 14 → 14**
- Spécifier le problème sous forme d'une fonction

  mo **: fonction (**u, v, w, x **: entier ≥ 0) → réel ≥ 0**
- Idée(s) d'algorithme ?

  sommer, soustraire le minimum et le maximum, diviser par 2

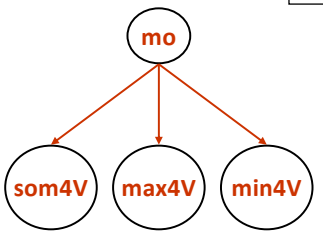
Moyenne olympique

Analyse descendante

2) Spécifier les fonctions intermédiaires

som4V **: fonction (**u,v,w,x **: entier ≥ 0) → entier ≥ 0**
min4V **: fonction (**u,v,w,x **: entier ≥ 0) → entier ≥ 0**
max4V **: fonction (**u,v,w,x **: entier ≥ 0) → entier ≥ 0**

1) Décomposer



3) Réaliser la fonction

```
mo(u, v, w, x) :
    retour :
      ( som4V(u, v, w, x)
          – min4V(u, v, w, x)
          – max4V(u, v, w, x)
      ) / 2
```

Noms formels de paramètres

Expression algébrique décrivant la fonction

4) Analyser les fonctions intermédiaires

voir Polycopié, exemple E1.1

Maximum de trois valeurs

"si … alors … sinon … "

Déterminer le maximum de trois entiers distincts deux à deux

max3V **: fonction (**a,b,c **: entier) → entier** *{Pré-condition : distincts 2 à 2}*

Idées
1. Comparer les valeurs deux par deux
2. Raisonner directement sur le résultat

Idée 1 : utiliser le maximum de deux valeurs

- Spécifier une fonction intermédiaire

  max2V **: fonction (**x,y **: entier) → entier**
- Réaliser max3V en utilisant max2V

  max3V**(**a, b, c**) : retour :** max2V**(**max2V**(**a, b**),** c**)**
- Réaliser max2V

  max2V**(**x, y**) : retour : si** x **>** y **alors** x **sinon** y

Expression conditionnelle

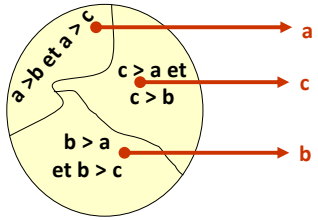
Maximum de trois valeurs

Analyse par cas, "selon ..."

Idée 2 : raisonner directement sur le résultat.

Ce ne peut être que l'une des trois données.

Partition du domaine



Réalisation

```
max3V(a, b, c) :
  retour :
    selon a, b, c
      a > b et a > c : a
      b > a et b > c : b
      c > a et c > b : c
```

Expression conditionnelle

Pour continuer

autres idées : polycopié, E1.2

généralisation à n nombres

1.1. Types et valeurs

fonctions, expressions, valeurs

1.0. Exemples introductifs

1.1. Types et valeurs

1.2. Composition de fonctions

1.3. Composition conditionnelle

1.4. Types construits

a) Notion de type

Définitions

- Valeurs
  - Rôle : représenter les informations du monde réel
  - Toute valeur a un type
- Types
  - Rôle : modéliser les valeurs possibles
  - Concept : ensemble de valeurs + famille d'opérations
  - Du point de vue langage
    - *nom du type*
    - *définition de l'ensemble de valeurs*
    - *convention de dénotation des valeurs*
    - *spécification des opérations*

b) Types de base

Définitions

Nom du type	entier	réel	caractère	booléen
Dénotation des valeurs	563	35.4	'a' '4' ';' ' '	vrai faux
opérations	+, -, *, / reste, quotient	+, -, *, /		et, ou, non

Opérations de comparaison

- =, <, ≠, ≤, >, ≥
- Opérandes de même type
- Type muni d'une relation d'ordre
- Résultat de type booléen

Il suffit de définir = et <	
A ≠ B	non (A=B)
A ≤ B	(A < B) ou (A = B)
A > B	B < A
A ≥ B	non (A < B)

c) Le type texte

Définitions

Nom : texte

Ensemble de valeurs : séquences de caractères

Dénotation des valeurs : "exemple", "a"

Opérations :

- texte vide : [ ]
- ajout à gauche : c ∘ t
- ajout à droite : t • c
- concaténation : t1 & t2

de type caractère

de type texte

Expression	Valeur
'a' ∘ "bcd"	"abcd"
"abc" • 'd'	"abcd"
"ab" & "cd"	"abcd"
"a" & "bcd"	"abcd"
'a' & "bcd"	

Propriétés : c ∘ t ≡ [c] & t t • c ≡ t & [c]

Autres opérations : voir Polycopié

EstVide, premier, fin, début, dernier

1.2. Composition de fonctions

fonctions, expressions, valeurs

1.0. Exemples introductifs
1.1. Types et valeurs
1.2. Composition de fonctions
1.3. Composition conditionnelle
1.4. Types construits

a) Expression arithmétique

Définition

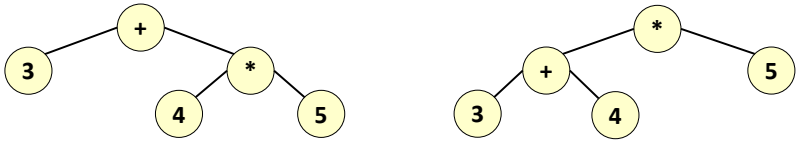
- Une expression est dite arithmétique si tout opérande dénote une valeur.
- Une expression arithmétique a un type et dénote une valeur

L'expression	de type	dénote la valeur
3+(4×5)	**entier**	**23**
(faux ou vrai) et vrai	**booléen**	**vrai**
(10 > 100) et vrai ou 'a' = 'b'	**booléen**	**faux**

b) Parenthésage, priorités d'opérateurs

- Quelle est la signification (valeur) de l 'expression 3 + 4 * 5 ?
- Dans quel ordre, les opérateurs sont-ils appliqués?

3 + (4 * 5) = 23 **OU** **(3 + 4) * 5 = 35**



3 + 4 * 5 → 3 + 20 → 23

(3 + 4) * 5 → 7 * 5 → 35

Définition

L'opérateur * est plus prioritaire que l 'opérateur +

La valeur de l'expression 3+4*5 est 23

c) Expression algébrique

Définition

- Dans une expression algébrique les opérandes peuvent être des noms.
- Une expression algébrique a un type et décrit une fonction
- Une expression algébrique ne peut être évaluée que dans un contexte qui fixe la valeur de chacun des noms apparaissant dans l'expression.

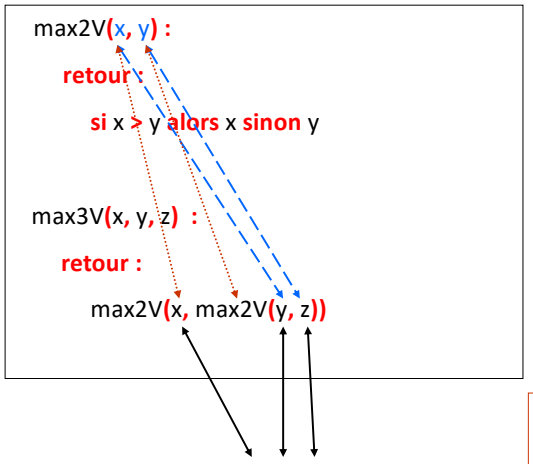
- a + (2 * b) est une expression de type
  entier, entier → entier
- (a < 3) et ((b < c) ou (7 > d)) est une expression de type
  entier, entier (ou réel), entier (ou réel), entier → booléen

Règle d'évaluation **Voir document "Type et valeur d'une expression"**

- Substitution aux noms des valeurs du contexte.
- Evaluation des opérandes dans un ordre quelconque.
- Application des opérateurs aux valeurs des opérandes

c) expression algébrique

Liaison des noms



Évaluation de max3V(4, 12, 5)

1. Appel de max3V
   x ↔ 4
   y ↔ 12
   z ↔ 5

2. Appel de max2V
   x ↔ y ↔ 12
   y ↔ z ↔ 5
   Résultat : 12

3. Appel de max2V
   x ↔ x ↔ 4
   y ↔ max2V(12, 5) ↔ 12
   Résultat : 12

d) Calcul du type d'une expression

Le type d'une expression est calculé en utilisant la spécification des opérations apparaissant dans l'expression.

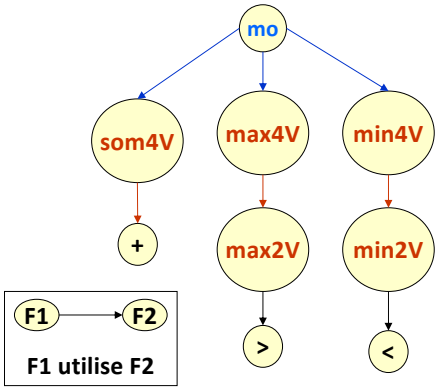
- L'expression **0 ≤ N ≤ 20** est **incorrecte.**
  Elle ne satisfait pas les règles de typage : la spécification de l'opérateur ≤ n'est pas respectée.
  entier ≤ entier ≤ entier ≡ booléen ≤ entier ≡ erreur
- L'expression **0 ≤ N et N ≤ 20** est **correcte.**
  entier ≤ entier **et** entier ≤ entier ≡ booléen **et** booléen ≡ booléen
- Exemple : calculer les types
  E ≡ **si** a **alors** z**+1 sinon 25** **a : booléen, z : entier, E : entier**

e) Fonctions

Analyse descendante

Fonctions : "briques" de base

- Prédéfinies dans le langage
- Disponibles en bibliothèque
- Définies par le programmeur

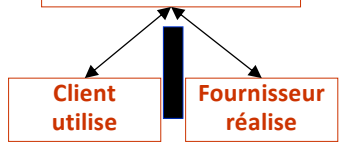


e) Fonctions
Trois points de vue

Définitions

- Spécification : quoi ?
  - Description de la relation entre donnée et résultat.
- Réalisation : comment ?
  - Algorithme exprimé sous forme d'une expression.
- Utilisation
  - Dans une expression, sans connaître la réalisation.

Spécification = Contrat



Savoir spécifier une fonction

Savoir utiliser une spécification sans connaître la réalisation

e) Fonctions
Spécification

Définitions

- Profil (signature) de la fonction

Nom de la fonction **: fonction (**Noms et types des paramètres**)** → Type du résultat

- Signification de la fonction
  - Commentaire précisant la signification
  - Pré-conditions : contraintes sur les paramètres
- La valeur résultat n'est garantie que si les pré-conditions sont vérifiées

Appartient **: fonction (**X, I, J **: entier)** → **booléen**

{ vrai ⇔ X ∈ [I…J]. Pré-condition : I ≤ J }

f) La construction soit … dans

Rôle

- Lier un nom et une valeur
- Fixer la portée de cette définition
  - Partie du texte où la définition est valide

Liaisons nom-valeur utilisables dans F (portée)

Expressions de type T

E ≡

soit nom1 **=** E1**,** nom2 **=** E2**,** . . .

dans F

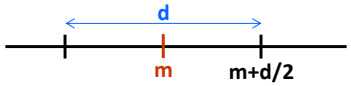
Règle d'évaluation

1. Évaluation des Ei
2. Substitution dans F
3. Évaluation de F

Maximum de deux valeurs
Expression "soit…dans…"

Le maximum de x et y se déduit

- du point milieu m
- de la distance d



max2V**(**x**,** y**) :**

retour:

Expression soit … dans → **soit** m **=** **(**x+y**)/2,** d **=** **abs(**x-y**)**
dans m **+** d**/2**

1.3. Composition conditionnelle

Fonctions, expressions, valeurs

1.0. Exemples introductifs

1.1. Types et valeurs

1.2. Composition de fonctions

1.3. Composition conditionnelle

1.4. Types construits

a) La construction SELON

Syntaxe, règles de types, évaluation

Expression de type T

Expressions de type booléen

E ≡

Selon noms caractérisant le domaine

EB1 : E1

EB2 : E2

EB3 : E3

Expressions de type T

Règle d'évaluation

1. Évaluation des EBi
   - Ordre quelconque : une et une seule EBi a la valeur vrai
2. Évaluation de la sous-expression

Autres formes : voir Polycopié

si …**alors**…**sinon**… **selon** … **autrement** …

b) Règles de partition

E ≡

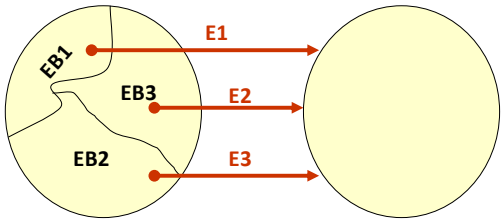
Selon

EB1 : E1

EB2 : E2

EB3 : E3

Partition du domaine



- Couverture du domaine

  Ne pas oublier de cas : quelles que soient les données

  EB1 ou EB2 ou EB3 a la valeur vrai

- Exclusion mutuelle des cas

  Pour les mêmes valeurs données, une seule valeur résultat :

  EB1 et EB2 a la valeur faux

  et EB1 et EB3 a la valeur faux

  et EB2 et EB3 a la valeur faux

c) Les opérateurs "et puis", "ou alors"

Définition

A **et puis** B ≡

si A **alors** B **sinon faux**

Définition

A **ou alors** B ≡

si A **alors vrai sinon** B

Contexte	Expression	Valeur
i ↔ 2, k ↔ 20	i > 0 **et puis** k / i > 5	**vrai**
i ↔ 2, k ↔ 20	i > 0 **et** k / i > 5	**vrai**
i ↔ 0, k ↔ 20	i > 0 **et puis** k / i > 5	**faux**
i ↔ 0, k ↔ 20	~~i > 0 et k / i > 5~~	

1.4. Types construits

Fonctions, expressions, valeurs

1.0. Exemples introductifs

1.1. Types et valeurs

1.2. Composition de fonctions

1.3. Composition conditionnelle

1.4. Types construits

a) Types et constructeurs de types

Définitions

- Type = Ensemble de valeurs + Famille d'opérations
  - nom, ensemble, dénotation des valeurs, opérations
- Type atomique (primitif) versus type complexe (construit)
  - un type complexe est une composition de types plus simples
    une date est décrite à l'aide de trois entiers: 11 9 2007
  - un type atomique est fourni par le langage
    entier *(atomique),* **texte** *(complexe)*
- Constructeurs de types
  - Éléments du langage fournis pour définir de nouveaux types
    produit de types, tableau, séquence, arbre
- n-uplets
  - le type d'un n-uplet de valeurs est le produit des types des valeurs (au sens ensembliste)

A propos d'heure d'horloge

- Concept : repère dans le temps, dans une journée
- Représentation : durée depuis le début du jour
  - nombre d'heures depuis le début du jour, de 0 à 23
  - nombre de minutes depuis le début de l'heure, de 0 à 59
  - nombre de secondes depuis le début de la minute, de 0 à 59
- Dénotation usuelle : texte composé à l'aide de 3 entiers

13 heures 42 minutes 10 secondes

Le type HeureH

Exemples de fonctions associées

HeureH **: type**

EgalH **: fonction (**H1**,** H2 **:** HeureH**) → booléen**

{ vrai ⇔ H1 et H2 sont identiques. }

InfH **: fonction (**H1**,** H2 **:** HeureH**) → booléen**

{ vrai ⇔ H1 précède H2 (ordre chronologique) }

Plus1s **: fonction (**H **:** HeureH**) →** HeureH

{ Heure du jour à la seconde suivant H. }

b) Produit de type

Définition d'un type produit : première notation

HeureH : type < entier sur [0 … 23],
entier sur [0 … 59],
entier sur [0 … 59]
>

On donne le type des champs et un commentaire précisant la signification des champs

{<a, b, c> étant de type HeureH, **a** *est le nombre d'heures,* **b** *le nombre de minutes et* **c** *le nombre de secondes.}*

Pour la réalisation, on nomme les champs localement (avec soit … dans)

```
EgalH (H1, H2) :
  retour:
    soit <h1, m1, s1> = H1, <h2, m2, s2> = H2
    dans    h1 = h2 et m1 = m2 et s1 = s2
```


b) Produit de type

Définition d'un type produit : deuxième notation

HeureH : type < nbh : entier sur [0 … 23],
nbm : entier sur [0 … 59],
nbs : entier sur [0 … 59]
>

On nomme les champs globalement au niveau du type

{H étant de type HeureH, **H.nbh** *dénote le nombre d'heures,* **H.nbm** *le nombre de minutes et* **H.nbs** *le nombre de secondes.}*

```
EgalH (H1, H2) :
  retour:
        H1.nbh = H2.nbh
    et  H1.nbm = H2.nbm
    et  H1.nbs = H2.nbs
```


Conclusion du chapitre 1

Éléments pour guider le travail hors séance

- Connaître les définitions

  Comprendre les notions sous-jacentes, savoir les utiliser
  - Type, fonction, spécification, réalisation.
  - Expressions, types de base

- S'approprier la notation algorithmique

- Savoir
  - Pratiquer une analyse descendante, une analyse par cas
  - Calculer le type d'une expression
  - Décrire un produit de types et utiliser des n-uplets

Vocabulaire

Savoir donner une définition, un exemple

- expression arithmétique, algébrique ; priorité d'opérateurs
- expression conditionnelle
- spécifier, réaliser une fonction ; paramètre formel, effectif
- profil, signature d'une fonction ; pré-condition
- type, atomique/complexe, primitif/construit
- constructeur de type
- produit de types, n-uplets
- calcul de type
- analyse descendante, décomposer un problème
- analyse par cas, partition

2. Raisonner sur les actions

2.0. Exemples introductifs

Langage des actions

sur la base d'exemples simples

Spécifications, assertions, invariants

Analyse quantitative

2.0. Exemples introductifs

2.1. Actions et notions associées

2.2. Compositions séquentielle et conditionnelle

2.3. Composition itérative – Invariant d'itération

⚠ Connaître les définitions
S'approprier le langage des actions
Savoir vérifier un algorithme à l'aide d'assertions

Échange de deux valeurs

Spécification d'une action

Décrire une action d'échange des valeurs de deux variables

Spécification

Echanger **: action (donnée-résultat** X, Y **: entier)**

{Echange les valeurs associées aux paramètres.

Exemple : si à l'état initial X = 35 et Y = 10, alors à l'état final X = 10 et Y = 35.}

Définitions

Spécifier une action : la nommer, choisir ses paramètres, les typer et décrire son effet sur l'état du système

La clause donnée-résultat indique que l'action modifie l'état des variables associées aux paramètres en fonction de leurs valeurs initiales.

***État initial (e.i.) d'une action* : état au début de l'exécution de l'action**

***État final (e.f .) d'une action* : état à la fin de l'exécution de l'action**

Échange de deux valeurs

Structure d'un programme

Lexique

A, B, C, D **: entier** ← **Variables globales**

Spécification de l'action

Echanger **: action (donnée-résultat** X,Y**: entier)**

{échange les valeurs des paramètres. e.i. :...,e.f. :... }

Algorithme

Utilisation de l'action

Lire(A) **; Lire**(B) **; Lire**(C) **; Lire**(D)

Echanger(A, C) ; Echanger(B,D) ← **paramètres effectifs**

Ecrire(A) **; Ecrire**(B) **; Ecrire**(C) **; Ecrire**(D)

Réalisation de l'action

Echanger(X, Y) : ← **paramètres formels**

Z **: entier** ← **variable locale**

Z ← X ; X ← Y ; Y ← Z ← **algorithme**

⚠ Maîtriser la notion de paramètre

Échange de deux valeurs

Trace de l'exécution

- Contexte : A a la valeur 3 et B a la valeur 4
- Trace de l'exécution de Echanger(A, B)

Dans la réalisation de Echanger, on remplace X par A, Y par B :

{A = 3, B = 4}

Z ← A

{A = 3, B = 4, Z = 3}

A ← B

{A = 4, B = 4, Z = 3}

B ← Z

{A = 4, B = 3, Z = 3}

états intermédiaires des variables

Échange de deux valeurs

Composition séquentielle, raisonnement sur les états

Description des états intermédiaires à l'aide d'assertions, en faisant abstraction des valeurs des variables

Echanger(X, Y) :

Noms de variables

Z : entier

{posons $X = x_0$, $Y = y_0$}

Z ← X

{$X = x_0$, $Y = y_0$, $Z = x_0$}

X ← Y

{$X = y_0$, $Y = y_0$, $Z = x_0$}

Y ← Z

{$X = y_0$, $Y = x_0$, $Z = x_0$}

Noms de valeurs associées aux variables (abstraction)

Le calcul des états intermédiaires repose sur la sémantique de l'instruction d'affectation

Mystère

Composition conditionnelle, raisonnement sur les états

Mystère : action (donnée-résultat U, V : entier)

Mystère(U, V) : si U > V alors Echanger(U, V)

(1) Calcul des états : en utilisant la spécification de l'action Echanger

Mystère(U, V) :

{posons $U = u_0$, $V = v_0$}

si U > V alors

Echanger(U, V)

{$u_0 > v_0$ et $U = v_0$ et $V = u_0$}

sinon {rien}

{$u_0 \leq v_0$ et $U = u_0$ et $V = v_0$}

{posons $U = u_1$, $V = v_1$: $u_1 \leq v_1$ et $<u_1, v_1>$ est une permutation de $<u_0, v_0>$}

La post condition de chaque cas

Implique

la post condition de la conditionnelle

(2) Interprétation : à l'état final, U ≤ V, quelles que soient les valeurs de U et de V à l'état initial ➔ Classement des deux valeurs

Classement de trois valeurs

Spécifications : pré-conditions et post-conditions

Décrire une action qui permute les valeurs de trois variables, de telle sorte qu'elles soient en ordre croissant

Classer3V : action (donnée-résultat X,Y,Z : entier)

Pré-condition → {e.i. posons $X = x_0$, $Y = y_0$, $Z = z_0$: x_0, y_0, z_0 distincts 2 à 2

Post-condition → e.f. posons $X = x_1$, $Y = y_1$, $Z = z_1$: $x_1 < y_1 < z_1$ et ???

$<x_1, y_1, z_1>$ est une permutation de $<x_0, y_0, z_0>$}

Définitions

Pré-condition **: hypothèse sur l'état initial**

Post-condition **: assertion sur l'état final**

La post-condition d'une action n'est garantie que si à l'état initial la pré-condition est vérifiée

Classement de trois valeurs

Composition conditionnelle : raisonnement sur les états

Voir polycopié E2.3

Idée : Placer la plus petite valeur, puis classer les autres valeurs

Classer3V(X, Y, Z) : {valeurs distinctes deux à deux}

{e.i. $X = x_0$, $Y = y_0$, $Z = z_0$: x_0, y_0, z_0 distincts 2 à 2}

{Placer la plus petite valeur dans X}

selon X, Y, Z

X < Y et X < Z : {rien}

Y < X et Y < Z : Echanger(X, Y) { $X = y_0$, $Y = x_0$, $Z = z_0$ et $y_0 < x_0$ et $y_0 < z_0$ }

Z < X et Z < Y : Echanger(X, Z)

{$X = x_1$, $Y = y_1$, $Z = z_1$: $x_1 < y_1$ et $x_1 < z_1$ et EstPerm($<x_1, y_1, z_1>$,$<x_0, y_0, z_0>$)}

compléter

et valider

{Classer les deux valeurs restantes}

si Y > Z alors Echanger(Y, Z)

{e.f. $X = x_2$, $Y = y_2$, $Z = z_2$: $x_2 < y_2 < z_2$ et EstPerm($<x_2, y_2, z_2>$,$<x_0, y_0, z_0>$)}

Lire l'exemple E2.3. et travailler sur le calcul des états

Combinaisons de 2 entiers parmi n

Spécification d'une action

Afficher les combinaisons de deux entiers pris parmi les entiers de l'intervalle [1…n]

- Un exemple
  - [1, 2, 3, 4] → [<1,2>, <1,3>, <1,4>, <2,3>, <2,4>, <3,4>]
- Spécification d'une action

  AfficherComb **: action (donnée** n **: entier > 0)**

  { Affiche les couples d'entiers pris dans [1… n] }
- Principe de solution
  - Dans chaque couple, placer le plus petit en première position
  - Pour chaque entier i de [1…n-1] énumérer les couples comportant i en première position.

    - Pour cela, énumérer les entiers strictement supérieurs à i.

Combinaisons de 2 entiers parmi n

Réalisation itérative, analyse quantitative

```
AfficherComb(n) :
    pour i allant de 1 à n - 1
        pour j allant de i + 1 à n
            Écrire(i, j)
```

Nombre d'appels de l'instruction Écrire pour n donné ?

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n*(n-1)/2$$

$$\sum_{j=i+1}^{n} 1 = n - (i+1) + 1 = n - i$$

$$1$$

Validation

Nombre de combinaisons de deux éléments pris parmi n

Poids d'un entier

Spécification d'une fonction

Le "poids" d'un entier est la somme des chiffres de sa représentation décimale.

- **3672 → 18**
- **0 → 0**

Spécification

Poids **: fonction (**E **: entier ≥ 0) → entier ≥ 0**

Profil (signature) de la fonction

Signification de la fonction → *{Somme des chiffres de la représentation décimale de E, par exemple, Poids(12556) = 19.}*

Définitions

Spécifier une fonction : la nommer, typer ses paramètres et ses résultats, et décrire la relation entre données et résultats

L'exécution d'une fonction ne modifie pas l'état des variables du contexte d'appel.

Réaliser une fonction : sous *forme fonctionnelle* à l'aide d'une *expression* ou sous *forme actionnelle* en composant des actions plus élémentaires.

L'indication de la valeur du résultat ("renvoi du résultat") est toujours placée à *la fin physique du texte de la réalisation, sous forme d'une expression* .

Poids d'un entier

Réalisation d'une fonction sous forme actionnelle, solution itérative

Principe

Sommer les chiffres. Pour les extraire de l'entier donné, procéder par divisions successives par 10.

```
Poids(n) :
    S : entier ≥ 0                          ← Variables locales
    Q : entier ≥ 0
    S ← 0 ; Q ← n                           ← Initialisation des variables
    tant que Q ≥ 10                         ← Condition de continuation
        {Poids(n) = Poids(Q) + S}           ← Invariant d'itération
        S ← S + Q reste 10                  ← Corps de l'itération
        Q ← Q quotient 10
    {Poids(n) = Poids(Q) + S et Q < 10}
    retour : S + Q                          ← Valeur renvoyée
```

Erreur ?

2.1. Actions et notions associées

Raisonner sur les actions

2.0. Exemples introductifs

2.1. Actions et notions associées

2.2. Compositions séquentielle et conditionnelle

2.3. Composition itérative – Invariant d'itération

a) Concepts, vocabulaire

A voir ou à revoir

Notions à maîtriser – cf polycopié

- Variables, Constantes, Valeurs
- Types
- Algorithme, Action, Fonction, Programme
- Lexique, Portée d'un nom, Liaison d'un nom
- Paramètres, passage de paramètre

Définitions

Une assertion énonce une propriété de l'état d'une variable ou d'un ensemble de variables à un instant de l'exécution.

Les assertions sont utilisées pour annoter les programmes par des pré-conditions et des post-conditions.

Une assertion est décrite en termes de noms de variables

b) Composition d'actions

Actions élémentaires

- Affectation
  - Syntaxe Nom_De_Variable ← expression
  - X ← 3 se lit "X prend pour valeur 3"
  - Règles
    - *Le nom doit apparaître dans le lexique*
    - *Le type de l'expression doit être celui de la variable*

- Entrées/Sorties
  - Lecture d'une entrée : attente de la saisie d'une valeur
    - *Syntaxe* **Lire(**Nom_De_Variable**)**
    - *Règle : la valeur lue doit être du type de la variable*
    - *Effet de Lire(X) ?*
  - Écriture vers une sortie : affichage d'une valeur
    - *Syntaxe* **Écrire(**expression**)**

b) Composition d'actions

à voir ou à revoir

- Composition séquentielle
  - A **;** B
- Composition conditionnelle
  - **selon** … **si** … **alors** … **sinon** … **si** … **alors** …
- Composition itérative
  - **tant que** … **répéter** … **jusqu'à** **pour** i **allant de** … **à** … **répéter** n **fois** …

voir polycopié

- Composition récursive
  - A**(**P**) :** … A**(**P1**)** …

chapitre 8

c) statut d'une variable / une action

Définitions

Définitions

- statut **Donnée**
  - l'action consulte la valeur de la variable
  - la valeur de la variable n'est pas modifiée par l'action
- statut **Résultat**
  - l'action construit une nouvelle valeur de cette variable
  - l'effet de l'action ne dépend pas de la valeur initiale
- statut **Donnée-résultat**
  - l'action peut modifier la valeur de la variable
  - l'effet de l'action peut dépendre de la valeur initiale de la variable.

Passage de paramètres

Paramètre	Statut		
Formel	**Donnée**	**Résultat**	**Donnée-Résultat**
Effectif	Expression	Variable	Variable

c) statut d'une variable / une action

Exemples

- X ← 3 — X est de statut *résultat*
- X ← Y + 4 — X est de statut *résultat*; Y est de statut *donnée*
- X ← X +1 — X est de statut *donnée-résultat*
- Lire(X) — ? la variable X est de statut *résultat*; la valeur lue est une donnée (entrée)
- Écrire(Y) — ? la variable Y est de statut *donnée*; la valeur écrite est un résultat (sortie)

d) Paramétrer

A propos de sommation

Paramétrer l'extrait d'algorithme suivant : *U ← U + 1*

Plus1 : **fonction** (X : **entier**) → **entier** *{Plus1(X) = X+1}*

U ← Plus1(U)

Incrémenter : **action** (***donnée-résultat*** *X*:**entier**)

{ e.i. X= x_0 ; e.f. X=x_0+1}

Incrémenter(U)

Ajouter : **action** (***donnée-résultat*** *X*:**entier** ;***donnée*** *A*: **entier**)

{ e.i. X= x_0, A=a_0 ; e.f. X=x_0+a_0 , A=a_0 }

Ajouter(U, 1)

SommerDans : **action** (***résultat*** *X*: **entier** ;
donnée *Y,Z* : **entier**)

{ e.i. Y= y_0, Z=z_0 ; e.f. X=y_0+z_0 , Y=y_0 , Z=z_0 }

SommerDans(U, U, 1)

Affecter : **action** (***résultat*** *X*: **entier** ;***donnée*** *Y*:**entier**)

{ e.i. Y= y_0 ; e.f. X=y_0 , Y=y_0 }

Affecter(U, U+1)
ou Affecter(U, Plus1(U))

e) Vérifier une réalisation

Somme : **fonction** (X, Y : **entier**) → **entier**
{Somme(X, Y) = X+Y}
retour : ~~**Écrire(**~~X+Y**)**

Où est l'erreur ?

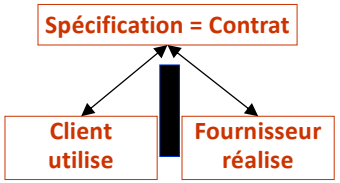
Accumuler : **action** (**donnée** X : **entier, donnée-résultat** Y : **entier**)
{ e.i. posons X = x0, Y = y0 ; e.f. Y = x0 + y0, , X=x_0 }
Accumuler(X, Y) :
~~**Lire(**X**) ; Lire(**Y**) ;**~~ Y ← X+Y
A, B, C : **entier**
Lire(A, B, C) ; Accumuler(A+B, C) ~~; Accumuler(A, B+C)~~

Où sont les erreurs ?

f) Actions et fonctions

Triple point de vue

- Spécification : quoi ?
  - **action** : description de l'effet par des assertions caractérisant les états initial et final.
  - **fonction** : description de la relation entre valeurs données et valeurs renvoyées.
- Réalisation : comment ?
  - **action** : algorithme exprimé en termes de variables, actions, fonctions, en utilisant les outils de composition.
  - **fonction** : en utilisant le langage des expressions (chapitre 1, 6, 7) ou sous forme "actionnelle" (voir Polycopié)
- Utilisation
  - En respectant la spécification, sans connaître la réalisation.





Savoir spécifier le statut des paramètres d'une action

Savoir utiliser une spécification sans connaître la réalisation

f) Actions et fonctions

Transformation d'une fonction en action : spécifications

Max2V : **fonction** (X, Y : **entier**) → **entier**
{maximum des deux valeurs données.}
Max3V : **fonction** (X, Y, Z : **entier**) → **entier**
{maximum des trois valeurs données.}

CréerM2V : **action (donnée** X,Y : **entier ; résultat** M : **entier)**
{ à l'état final, M = Max2V(X, Y) }

CréerM3V: **action (donnée** X,Y,Z : **entier ;**
résultat M : **entier)**
{ à l'état final, M = Max3V(X, Y, Z) }

f) Actions et fonctions

Transformation d'une fonction en action : réalisations, utilisations

```
{réalisation des deux fonctions}
Max2V(X, Y) :
  retour:
     si X > Y alors X sinon Y
Max3V(X, Y, Z) :
  retour:
     Max2V(X, Max2V(Y, Z))

{Exemple d'utilisation}
A, B, C : entier
Lire(A) ; Lire(B) ; Lire(C)
Écrire(Max3V(A, B, C))
```

```
{réalisation des deux actions}
CréerM2V(X, Y, M) :
     M ← si X > Y alors X sinon Y
CréerM3V(X, Y, Z, M) :
     W : entier
     CréerM2V(Y, Z, W)
     CréerM2V(X, W, M)

{Exemple d'utilisation}
A, B, C : entier
D : entier
Lire(A) ; Lire(B) ; Lire(C)
CréerM3V(A, B, C, D)
Écrire(D)
```


2.2. Compositions séquentielle et conditionnelle

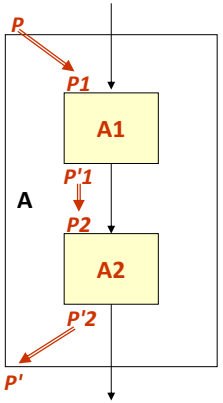
Raisonner sur les actions

2.0. Exemples introductifs
2.1. Actions et notions associées
2.2. Compositions séquentielle et conditionnelle
2.3. Composition itérative – notion d'invariant

a) Composition séquentielle

Grille d'étude d'une instruction

- Syntaxe : A1 ; A2
- Évaluation : exécuter A1 puis A2
- Analyse quantitative
  - toute exécution de A entraîne une exécution de A1 et une exécution de A2
- Cohérence : compatibilité entre
  - pré-condition *P* de A et pré-condition *P1* de A1
  - post-condition *P'1* de A1 et pré-condition *P2* de A2
  - post-condition *P'2* de A2 et post-condition *P'* de A



rappel : P ⇒ Q est équivalent à non P ou Q
en particulier, si P est faux, P ⇒ Q est vrai

si P ⇒ Q est vrai, par exemple A > 3 ⇒ A > 0
Q est une propriété plus faible (moins contraignante) que P

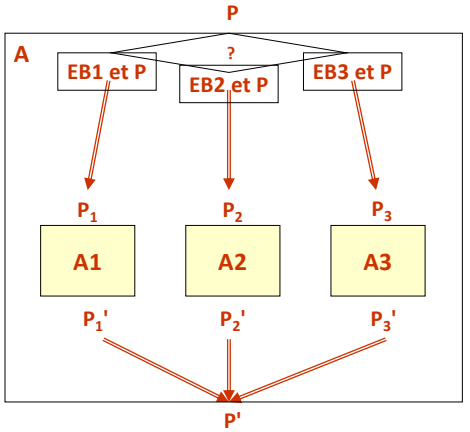
b) Composition conditionnelle

Cohérence entre les états

caractérisent le domaine

A ≡
Selon noms
EB1 : A1
EB2 : A2
EB3 : A3

Expressions de type booléen



Voir polycopié
Règle de partition, mode d'évaluation
Autres formes de conditionnelles

2.3. Composition itérative – Invariant

Raisonner sur les actions

2.0. Exemples introductifs
2.1. Actions et notions associées
2.2. Compositions séquentielle et conditionnelle
2.3. Composition itérative – Invariant d'itération

a) La construction TANT QUE

Expression de type booléen

```
A0
{P1}
tant que CC
      {P2}
      A1
      {P3}
{P4}
A2
```

Les quatre points d'observation d'une itération

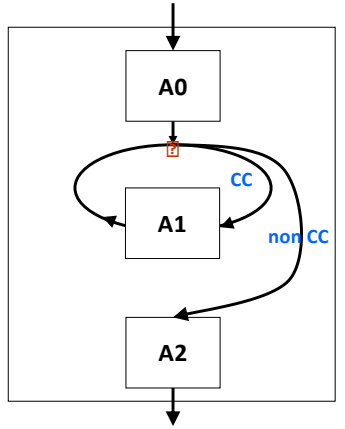


Schéma d'exécution

Autres formes : voir polycopié
répéter n fois pour i allant de … à … répéter … jusqu'à

b) Analyse quantitative

Principes

Nombre d'appels de l'opérateur + ?

```
x ← 0
pour i allant de 1 à n
    x ← x + 1 ; x ← x + 1
```

$$\sum_{i=1}^{n} 2 = 2n$$

```
x ← 0
pour i allant de 1 à n
    x ← x + 2
pour j allant de 1 à m
    x ← x + 3
```

$$\sum_{i=1}^{n} 1 + \sum_{j=1}^{m} 1 = n+m$$

Mode de calcul

- **Nombre d'appels d'une opération : somme des appels du corps de l'itération.**
- **Composition séquentielle d'itérations : somme des résultats obtenus pour chaque itération.**

b) Analyse quantitative

Itérations emboîtées, principe

Nombre d'appels de l'opérateur + ?

```
x ← 0
pour i allant de 1 à n
    pour j allant de 1 à i
        x ← x + 2
```

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 \qquad \sum_{j=1}^{i} 1$$

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = n(n+1)/2$$

Procéder de l'itération la plus interne vers l'itération la plus externe

b) Analyse quantitative

Itérations emboîtées

Nombre d'appels de l'opérateur + ?

$c = \sum_{i=1}^{n} ?$

```
x ← 0
pour i allant de 1 à n
    pour j allant de 1 à i
        x ← x + 2
    x ← x + 1
    pour k allant de 1 à n
        x ← x + 1 ; x ← x + 1
        pour j allant de 1 à m
            x ← x + 2
```

$\sum_{j=1}^{i} 1 = i$

1

1

$\sum_{k=1}^{n} (2 + m) = n(2+m)$

2

$\sum_{j=1}^{m} 1 = m$

1

$$c = \sum_{i=1}^{n} (i + 1 + n(2+m)) = n(n+1)/2 + n + n^2(2+m)$$

c) Invariant d'itération

- Invariant d'itération
  - Caractérise l'action A du corps de l'itération.
  - Propriété commune de sa pré-condition (en P2) et de sa post-condition (en P3).

Définition

INV est un invariant de l'itération si et seulement si

CC et INV vraie en P2 ⇒ INV vraie en P3

```
{P1  : INV}
tant que CC
    {P2  : INV}
    A
    {P3  : INV}
{P4  : non CC et INV}
```

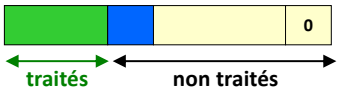
- Utilisation : établir une propriété de l'itération
  - si **INV** est un invariant de l'itération
  - et si **INV** est vraie **en P1**
  - on peut conclure que si l'itération se termine **non CC et INV** est vraie **en P4**

Somme d'entiers

Itération, invariant

Réaliser un algorithme qui affiche la somme d'une suite d'entiers > 0 marquée par 0 et traitée à la volée.

{principe : parcours de la suite}




```
S : entier ≥ 0                          {pour accumuler}
EC : entier ≥ 0                         {entier courant}
Marque : constante 0 de type entier
S ← 0 ; Lire(EC)
tant que  EC ≠ Marque
        {S est la somme de tous les entiers précédant EC}
        S ← S + EC
        Lire(EC)
Écrire (S)
```

Invariant(s) de l'itération ?

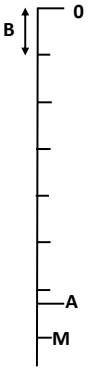
A propos de multiples

Itération, invariant

Décrire une fonction qui détermine si A est multiple de B.

{principe : recherche dans la suite des multiples}

Ex : A n'est pas multiple de B




```
EstMult : fonction (A : entier ≥ 0, B : entier > 0) → booléen
EstMult(A, B) :
    M : entier ≥ 0                      {Multiple courant}
    M ← 0
    tant que M < A
        {M est un multiple de B}
        M ← M + B
    {M ≥ A et M est un multiple de B}
    retour : A=M                        {A est un multiple de B}
```

Invariant(s)?

Preuve de correction ?

d) vérifier une itération

- S'assurer de la terminaison
  - Identifier une quantité entière dépendant des variables et montrer qu'elle est toujours ≥ 0 et qu'elle décroît à chaque étape.
- S'assurer de la correction partielle

  "partielle" = sous l'hypothèse que l'itération se termine.

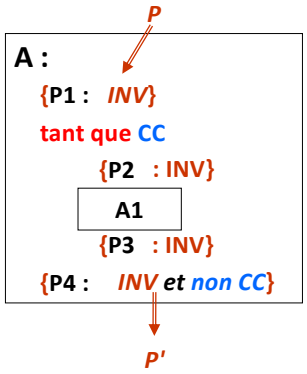
  Pour cela trouver une assertion INV telle que

  INV est un invariant de l'itération

  et INV est vraie en P1

  et $P \Rightarrow INV$

  et INV *et* $\text{non } CC \Rightarrow P'$





L'évaluation de CC
ne doit pas modifier les variables.
L'initialisation des variables
DOIT se trouver juste avant l'itération

Mystère

Invariants d'itération

```
n : entier > 0                          {donnée}
S : entier  > 0
i : entier  sur [1…n]
j : entier > 0
Lire (n) ; i ← 1 ; j ← 3 ; S ← 1
P1  tant que i ≠ n
P2      {j=2*i+1 et S = i² }
        S ← S + j ; i ← i + 1 ; j ← j + 2
    {S = n² }
P4  Écrire(S)
```

Assertion	Invariant ?	en P1 ?
i est impair	non	oui
j est pair	oui	non
j est impair	oui	oui
i ∈ [1…n]	oui	oui
S > 0 et j > 0	oui	oui
S = i²	non	oui
j = 2*i + 1 et S = i²	oui	oui

Conclusions

- i ∈ [1…n], S>0 et j>0 sont des invariants, vrais en P1
  - **on annote le lexique en conséquence**
- **n-i ≥ 0** et **n-i** décroît à chaque étape : l'itération se termine
- j=2*i+1 et S = i² est un invariant, vrai en P1
  - donc en P4, i = n et j=2*i+1 et S = i² ⇒ en P4, S = n²

d) Vérifier une itération

Preuve d'invariance

- Montrer que INV est un invariant
  - Hypothèse : l'assertion CC et INV est satisfaite en P2
  - Sous cette hypothèse montrer que INV est satisfaite en P3
- Exemple : Dans l'algorithme précédent

  Montrer que $i \in [1...n]$ est un invariant de l'itération

  CC : $i \neq n$ A : $S \leftarrow S + j$; $i \leftarrow i + 1$; $j \leftarrow j + 2$
- Démonstration

  On part de l'état initial $i \neq n$ et $i \in [1...n]$ ce qui implique $i \in [1...n-1]$

  $\{i \in [1...n-1]\}$ $S \leftarrow S + j$ $\{i \in [1...n-1]\}$; $i \leftarrow i + 1$ $\{i \in [2...n]\}$

  $j \leftarrow j + 2$ $\{i \in [2...n]\}$

  Pour conclure : $i \in [2...n]$ ce qui implique $i \in [1...n]$ CQFD



Dans ce calcul on n'observe que le texte de l'action A

```
{P1}
tant que CC
      {P2}
      A
      {P3}
{P4}
```


Conclusion du chapitre 2

Éléments pour guider le travail hors séance

- Connaître les définitions

  Comprendre les notions sous-jacentes, savoir les utiliser
  - état, assertion, spécification, pré-condition, post-condition.
  - statut des paramètres : "donnée", "résultat", "donnée-résultat"
  - portée des variables
  - invariant d'itération
- S'approprier la notation algorithmique
- Savoir
  - Utiliser les outils de composition d'actions
  - Dénombrer les opérations engendrées par l'exécution d'un algorithme
  - Vérifier une réalisation : elle respecte
    - *le type et le statut des paramètres*
    - *la post-condition sous l'hypothèse de la pré-condition*
  - Démontrer qu'une assertion est un invariant d'une itération
  - Utiliser un invariant pour déterminer une propriété à l'issue d'une itération

Vocabulaire

Savoir donner une définition, un exemple

- spécifier, réaliser une action, une fonction
- pré-condition, post-condition, assertion, invariant d'itération
- variable locale, globale
- paramètre formel, effectif – passage de paramètre
- statuts des paramètres : donnée, résultat, donnée-résultat

3. Séquences et tableaux

3.0. Exemples introductifs

Raisonnement abstrait sur les séquences
Représentation des séquences à l'aide de tableaux
Schémas itératifs de traitement de séquences
Tableaux

3.0. Exemples introductifs

3.1. Tableaux

3.2. Séquences

3.3. Ensembles, relations, piles, files



Maîtriser la notion de tableau
Connaître les schémas itératifs et savoir les utiliser
Connaître les exemples, savoir les réutiliser (analogie)
Savoir exprimer un principe en termes de séquences

Nombre d'occurrences d'une valeur

Etapes de résolution d'un problème : de l'abstrait au concret

Déterminer le nombre d'occurrences d'un entier E dans une séquence S.

Étapes de résolution

Énoncé, spécification, principe, réalisation, correction, analyse quantitative

Spécification : questions
Action ou fonction ? → fonction (il n'y a pas lieu de modifier les données)
Paramètres ? → un entier, une séquence (lecture de l'énoncé)
Types mis en jeu ? → un type nommé (pour faire abstraction de la représentation de la séquence)

NbOc **: fonction (**E **: entier,** …… **) → entier ≥ 0**
{ Nombre d'occurrences de E dans la séquence }

précision sur le domaine de variation

Caractérisation de la séquence

Nombre d'occurrences d'une valeur

Analyse : reconnaissance d'un problème de parcours

On raisonne en faisant abstraction de la représentation de la séquence

Principe de l'algorithme
Pour chaque élément de S égal à E, incrémenter un compteur C.
(instance d'un problème de "parcours")

Définition

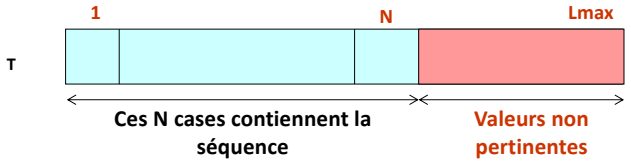
∀z∈S tel que z=E, ….

***Parcours** d'une séquence*

Application d'une même action, une fois et une seule, à chacun des éléments de la séquence dans l'ordre de la séquence.

Nombre d'occurrences d'une valeur

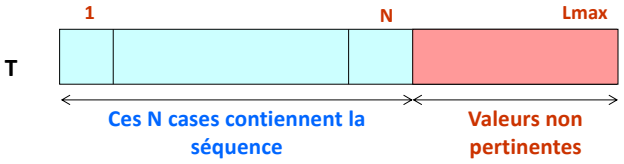
Représentation contiguë d'une séquence dans un tableau



- Contigüe: deux **éléments consécutifs** de la séquence se trouvent dans deux cases **d'indices consécutifs**.
- Longueur explicite : une variable N pour la **longueur réelle** de la séquence (différent de la taille du tableau qui est la longueur max de la séquence Lmax)
- Non contigüe : si deux éléments consécutifs de la séquence étaient dans deux cases d'indices non consécutifs
- Non explicite : autre représentation possible sans la longueur mais avec une marque de fin

Nombre d'occurrences d'une valeur

Représentation contiguë d'une séquence dans un tableau



Trois entités à définir dans un lexique

Un tableau T, sa dim. max Lmax, un nombre d'elt N (longueur expl.)

Exemple : définition d'une "séquence d'entiers"

Lmax : **constante de type entier > 0** *{longueur maximum}*

T : **tableau sur [1…Lmax] d'entier**

N : **entier sur [0…Lmax]** *{N=0 ⇔ séquence vide}*

NbOc : **fonction (**E : **entier,** T : **tableau sur [1…Lmax] d'entier,**
N : **entier sur [0…Lmax]) → entier ≥ 0**

Erreur classique

Ne pas confondre N et Lmax

N est une variable, Lmax une constante

Nombre d'occurrences d'une valeur

Réalisation : application du schéma de parcours

NbOc(E, T, N) :

C : **entier ≥ 0** *{résultat : compteur du nombre d'occurrences}*

C ← 0

pour p **allant de** 1 **à** N

{Invariant : C est le nombre d'éléments de $T_{[1…p-1]}$ de valeur E }

si T_p = E alors C ← C + 1

retour : C

Dénote une tranche de tableau

Analyse quantitative

Nombre d'accès au tableau T : N

Nombre d'additions : Valeur finale de C

Savoir résoudre un problème par étapes successives
Savoir identifier un problème de parcours
Savoir utiliser le schéma de parcours

Le premier entier positif

Problème de recherche dans une séquence

Déterminer la valeur du premier entier positif ou nul d'une séquence d'entiers

Spécification

TabEnt : **type tableau sur [1..Lmax] d'entier**

Ent : **type entier sur [0..Lmax]**

PremPos : **fonction (**T : TabEnt, N : Ent**) → booléen, entier ≥ 0**

{Posons <B, E> = PremPos(T, N) : si $T_{[1…N]}$ contient un entier ≥ 0, B a la valeur vrai et E est le premier entier ≥ 0 de $T_{[1…N]}$; sinon B a la valeur faux et la valeur de E n'est pas pertinente.}

Principe de l'algorithme

Énumérer les éléments de la séquence jusqu'au premier entier positif, s'il existe. Arrêter dès que l'on trouve ou que l'on atteint la fin.

Définition $\exists z \in T_{[1…N]}$ tel que P(z)

***Recherche** dans une séquence*

Identification du premier élément d'une séquence vérifiant une propriété donnée (dans l'ordre de la séquence), s'il existe.

Le premier entier positif

Réalisation : application du schéma de recherche

PremPos(T, N) :

i : **entier**

i ← 1

tant que i ≠ N+1 **et puis non** ($T_i \geq 0$)

{Invariant : $\forall x \in T_{[1…i-1]}$, $x < 0$}

i ← i + 1

{ (i=N+1 ou alors $T_i \geq 0$) et ($\forall x \in T_{[1…i-1]}$, $x < 0$) }

retour : **si** i = N +1 **alors** <faux, 0> *{non trouvé}*

sinon <vrai, T_i > *{trouvé en i}*

$T_i < 0$

Les formulations suivantes sont incorrectes

tant que $T_i < 0$ …

tant que i ≠ N +1 et $T_i < 0$ …

tant que $T_i < 0$ et i ≠ N +1 …

tant que $T_i < 0$ et puis i ≠ N +1 …

Appartenance

Existence d'un élément vérifiant une propriété

Décrire une fonction d'appartenance d'une valeur à une séquence

Appartient : **fonction** (E : **entier**, T : TabEnt, N : Ent) → **booléen**

{ vrai ⇔ $E \in T_{[1...N]}$ }

Schéma de recherche

Appartient(E, T, N)

i : **entier sur** [1…N + 1]

i ← 1

tant que i ≠ N + 1 **et puis** $T_i \neq E$

i ← i + 1

retour : i ≠ N +1

si i ≠ N + 1 **alors vrai sinon faux**

arrêt dès que possible

Schéma de parcours

{$E \in T_{[1...N]} \Leftrightarrow T_1 = E$ ou … $T_i = E$ ou …}

analogie avec Somme

Appartient(E, T, N) :

Ap : **booléen**

Ap ← **faux**

pour i **allant de** 1 **à** N

{Invariant : Ap ⇔ $E \in T_{[1...i-1]}$}

Ap ← Ap **ou** T_i = E

retour : Ap

facile à composer (itérations emboîtées)

Tous positifs

Quel schéma appliquer ?

Déterminer si les entiers d'une séquence sont tous positifs

TousPos: **fonction** (T : TabEnt, N : Ent) → **booléen**

{ vrai ⇔ $\forall e \in T_{[1...N]}$, e ≥0 }

Idée 1: appliquer le schéma de parcours

Introduction d'une variable de nom TP :

TP : un booléen {accumulateur}

Invariant de l'itération :

TP ⇔ $\forall e \in T_{[1...i-1]}$, e ≥0

Idée 2: appliquer le schéma de recherche

Les éléments de la seq sont tous positifs si et seulement si il n'existe pas d'élément négatif

TousPos(T,N) vrai ⇔ non ($\exists e \in T_{[1...N]}$, e <0)

Appliquer le schéma avec la propriété *"être négatif "*

TousPos(S) a la valeur vrai si et seulement si *"on ne trouve pas"*

Exercice de rédaction : donner les réalisations correspondant à ces deux idées

Bleu, blanc, rouge

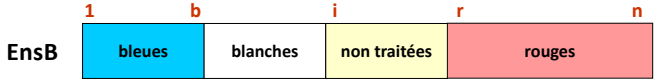
Plusieurs séquences dans un tableau

On donne, dans un tableau, un ensemble de billes de couleurs bleu, blanc et rouge. Réaliser un algorithme qui réarrange le tableau de manière à regrouper, dans l'ordre, les bleues puis les blanches puis les rouges.

Contrainte : uniquement par échanges (↔) et sans tableau supplémentaire.

Principe

Parcours du tableau en maintenant quatre zones : une séquence de billes bleues, une séquence de billes blanches, la séquence des billes non traitées, une séquence de billes rouges.



Traitement de la bille courante

La placer dans la bonne séquence selon sa couleur, par échange.

Bleu, blanc, rouge

Réalisation : exploitation de l'invariant

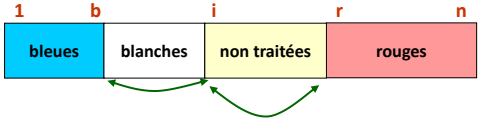
Bille : **type** [bleu, blanc, rouge] *{type énuméré}*

n : **constante de type entier ;** EnsB : **tableau sur** [1…n] **de** Bille

b : **entier sur** [0…n] *{indice de la dernière bleue}*

r : **entier sur** [1…n+1] *{indice de la première rouge}*

i : **entier sur** [1…n+1] *{pour le parcours. La dernière blanche est en i -1}*



b ← 0 ; r ← n + 1 ; i ← 1

tant que i ≠ r

{invariant : voir dessin}

selon $EnsB_i$

$EnsB_i$ = bleu : $EnsB_i \leftrightarrow EnsB_{b+1}$; b ← b + 1 ; i ← i + 1

EnsBi = blanc : i ← i + 1

EnsBi = rouge : $EnsB_i \leftrightarrow EnsB_{r-1}$; r ← r - 1

3.1. Tableaux

3. Séquences et tableaux

3.0. Exemples introductifs

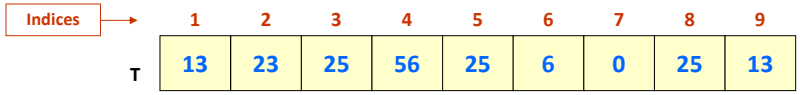
3.1. Tableaux

3.2. Séquences

3.3. Ensembles, relations, piles, files

a) Notion de tableau

Indiciation



T : tableau sur [1…9] d'entier

Définitions

- A chaque case du tableau correspond une variable
  - si la variable **i** contient l'indice de cette case, son nom est T_i (ou T[i])
  - la valeur de la variable de nom T_4 est **56**.
- La valeur du tableau de nom **T** est formée des valeurs de ses cases, dans l'ordre des indices.

a) Notion de tableau

Définitions

Spécification d'un tableau : forme générale

bi, bs **: constante de type entier** *{bornes inférieure et supérieure}*

{bi et bs sont des constantes dont la valeur est fixée au moment de coder le programme ; bi n'a pas nécessairement la valeur 0 ou la valeur 1.

Notion de tableau vide : bi > bs}.

Nom_tableau **: tableau sur [**bi…bs**] de** Type_Élément

Indiciation

Toute expression à valeur entière prise dans l'intervalle [bi…bs] peut servir d'indice



Erreur majeure : débordement de tableau

b) Opérations sur les tableaux

- Il n'y a pas d'opération globale.
- Toute opération est réalisée en termes des opérations sur les éléments du tableau.

Exemples

n **: constante de type entier > 0**

TA, TB **: tableau sur [**1…n**] d'entier**

- Mettre des zéros partout dans un tableau

  pour i **allant de** 1 **à** n **:** $TA_i \leftarrow 0$

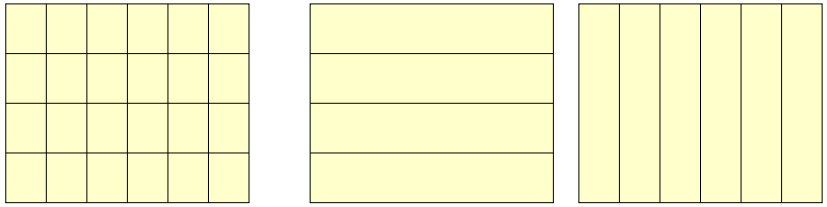
- Recopier (affectation entre tableaux)

  pour i **allant de** 1 **à** n **:** $TB_i \leftarrow TA_i$

c) Tableaux à plusieurs dimensions

Plusieurs points de vue

Plusieurs manières de considérer un tableau à deux dimensions de L lignes et C colonnes



Ensemble de cases | **Ensemble de lignes** | **Ensemble de colonnes**

Exemple de définition d'un tableau à deux dimensions

L **: constante de type entier ≥ 1** *{nombre de lignes}*

C **: constante de type entier ≥ 1** *{nombre de colonnes}*

M **: tableau sur [**1…L**] de tableau sur [**1…C**] d'entier**

{M $_{i,j}$ est le nom associé au j° élément de la i°ligne, c'est-à-dire celui se trouvant à l'intersection de la ligne i et de la colonne j.}

c) Tableaux à plusieurs dimensions

Principes d'analyse

- Composer les schémas itératifs (parcours, recherche)
  - on raisonne sur une suite de lignes (ou colonnes) et dans chaque ligne (ou colonne) sur une suite de cases.
- "Linéariser" le tableau à deux dimensions

  Se ramener à une seule dimension, en considérant une suite de couples **<**i**,** j**>** :
  - premier couple : **<**1**,** 1**>**
  - dernier couple : **<**L**,** C**>**
  - successeur de **<**i**,** j**>** : **si** j **<** C **alors <**i**,** j**+**1**> sinon <**i**+**1**,** 1**>**
  - Le couple **<**L**+**1**,** 1**>** peut servir de "marque" lors du traitement de cette suite de couples.

Appartenance

Existence dans un tableau de tableaux

*La valeur **E** appartient-t-elle à un tableau de **n** lignes et **m** colonnes ?*

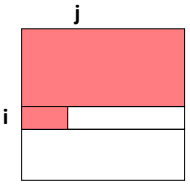
n**,** m **: constante de type entier**

Tab2D **: type tableau sur [**1…n**] de tableau sur [**1…m**] d'entier**

Ap **: fonction (**E **: entier,** M **:** Tab2D**) → booléen** *{ vrai ⇔ E ∈ M }*

Version 1 : deux parcours emboîtés

```
Ap(E, M) :
    Existe : booléen
    Existe ← faux
    pour i allant de 1 à n
        {Invariant : Existe a la valeur vrai ⇔ E ∈ M[1...i-1],[1..m]}
        pour j allant de 1 à m
            {Invariant : Existe a la valeur vrai ⇔ E ∈ M[1...i-1],[1..m] ou E ∈ M i,[1...j-1]}
            Existe ← Existe ou M i, j = E
    retour : Existe
```



Appartenance

Linéarisation d'un tableau de tableaux

Version 2 : deux recherches emboîtées

PIEGE

A faire en exercice (voir exercice E3.10 du polycopié)

Version 3 : linéarisation

Recherche sur la suite des couples <i,j>

```
Ap(E, M) :
    i : entier sur [1…n+1] ; j : entier sur [1…m]
    i ← 1 ; j ← 1                                   {premier couple}
    tant que i ≠ n+1 et puis M i, j ≠ E              {marque = <n+1,1>}
        si j < m alors j ← j + 1                     {couple suivant}
        sinon i ← i+1 ; j ← 1
    retour : i ≠ n+1
```

3.2. Séquences

3. Séquences et tableaux

3.0. Exemples introductifs

3.1. Tableaux

3.2. Séquences

3.3. Ensembles, relations, piles, files

De l'abstrait au concret :

Représentation des séquences (structure logique) à l'aide de tableaux (structure concrète)

Principes, schémas itératifs, schémas de modification

a) Faire abstraction de la représentation

Le constructeur de type "Séquence de ..."

Définitions

Notion de séquence

Collection d'éléments de même type ; l'ordre entre les éléments est significatif ; un même élément peut apparaître plusieurs fois.

Constructeurs et sélecteurs

***[ ]** (séquence vide), e ∘ S (ajout à gauche),*

*S ● e (ajout à droite), S1 **&** S2 (concaténation)*

EstVide?**, **premier**, **fin**, **début**, **dernier

Séquence triée

Le type des éléments de la séquence est muni d'une relation d'ordre (<).

La séquence est triée si et seulement, E et F étant deux éléments quelconques de la séquence, E avant F ⇔ E ≤ F

a) Faire abstraction de la représentation

Schémas itératifs : expression de l'accès séquentiel

Parcours

Initialiser le traitement

Démarrer

tant que non fini

  Traiter l'élément courant

  Avancer

Recherche du premier élément vérifiant une propriété P

Démarrer

tant que non fini **et puis non** trouvé

  Avancer

{on a trouvé ⇔ non fini}

Signification des termes

Démarrer : Initialiser l'élément courant avec le premier

Avancer : Passer à l'élément suivant

fini : on a énuméré tous les éléments

trouvé : l'élément courant satisfait P

Leur formalisation dépend de la représentation de la séquence

b) Principes de la représentation contiguë

d'une séquence dans un tableau

Définitions

- Le tableau est le contenant, de taille constante fixée a priori.
- La séquence est le contenu, de taille variable.
- **Contiguïté** : deux éléments consécutifs de la séquence se trouvent dans deux cases d'indices consécutifs.
- Premier élément : il est placé en première position du tableau.
- Dernier élément : deux méthodes pour le caractériser
  1. *Représentation contiguë avec **longueur explicite***
     Une variable contient **la longueur de la séquence (= nombre d'éléments)**.
  2. *Représentation contiguë avec marque*
     Une valeur spécifique est placée après le dernier élément de la séquence (*marque de fin*).

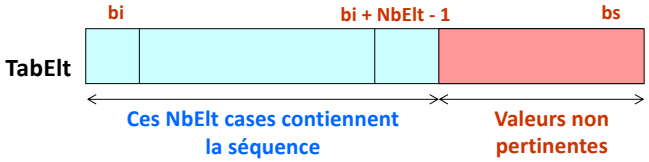


Erreur classique

confondre la taille du tableau et la taille de la séquence

c) Représentation avec longueur explicite

dans un tableau



Longueur max. = bs-bi+1 Séquence vide ⇔ NbElt = 0

Élément : **type**

bi, bs : **constante de type entier**

TabElt : **tableau sur** [bi … bs] **d'Elément**

NbElt : **entier sur** [0 … bs-bi+1]

Cas particulier (bi = 1)

Lmax : **constante de type entier > 0** *{longueur max.}*

TabElt : **tableau sur** [1…Lmax] **d' Élément**

NbElt : **entier sur** [0…Lmax]

d) Schémas de traitement séquentiel

Représentation contiguë avec longueur explicite

Élément : **type ;** bi, bs **: constante de type entier**

TabElt **: tableau sur** [bi…bs] **d'Élément ;** NbElt **: entier sur** [0…bs-bi+1]

Parcours

Initialiser le traitement

pour IC **allant de** bi **à** bi+NbElt-1

Traiter l'el. $TabElt_{IC}$

Recherche de l'indice du premier élément vérifiant la propriété P (indices croissants)

P : **fonction** (E: Élément) **→ booléen** *{Propriété}*

IC **: entier sur** [bi … bs+1] *{indice courant}*

IC ← bi

tant que IC < bi + NbElt **et puis non P(**$TabElt_{IC}$**)**

IC ← IC + 1

{on a trouvé en $TabElt_{IC}$ *⇔ IC ≠ bi + NbElt}*

Parcours partiel

Initialiser le traitement ; IC ← bi

tant que IC < bi + NbElt **et puis non P(**$TabElt_{IC}$**)**

Traiter l'el. $TabElt_{IC}$ **;** IC ← IC + 1

Traitement séquentiel d'un tableau : remplacer NbElt par bs-bi+1

Suppression des zéros

Version 1 : construire une nouvelle séquence

Polycopié E3.13

Décrire une action de suppression des zéros d'une séquence d'entiers

Lmax **: constante de type entier > 0** *{longueur maximum}*

TabEnt **: type tableau sur** [1…Lmax] **d'entier**

Ent **: type entier sur** [0…Lmax] *{N=0 ⇔ séquence vide}*

Sup_Zéros_1 **: action (donnée** T **:** TabEnt, N **:** Ent**; résultat** T' **:** TabEnt, N' **:** Ent**)**

Principe

Pour chaque él. non nul de $T_{[1…N]}$ (parcours), le placer dans $T'_{[1…N']}$ (aj. en queue)

Sup_Zéros_1(T, N, T', N') :

N' ← 0 *{mise à vide du résultat}*

pour i **allant de** 1 **à** N *{parcours de la donnée}*

{$T'_{[1…N']}$ *représente* $T_{[1…i-1]}$ *sans ses zéros}*

si $T_i \neq 0$ **alors**

N' ← N' + 1 ; $T'_{N'} \leftarrow T_i$ *{ajout en queue du résultat}*

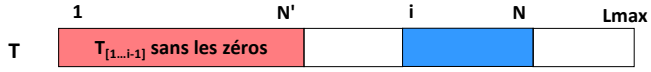
Suppression des zéros

Version 2 : modifier la séquence donnée

Sup_Zéros_2 **: action (donnée-résultat** T **:** TabEnt, N **:** Ent**)**

Principe

Le même : le résultat est implanté au début du tableau T entre 1 et N'



Sup_Zéros_2(T, N) :

N' **: entier sur** [0…Lmax] *{N' ≤ N}*

N' ← 0 *{mettre à vide le résultat}*

pour i **allant de** 1 **à** N *{parcours de la donnée}*

{$T_{[1…N']}$ *contient* $T_{[1…i-1]}$ *sans ses zéros}*

si $T_i \neq 0$ **alors**

N' ← N'+1 ; $T_{N'} \leftarrow T_i$ *{ajout en queue du résultat}*

N ← N' *{mise à jour nombre éléments résultat}*

Intersection de deux ensembles

Analyse par abstractions successives

Réaliser un algorithme d'intersection de deux ensembles d'entiers.

Principe : E3 ←E1∩E2
pour chaque élément de E1,
s'il appartient à E2,
le placer dans E3.

Représentation des ensembles par des séquences
Mise en œuvre des schémas
Parcours de E1
Recherche dans E2
si trouvé alors Ajout en queue dans E3

Lexique : séquences sous forme contiguë

Lmax **: constante de type entier > 0** *{cardinal max. des ensembles}*

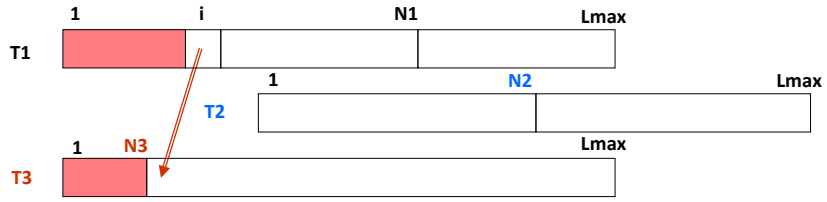
T1, T2, T3 **: tableau sur [**1…Lmax**] d'entier**

N1, N2, N3 **: entier sur [**0…Lmax**]**

Intersection de deux ensembles

Réalisation

invariant sous forme d'un dessin



j **: entier sur [**1…Lmax+1**]** *{pour la recherche dans E2}*
N3 ← 0 *{mettre à vide E3}*
pour i **allant de** 1 **à** N1 *{parcours de E1}*
{Invariant : $T3_{[1…N3]}$ représente $T1_{[1…i-1]} \cap T2_{[1…N2]}$}
j ← 1 *{recherche dans E2}*
tant que j ≠ N2+1 **et puis** $T2_j \neq T1_i$: j ← j+1
si j ≠ N2+1 **alors**
N3 ← N3+1 ; $T3_{N3} \leftarrow T1_i$ *{ajout dans E3, en queue}*

e) Vie d'une séquence

Actions de modification

Trois schémas de base à connaître
construire une séquence vide, ajouter ou supprimer un élément.

Cas général
- L'ordre des éléments de la séquence est significatif

Cas particuliers
(spécification et réalisation)
- L'ordre des éléments de la séquence n'est pas significatif
  la séquence sert à représenter un ensemble
- la séquence est triée
- l'ordre des éléments de la séquence caractérise l'ordre d'arrivée
  file d'attente : même ordre
  pile : ordre inverse

e) Vie d'une séquence

Construire la séquence vide

Ce qu'il ne faut pas faire
Remplir tout le tableau avec une même valeur (0, espace, …)

Longueur explicite à 0

N ← 0

e) Vie d'une séquence
Ajouter un élément (longueur explicite)

Ajouter un élément de valeur V à l'indice i

Pré-condition : (N < bs-bi+1) et (bi ≤ i ≤ bi + N)

- Cas général : l'ordre des éléments de la séquence est pertinent
  - Dans une séquence vide (N = 0, i = bi), ou en queue (i = bi + N)
    - N ← N + 1 ; T_i ← V
  - Dans une séquence non vide, en tête ou au milieu
    - Décaler à droite d'une position à partir de la position i
    - N ← N +1 ; T_i ← V
- Cas particulier (fréquent) : l'ordre n'est pas pertinent.

  La séquence représente un (multi) ensemble.
  - On ajoute toujours en queue (parce que c'est le plus facile)
    - N ← N + 1 ; T_N ← V

e) Vie d'une séquence
Supprimer un élément (longueur explicite)

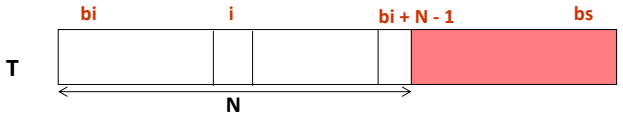
Supprimer l'élément se trouvant à l'indice i

Précondition : (N > 0) et (bi ≤ i ≤ bi + N - 1)

- Cas général : l'ordre des éléments de la séquence est pertinent
  - Dans une séquence singleton (N = 1, i = bi), ou en queue (i = bi + N - 1)
    - N ← N - 1
  - Dans une séquence non vide, en tête ou au milieu
    - Décaler à gauche d'une position à partir de la position i+1
    - N ← N - 1
- Cas particulier : l'ordre n'est pas pertinent.

  La séquence représente un (multi) ensemble.
  - On remplace par le dernier élément (parce que c'est le plus facile)
    - T_i ← T_{bi+N-1} ; N ← N - 1

e) Vie d'une séquence
Opérations de décalage (longueur explicite)



Décalage à droite d'une position

Utilisé pour l'ajout en i (milieu ou tête)

$\{T_{[i+1 \ldots bi+N]} \leftarrow T_{[i \ldots bi+N-1]}\}$

pour k **allant de** bi + N **à** i+1**, pas -1 :** $T_k \leftarrow T_{k-1}$

Décalage à gauche d'une position

Utilisé pour la suppression en i (milieu ou tête)

$\{T_{[i \ldots bi+N-2]} \leftarrow T_{[i+1 \ldots bi+N-1]}\}$

pour k **allant de** i **à** bi + N - 2 **:** $T_k \leftarrow T_{k+1}$

f) Séquences triées
Algorithmes importants

- Insertion d'un élément de valeur V dans une séquence triée
  - Chercher l'indice K d'insertion : algorithme de recherche
  - Décaler à droite
  - Placer V à l'indice K

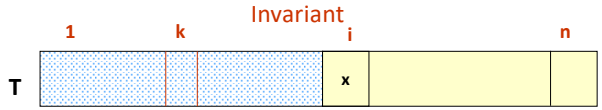
  Voir polycopié pour : réalisation, affinement, variantes
- Recherche dichotomique.
  - Uniquement pour une représentation contiguë (calcul d'indices)

  Voir polycopié

  Résultat : réduction logarithmique du coût
- Algorithmes de tri
  - Ils sont nombreux et largement étudiés

Tri par insertion

Prendre les éléments un par un dans un ordre quelconque. Les éléments déjà pris étant triés, placer le nouvel élément à sa place. Recommencer avec les éléments restants.



T contient une permutation des valeurs initiales de T. $T_{[1\ldots i-1]}$ est trié et contient une permutation des valeurs initiales de $T_{[1\ldots i-1]}$.

Exercice de rédaction : donner la réalisation

Voir aussi le tri par sélection du minimum (polycopié E3.4)

g) A propos de gestion mémoire

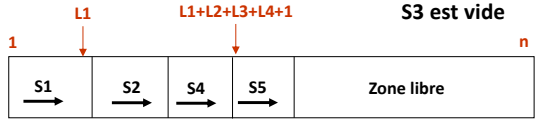
Cas des séquences implantées dans un seul tableau

Séquences de même type

solution 1 : un tableau par séquence

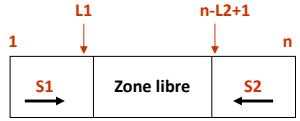
solution 2 : un seul tableau

plusieurs séquences



Le premier élément de S_i se trouve en position $L_1+\ldots+L_{i-1}+1$
Toutes les opérations nécessitent des décalages
(sauf pour les opérations en queue de S5)

deux séquences



S2 est implantée "en sens inverse"
Zone libre au milieu :
favorise les opérations en queue

3.3. Ensembles, relations, piles, files

3. Séquences et tableaux

3.0. Exemples introductifs
3.1. Tableaux
3.2. Séquences
3.3. Ensembles, relations, piles, files

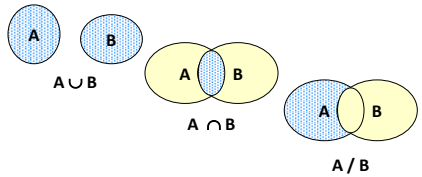
Représentation de structures

sous forme de séquences ou de tableaux

a) Ensembles

Algorithmes

- Opérations ensemblistes
  - Union
  - Intersection
  - Différence



- Actions de modification
  - Créer l'ensemble vide
  - Ajouter un élément
  - Supprimer un élément

a) Ensembles

Représentations

- Séquence d'éléments
  - Représentation contiguë : ajout en queue, suppression par échange avec le dernier (pas de décalage)
- Vecteur de booléens

  dans le cas de sous-ensembles d'un ensemble **E** de cardinal **n** fini
  - Les éléments de **E** sont identifiés par des entiers de 1 à n.
  - Chaque sous-ensemble **F** considéré est représenté par un vecteur **V** :

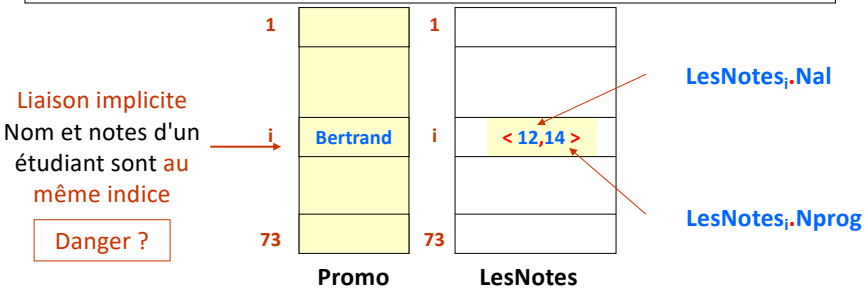
    V **: tableau sur [**1…n**] de booléen**

    {V_i a la valeur vrai si et seulement si l'élément identifié par i appartient à F}
  - Exercices : voir polycopié E3.19, E3.20

b) Relations n-aires

Résultats d'examens

Pour chaque étudiant, on connaît ses notes d'algorithmique et de programmation. Représenter les résultats à l'aide de tableaux.



NbEt **: constante de type entier =** 73 **;** Note **: type entier sur [**0…20**]**

Résultat **: type <**Nal **:** Note **;** Nprog **:** Note**>**

LesNotes **: tableau sur [**1… NbEt**] de** Résultat

{l'étudiant de nom Promo$_i$ a obtenu les notes LesNotes$_i$.Nal en algorithmique et LesNotes$_i$.Nprog en programmation.}

c) Relation binaire

Une représentation à l'aide d'une matrice de booléens

R** relation binaire sur un ensemble **E** (cardinal n) : sous-ensemble de **E** × **E

Exemple : liaisons routières

- **E** : ensemble de villes
- Le couple <ville1, ville2> appartient à **R** ⇔ il existe une liaison routière de la ville1 vers la ville2.

Représentation par une matrice de booléens

- On identifie chaque élément de **E** par un entier compris entre **1** et **n**.
- La relation **R** est représentée par une matrice de booléens

n **: constante de type entier > 0**

R **: tableau sur [**1…n**] de tableau sur [**1…n**] de booléen**

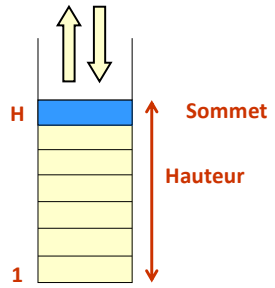
{$R_{i,j}$ a la valeur vrai ⇔ le couple formé de l'élément identifié par i et de l'élément identifié par j, appartient à la relation.}

Exercice : étudier d'autres représentations

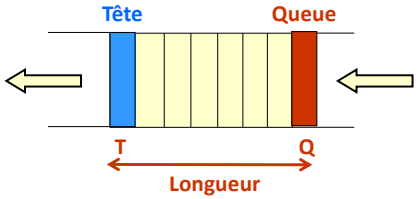
- ensemble de couples (sous forme d'une séquence)
- ensemble de villes et pour chaque ville, ensemble des villes qui lui sont reliées (séquence de séquences).

d) Piles

- Ajout et suppression se font à la même extrémité
  - Dernier arrivé, Premier servi
    - *En anglais : Last In First Out (LIFO)*
- Représentations
  - Celles utilisées pour les séquences
    - Noms usuels des primitives
      - Empiler(x) : ajouter la valeur x (au sommet)
        - *En anglais : Push*
      - Dépiler(x) : donner à x la valeur du sommet et supprimer le sommet ("décapiter")
        - *En anglais : Pop*



e) Files d'attente



- Ajout à une extrémité, suppression à l'autre
  - Premier arrivé, Premier servi
    - *En anglais : First In First Out (FIFO)*
- Représentations
  - Celles utilisées pour les séquences

Conclusion du chapitre 3

Éléments pour guider le travail hors séance

- Connaître les définitions

  Comprendre les notions sous-jacentes, savoir les utiliser
  - tableau, indirection, séquence, séquence triée, pile, file d'attente
  - schémas de parcours, de recherche, de modification
- Connaître les représentations de structures avec des tableaux

  Principes, lexiques généraux et schémas d'opérations
  - représentation contiguë d'une séquence
  - représentation d'un ensemble, d'une relation binaire ou n-aire
- Savoir
  - Construire une solution par abstractions successives
    - *ensemble → séquence → tableau*
    - *principe → schéma à appliquer → codage*
  - Ré-utiliser les exemples du chapitre (analogie)
  - Exprimer un principe de solution en termes de schémas
  - Composer les outils de structuration de données : n-uplets, tableaux
  - Appliquer les schémas lors de la réalisation
  - Vérifier un algorithme : respect des types, invariants, etc.

Vocabulaire

Savoir donner une définition ,un exemple

- tableau, tableau à deux dimensions, indice, rang
- linéarisation d'un tableau à deux dimensions
- parcours d'un tableau, recherche dans un tableau
- décalage

- séquence
- représentation contiguë ; longueur explicite, marque
- parcours d'une séquence, recherche dans une séquence
- séquence triée ; tri d'une séquence
- pile, file d'attente

4. Séquences et chaînage

4.1. Principes de représentation chaînée

Indirection
Représentation des séquences par des listes chaînées
Schémas itératifs de traitement de séquences

4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

Maîtriser l'indirection
Connaître les schémas et savoir les appliquer
- *pour exprimer un principe de solution*
- *pour réaliser la solution*

a) Indirection

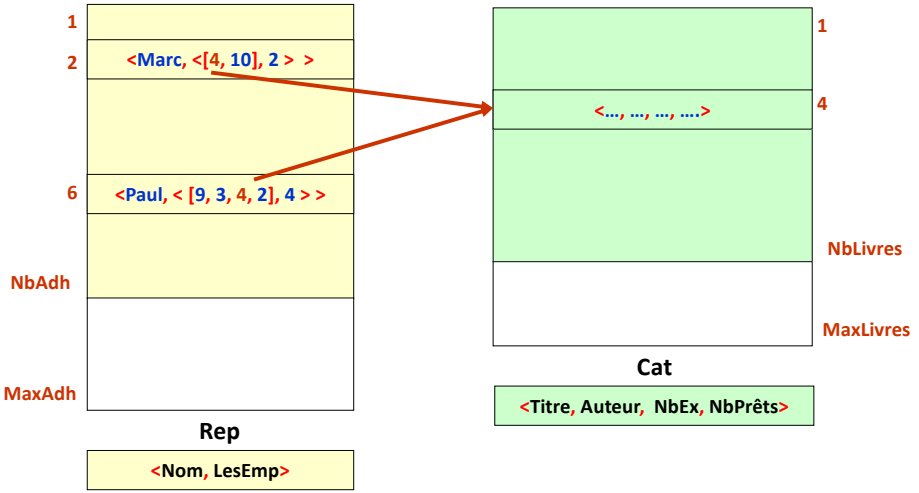
- Cellule : case d'un tableau, champ d'un produit, mot mémoire
- Adresse : indice d'un tableau, adresse en mémoire
  un compilateur associe une adresse en mémoire à chaque identificateur de variable

Définitions

- Accès direct : accès au contenu d'une cellule par son adresse
- Indirection, accès indirect
  - Une cellule d'adresse A contient l'adresse B d'une autre cellule.
  - L'adresse A permet d'accéder indirectement au contenu de la cellule d'adresse B.
- Liens explicites : sont représentés par des adresses.
  - *Succession dans une séquence* : à partir d'un élément, accès indirect à son successeur

Gestion d'une bibliothèque

Indirection : partage d'informations



b) Séquences : représentation chaînée

Principes

Représenter chacun des éléments de la séquence.

et représenter la fonction de succession.

Représentation contiguë de S dans un tableau T

- E avant F dans S ⇔ indice associé à E < indice associé à F.
- La fonction de succession est calculée (calcul sur les indices)
  - Indice de Suc(E) = Indice de E + 1
  - Accès direct au successeur

Représentation chaînée de S dans un tableau T

- Les éléments de S sont dispersés (non contigus) dans T.
- La fonction de succession est tabulée (chaînage)
  - chaque élément est associé à son successeur par un lien explicite (adresse du successeur).
  - Accès indirect au successeur

b) Séquences : représentation chaînée

Conventions de représentation

Définitions

- tête de liste :
  - adresse de la cellule contenant le premier élément de la liste
- queue de liste
  - adresse de la cellule contenant le dernier élément de la liste
- Nil : valeur d'adresse particulière (lien vers "nulle part")

Conventions

- Accès à la liste
  - une variable contient la tête de liste
- Accès à la valeur d'un élément
  - indirectement via le prédécesseur
- Liste vide : la tête de liste a pour valeur Nil
- Adresse du successeur du dernier élément : Nil

Répertoire téléphonique

Représentation contiguë (tableau de tableaux)

Une liste (colonne) pour chaque groupe d'initiales (non trié)

T

	1	2	3	4
1	<Carol, 05632842>	<Jean, …>		<Zoé, …>
2	<Anne, …>	<Henri, …>		<Xavier, …>
3	<Fernand, …>	<Martine, …>		
4	<Agnès, …>			
5				
6				
7				
8				
9				

L

4	3	0	2

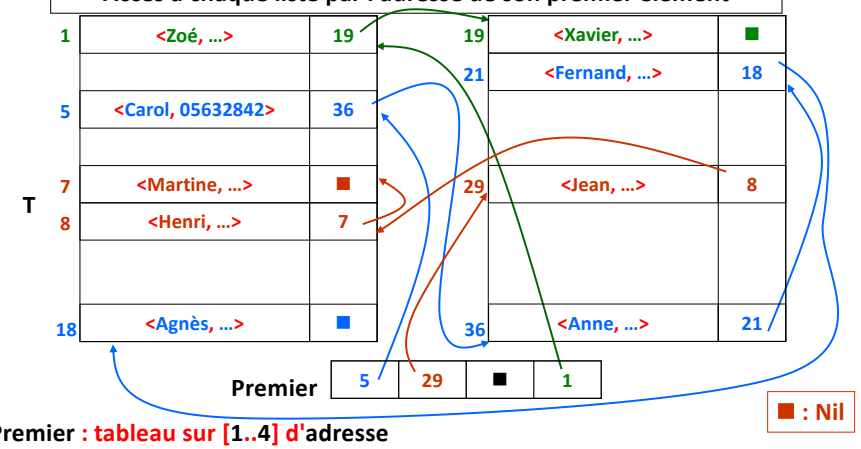
L : tableau sur [1..4] d'entier

T : tableau sur [1..9] de tableau sur [1..4] de <Nom, Num>

Répertoire téléphonique

Représentation chaînée

Une liste marquée pour chaque groupe d'initiales (non triée)
Accès à chaque liste par l'adresse de son premier élément



Premier : tableau sur [1..4] d'adresse

T : tableau sur [1..36] de <<Nom, Numéro>, adresse>

Une liste ordonnée de prénoms

Représentation contiguë

Insertions successives de Jean, Valentin, Anne, Laure, Julien, Agnès, Léa
puis suppressions successives de Agnès, Valentin, Laure

T : tableau sur [1..n] de Nom

NbP = 7

	Après les insertions
1	Agnès
2	Anne
3	Jean
4	Julien
5	Laure
6	Léa
7	Valentin
n	

NbP = 4

	Après les suppressions
1	Anne
2	Jean
3	Julien
4	Léa
n	

Représentation contiguë
Insertions et suppressions par décalage : les éléments sont déplacés

Une liste ordonnée de prénoms

Représentation chaînée

Insertions successives de
Jean, Valentin, Anne, Laure, Julien, Agnès, Léa

puis suppressions successives de
Agnès, Valentin, Laure

T : tableau sur [1..n]

de <Nom, Adresse>

Premier = 6

1	Jean	5
2	Valentin	■
3	Anne	1
4	Laure	7
5	Julien	4
6	Agnès	3
7	Léa	2
n		

Après les insertions

Premier = 3

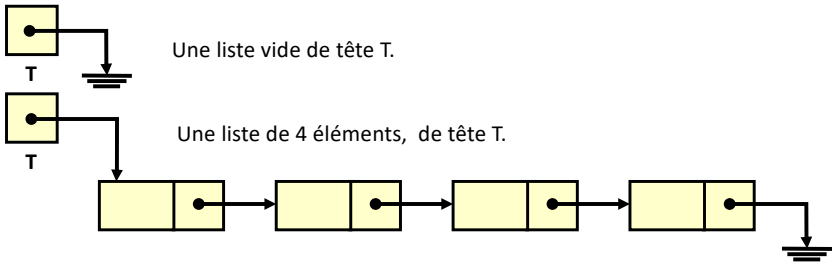
1	Jean	5
2	Valentin	■
3	Anne	1
4	Laure	7
5	Julien	7
6	Agnès	3
7	Léa	■
n		

Après les suppressions

Représentation chaînée
Les éléments sont dispersés dans la mémoire
Insertion et suppressions par modification des liens : pas de déplacement des éléments

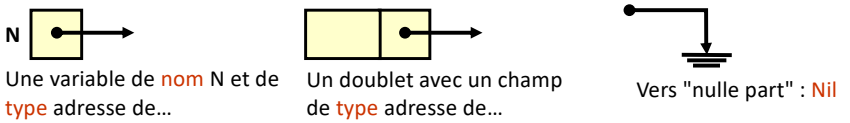
c) Raisonner sur les listes chaînées

Faire abstraction des valeurs d'adresse



Chaque flèche donne deux informations

Point d'arrivée = valeur d'adresse ; Point de départ = endroit où l'adresse est stockée



d) Gestion de l'espace mémoire

Problématique

Un espace mémoire : ensemble de cellules

Chaque cellule : <Élément, Adresse du successeur>

- Une cellule est libre ou occupée
- Mémoire libre = ensemble des cellules libres

Deux acteurs	Dans les algorithmes
• usager de la mémoire	• Traitement des listes chaînées
• gestionnaire de la mémoire	• Gestion de l'espace mémoire

d) Gestion de l'espace mémoire

Interface de gestion mémoire

Faire abstraction de la gestion mémoire

- X est une variable de type adresse
- **Allouer(**X**)** *X est un paramètre de statut résultat*
  - L'usager demande qu'une cellule libre lui soit réservée et que son adresse lui soit fournie dans la variable de nom X.
  - Le gestionnaire mémoire recherche une cellule libre, la marque comme étant occupée, et fournit son adresse dans X.
- **Libérer(**X**)** *X est un paramètre de statut donnée*
  - L'usager ne veut plus utiliser la cellule d'adresse X. Il la restitue au gestionnaire de la mémoire.
  - Le gestionnaire la marque comme étant à nouveau libre.

Statut des paramètres ?

e) Représentation des liens : pointeurs

Faire abstraction de la représentation de l'espace mémoire

Besoin linguistique

Manipuler des adresses sans connaître leurs valeurs

- Définir des variables dont les valeurs sont des adresses
- Accéder au contenu associé à une adresse
- Comparer des adresses (=, ≠)
- Disposer de primitives de gestion de l'espace mémoire
  - pour fournir des valeurs d'adresse
- On complète le langage par un constructeur de type
  - Forme générale : **pointeur de** T où T est un nom de type

e) Représentation des liens : pointeurs

Le constructeur de type "pointeur de ... "

- Définition de variables, de types
  - Variable **X : pointeur d'entier**
  - Type **adEnt : type pointeur d'entier**
  - Cellules de listes chaînées d'entiers

    adCelEnt : type pointeur de CelEnt

    CelEnt : type <E : entier, Suc : adCelEnt>

    T : adCelEnt
- Une valeur d'adresse particulière de nom Nil
  - lien vers "nulle part" (n'est associé à aucune cellule)

e) Représentation des liens : pointeurs

Opérations

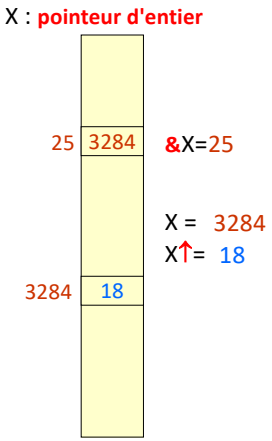
- ↑ ("déréférençage") : accès indirect
  - N est le nom d'une variable de type **pointeur de** T
  - N↑ est le nom d'une variable de type T

    *En C : *N*
  - Nil↑ est incorrect
- **&** ("référençage") : Adresse associée à un nom
  - N est le nom d'une variable de type T
  - **&**N est le nom d'une variable de type **pointeur de** T

    - sa valeur est l'adresse associée au nom N
  - Propriétés

    *- A de type **pointeur de** T : A ≡ **&**(A↑)*

    *- B de type T : B ≡ **(&**B)↑*
- = et ≠ : comparaison de pointeurs de même type
- **Allouer**, **Libérer**, **MémoireSaturée** : gestion mémoire



f) Accès aux valeurs d'une liste chaînée

Expression de chemins d'accès

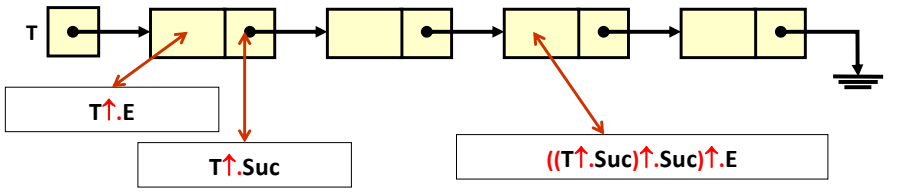
Élément : **type**

adCel : **type pointeur de** Cel

Cel : **type <**E : Élément, Suc : adCel**>**

T : adCel

Pour chaque flèche traversée sur le dessin, une indirection (↑) dans l'expression



en C

T↑.Suc s'écrit : (*T).Suc mais aussi T->Suc

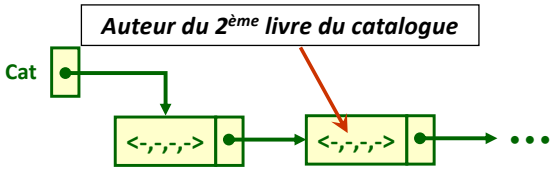
(T↑.Suc)↑.E s'écrit : (*(*T).Suc).E mais aussi (T->Suc)->E

Gestion d'une bibliothèque
Représentation chaînée

Catalogue, répertoire, ensemble d'emprunts sont représentés par des listes chaînées

```
{catalogue}
    AdCat : type pointeur de CelCat
    CelCat : type <Liv : Livre, LivreSuiv : AdCat>
    Cat : AdCat                                    {tête de liste}
{Ensembles d'emprunts}
    AdEmp : type pointeur de CelEmp
    CelEmp : type <Emp : AdCat, EmpSuiv : AdEmp>
{répertoire}
    AdRep : type pointeur de CelRep
    CelRep : type <Adh : Adhérent, AdhSuiv : AdRep>
    Rep : adRep                                    {tête de liste}
```


Gestion d'une bibliothèque
Chemins dans la structure : exemple 1



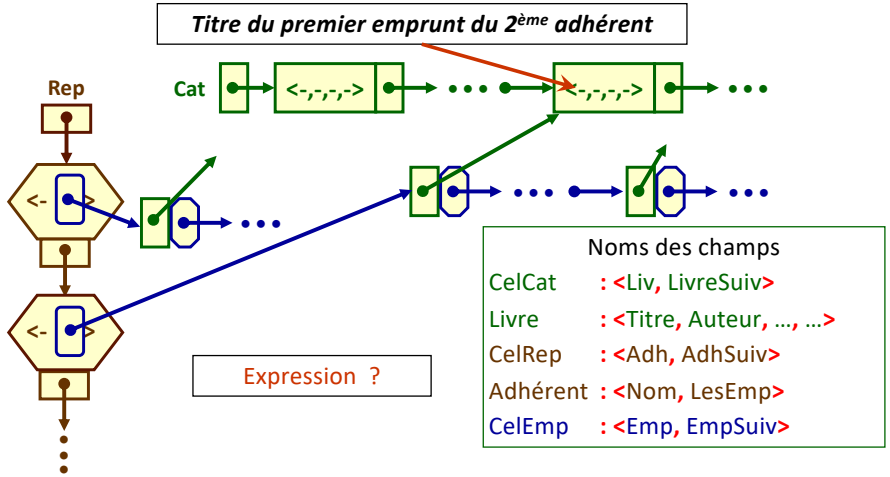
Expression ?

Noms des champs
CelCat : <Liv, LivreSuiv>
Livre : <Titre, Auteur, …, …>

((Cat↑.LivreSuiv)↑.Liv).Auteur

Vérifier les expressions de chemin
Nombre d'indirections, type de chaque terme de l'expression

Gestion d'une bibliothèque
Chemins dans la structure : exemple 2



Expression ?

Noms des champs
CelCat : <Liv, LivreSuiv>
Livre : <Titre, Auteur, …, …>
CelRep : <Adh, AdhSuiv>
Adhérent : <Nom, LesEmp>
CelEmp : <Emp, EmpSuiv>

(((Rep↑. AdhSuiv)↑.Adh.LesEmp)↑.Emp)↑.Liv.Titre

4.2. Traitement des listes chaînées
Séquences et chaînage

4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

Où l'on retrouve les schémas
traitement séquentiel, construction, modification de listes chaînées
et les étapes de résolution
Énoncé, spécification, principe, réalisation, correction, analyse quantitative

Maîtriser l'expression de l'indirection
Connaître les exemples, savoir les réutiliser (analogie)
Savoir donner un principe de solution en termes de schémas

a) Traitement séquentiel

Représentation chaînée

adCel : **type pointeur de** Cel
Cel : **type <**Elt : Elément**,** Suc : adCel**>**
T, AC : adCel *{tête et adresse courante}*

Parcours

initialiser le traitement
AC ← T
tant que AC ≠ Nil
    Traiter l'el. courant AC↑.Elt
    AC ← AC↑.Suc

Recherche de l'adresse du premier élément vérifiant la propriété P (sens du chaînage)

P : **fonction(**E : Elément**)→booléen** *{propriété}*
AC ← T
tant que AC ≠ Nil **et puis non** P**(**AC↑.Elt**)**
    AC ← AC↑.Suc
{trouvé en AC ⇔ AC ≠ Nil}

Parcours partiel

Initialiser le traitement ; AC ← T
tant que AC ≠ Nil **et puis non** P**(**AC↑.Elt**)**
    Traiter l'el. courant AC↑.Elt ; AC ← AC↑.Suc

Comparer avec les schémas donnés au chapitre 3 (transparents 24 et 27).
Ici on raisonne sur la séquence des adresses ; Nil sert de marque.

Nombre d'éléments positifs

Traitement d'une liste chaînée, spécification

Déterminer le nombre d'éléments positifs dans une liste chaînée d'entiers.

Spécifications

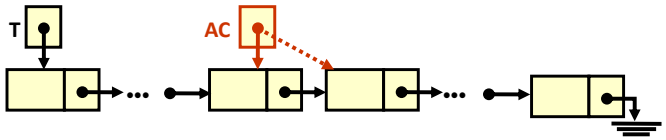
Lexique pour la représentation chaînée ?
Fonction ou action ? Paramètres ?

{on particularise le lexique général}
CelE : **type <**E : entier**,** Suc : adCelE**>** *{ suffixe E pour "entier"}*
adCelE : **type pointeur de** CelE

NbPos : **fonction (**T : adCelE**) → entier ≥ 0**
{ Nombre d'éléments positifs de la liste de tête T }

Nombre d'éléments positifs

Réalisation, parcours d'une liste chaînée



INVARIANT

Parcours de la séquence des adresses, marquée par Nil

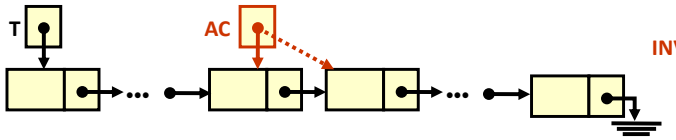
NbPos**(**T**)** :
    AC : adCelE *{adresse courante}*
    Nb : **entier ≥ 0**
    AC ← T
    Nb ← 0
    tant que AC ≠ Nil
        {Nb = nombre de positifs avant l'élément courant}
        si AC↑.E **> 0 alors** Nb ← Nb **+** 1
        AC ← AC↑.Suc
    retour : Nb

Premier élément positif

Recherche dans une liste chaînée

Déterminer l'adresse du premier élément positif dans une liste chaînée.

adPremPos : **fonction (**T : adCelE**) →** adCelE
{ Adresse du premier élément positif de la liste d'adresse de tête T, ou Nil s'il n'y en a pas }



INVARIANT

Recherche sur la séquence des adresses, marquée par Nil

adPremPos**(**T**)** :
    AC : adCelE *{adresse courante}*
    AC ← T
    tant que AC ≠ Nil **et puis non (**AC↑.E**> 0)**
        AC ← AC↑.Suc
    retour : AC

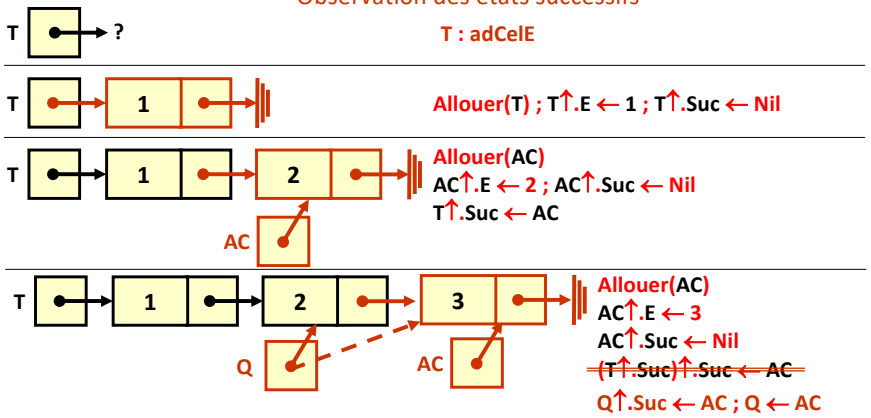
b) Construction d'une liste chaînée

Polycopié E4.2

Expérimentation

Construire une liste chaînée des entiers de 1 à n (vide si n = 0)

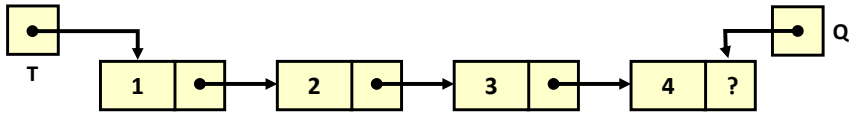
Observation des états successifs



Idée : maintenir l'adresse de queue

b) Construction d'une liste chaînée

Version 1 : par ajouts en queue



Version 1 : parcours des entiers de 1 à n

- Ajouts successifs en queue
- Cas particuliers : liste vide, création du premier élément

T, Q : adCelE *{Q adresse de queue}*

AC : adCelE *{pour créer la cellule de l'élément courant}*

si n = 0 **alors** créer liste vide

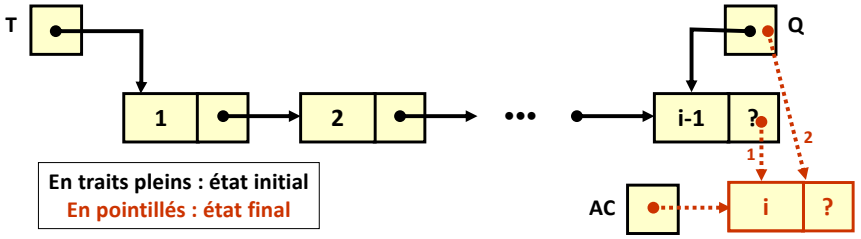
sinon créer liste singleton avec la valeur 1

pour i **allant de** 2 **à** n : Ajouter en queue la valeur i

Mettre Nil à la fin

b) Construction d'une liste

Ajouter un élément en queue de liste

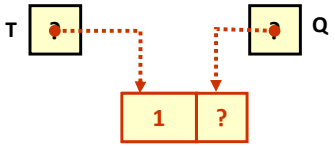


En traits pleins : état initial
En pointillés : état final

- Allocation d'une nouvelle cellule : **Allouer(AC)**
- Stockage de la valeur d'élément (i) : **AC↑.E ← i**
- Connexion à la liste, ajout en queue : **Q↑.Suc ← AC**
- Mise à jour de l'adresse de queue : **Q ← AC**

b) Construction d'une liste

Création du premier élément d'une liste



- Allocation de la cellule de tête : **Allouer(T)**
- Stockage de la valeur d'élément (1) : **T↑.E ← 1**
- Initialisation de l'adresse de queue : **Q ← T**

b) Construction d'une liste

Version 2 : par ajouts en tête

Parcours des entiers de n à 1

- On procède par ajouts successifs en tête
- Pas de cas particulier.

Créer liste vide

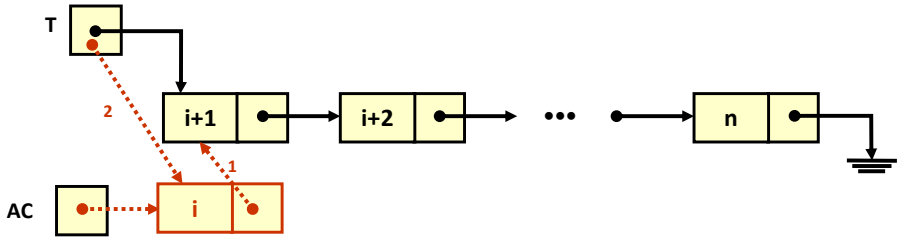
pour i **allant de** n **à** 1**, pas -1 :** Ajouter en tête (valeur i)

⚠ Ajout en queue : préserve l'ordre d'arrivée

Ajout en tête : inverse l'ordre d'arrivée

b) Construction d'une liste

Ajouter un élément en tête d'une liste



- Allocation d'une nouvelle cellule : **Allouer(AC)**
- Stockage de la valeur d'élément (i) : **AC↑.E ← i**
- Connexion à la liste, ajout en tête : **AC↑.Suc ← T**
- Mise à jour de l'adresse de tête : **T ← AC**

b) Construction d'une liste chaînée

Par "recopie" des éléments d'une séquence donnée

Deux méthodes à connaître

Construction par ajouts successifs en queue

L'ordre des éléments est conservé

Initialiser le résultat avec le premier élément

Initialiser l'adresse de queue

Pour chaque élément de la liste donnée

l'ajouter en queue de la liste résultat

Construction par ajouts successifs en tête

L'ordre des éléments est inversé

Initialiser le résultat avec une liste vide

Pour chaque élément de la liste donnée

l'ajouter en tête de la liste résultat

c) Schémas de modification

d'une liste chaînée

Des schémas à maîtriser

- Créer une liste vide T ← Nil
- Ajouter une cellule
- Supprimer la cellule
- Libérer une liste

Grille d'analyse

- Séquence vide
- Séquence non vide
  - *en tête, au milieu ou en queue*

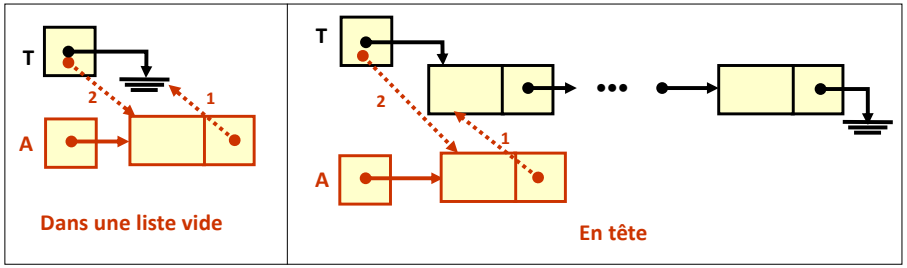
Regroupement des cas

	Ajouter	Supprimer
contiguë	vide - en queue	en queue
	au milieu - en tête	au milieu - en tête
chaînée	vide - en tête	en tête
	au milieu - en queue	au milieu - en queue

c) Schémas de modification

Ajout dans une liste vide ou en tête

Élément : **type**
adCel : **type pointeur de** Cel
Cel : **type** <Elt : Elément, Suc : adCel>
T : adCel



Création de l'élément

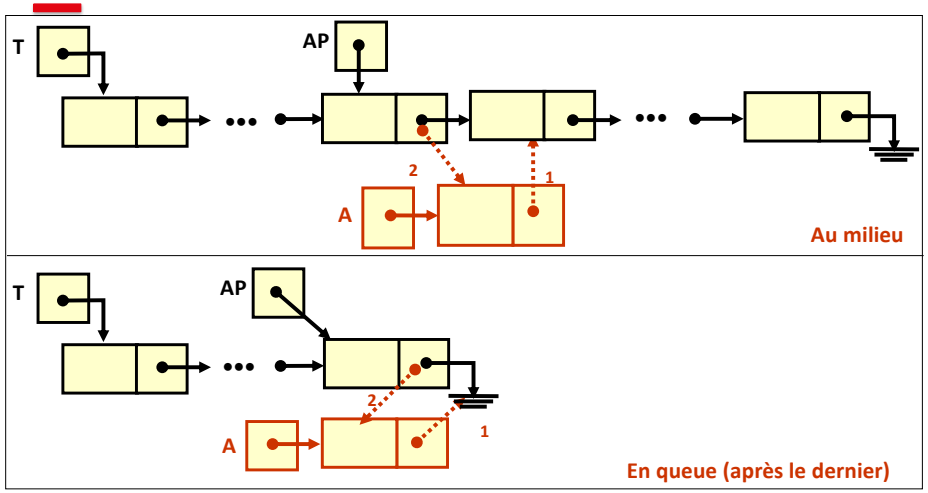
Allouer(A) ; A↑.Elt ← …

Connexion de A en tête

A↑.Suc ← T ; T ← A

c) Schémas de modification

Ajout après la cellule d'adresse AP

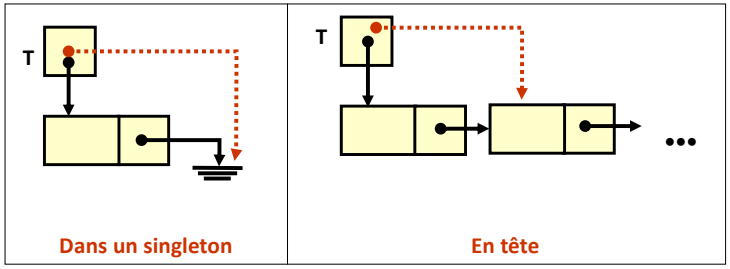


Connexion de A après AP

A↑.Suc ← AP↑.Suc ; AP↑.Suc ← A

c) Schémas de modification

Suppression en tête



Suppression en tête

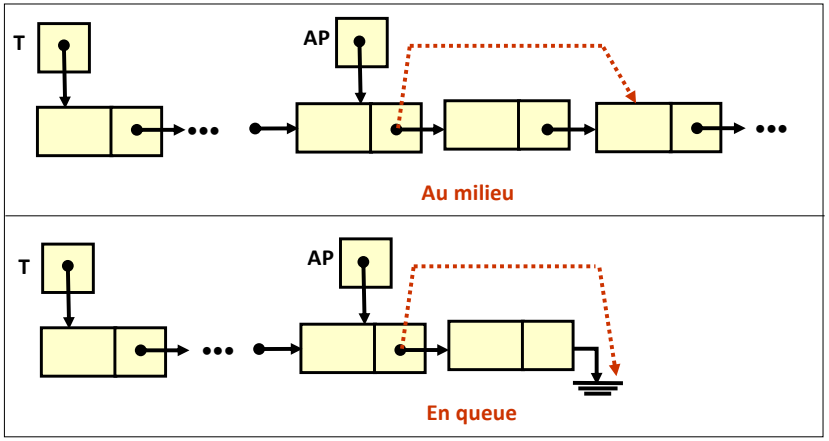
~~**Libérer(T) ; T ← T↑.Suc**~~ Erreur

X ← T ; T ← T↑.Suc ; Libérer(X)

Libérer(A) ne modifie pas la valeur de A, mais peut modifier la valeur de A↑

c) Schémas de modification

Suppression du successeur de AP



Suppression après AP

X ← AP↑.Suc ; AP↑.Suc ← X↑.Suc ; Libérer(X)

c) Schémas de modification

Libération d'une liste

On procède par suppressions en tête successives

```
{liste de tête T}
tant que T ≠ Nil
    X ← T
    T ← T↑.Suc
    Libérer(X)
```

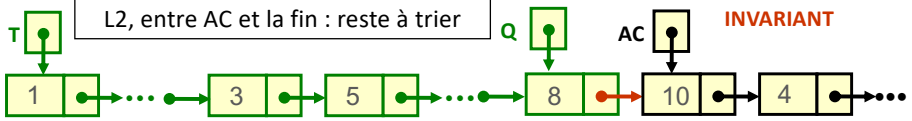

Tri par insertion d'une liste chaînée

Analyse : mettre en place l'invariant (par un dessin)

Polycopié E4.5

L1, entre T et Q : partie déjà triée
L2, entre AC et la fin : reste à trier

Q↑.Suc = AC
INVARIANT



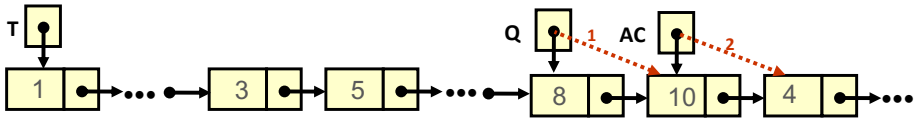
Ébauche de l'algorithme

Initialiser L1 avec le premier doublet donné, L2 avec le reste
tant que L2 n'est pas vide
    selon AC
        AC doit être inséré en queue de L1 (AC est en place) :
            mettre à jour Q et AC
        AC doit être inséré en tête de L1 :
            insérer en tête et mettre à jour Q↑.Suc et AC
        AC doit être inséré au milieu de L1 :
            chercher le point d'insertion
            insérer et mettre à jour Q↑.Suc et AC

Tri par insertion d'une liste chaînée

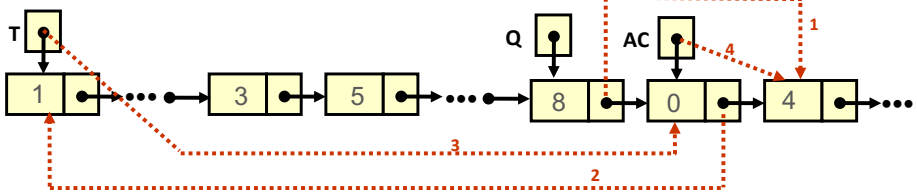
Insertion, cas 1 et 2

Cas où le doublet courant est en place : $Q\uparrow.E \leq AC\uparrow.E$



Q ← AC ; AC ← Q↑.Suc

Cas où le doublet courant doit être inséré avant T : $AC\uparrow.E < T\uparrow.E$

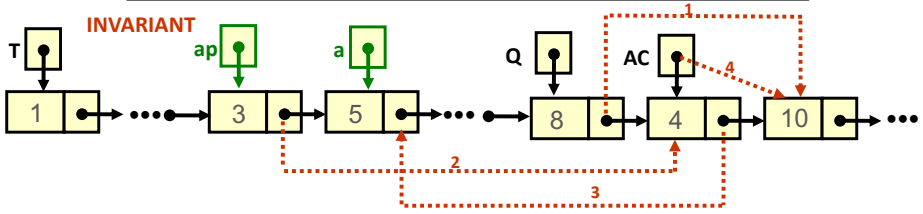


Q↑.Suc ← AC↑.Suc ; AC↑.Suc ← T ; T ← AC ; AC ← Q↑.Suc

Tri par insertion d'une liste chaînée

Insertion, cas 3

Cas où le doublet courant doit être inséré entre T et Q :
$T\uparrow.E \leq AC\uparrow.E$ et $AC\uparrow.E < Q\uparrow.E$



{Point d'insertion ?}

```
ap ← T ; a ← T↑.Suc
tant que a↑.E ≤ AC↑.E
    ap ← a ; a ← a↑.Suc
```

Q↑.E sert de sentinelle

{Mise à jour des liens }

```
Q↑.Suc ← AC↑.Suc
ap↑.Suc ← AC
AC↑.Suc ← a
AC ← Q↑.Suc
```

4.3. Variantes pour la représentation

Séquences et chaînage

4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

a) Principes

Enrichir la représentation d'une liste chaînée

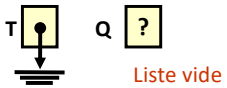
- Pour faciliter l'écriture d'un algorithme
- Le cas échéant, de manière temporaire

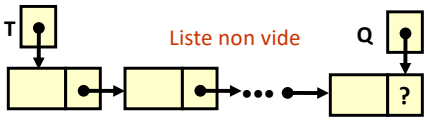
Pointeur de queue
Élément fictif en tête
Liste circulaire avec fictif entre queue et tête
Double chaînage



Etudier chaque variante
- *Adapter les schémas*
- *Reprendre un exercice simple avec la variante*

b) Liste avec pointeur de queue

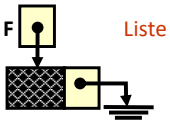


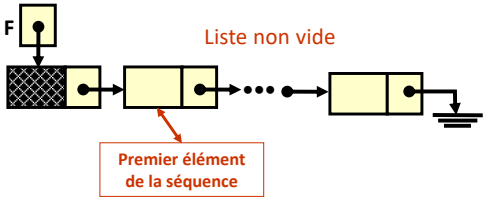


- Favorise les ajouts en queue
- Cas particuliers
  - Suppression en queue ≠ suppression au milieu
  - Ajout dans liste vide ≠ ajout en tête
- Utile pour la "recopie" d'une séquence

c) Liste avec élément fictif de tête

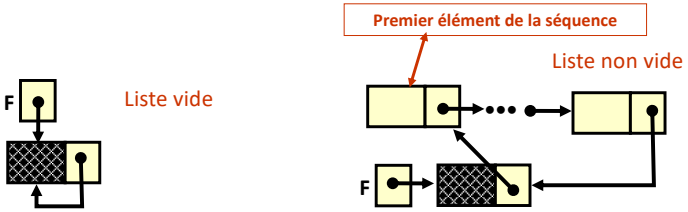
Unifier les cas





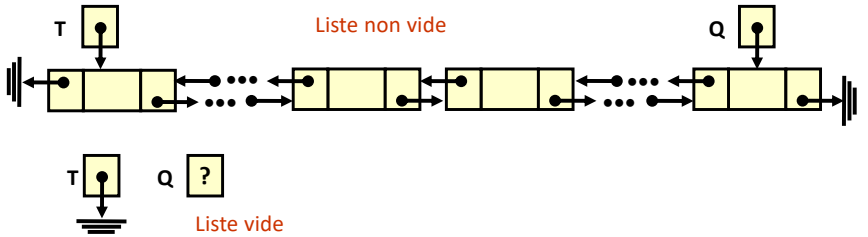
- Accès à la liste par l'adresse F de l'élément fictif
- Ajout en tête ≡ ajout après F
- Suppression en tête ≡ suppression après F
- Non pertinent si on ne fait que des ajouts en tête
  - Ordre non pertinent
  - Pile

d) Liste circulaire avec élément fictif



- La suite des adresses des éléments de la séquence
  - Commence par F↑.Suc
  - Est marquée par F
- L'élément fictif peut servir pour placer une "sentinelle" lors d'une recherche

e) Liste doublement chaînée



Les cellules sont des triplets

adCel **: type pointeur de** Cel

Cel **: type <**Elt **:** Elément**,** Suc **:** adCel**,** Préd **:** adCel**>**

Enrichissements supplémentaires

liste doublement chaînée, circulaire avec élément fictif

4.4. Interclassement de séquences triées

4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

a) Position du problème

- Données
  - Deux séquences triées **S1** et **S2**
- Résultat
  - Une séquence triée **S3**, union de **S1** et **S2**
- Condition de réalisation
  - Exploiter le fait que les séquences sont triées

Diverses formes d'énoncé

- Création
  - S3 ← S1 ∪ S2
- Modification
  - S1 ← S1 ∪ S2

b) Principe d'interclassement
de séquences triées

Parcours simultané des deux séquences

- Tant qu'aucune des deux séquences n'est épuisée
  - Comparer les deux éléments courants
  - Ajouter le cas échéant l'un des éléments à la liste résultat
  - Passer à l'élément suivant
- Recopier, le cas échéant, la séquence non épuisée

Application : opérations ensemblistes

Union

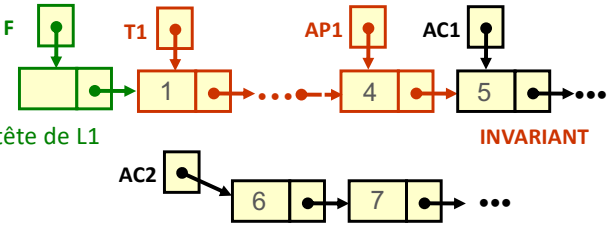
Intersection

Différence

c) Interclassement de deux listes triées
Analyse

Polycopié E4.6

L1 ← L1 ∪ L2



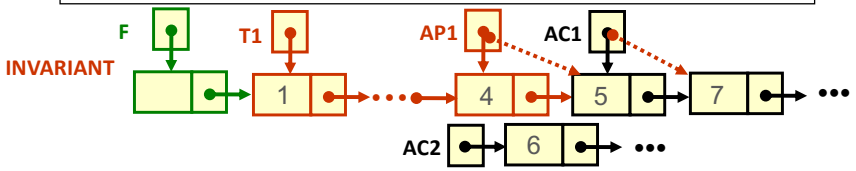
F : fictif ajouté en tête de L1

Invariant

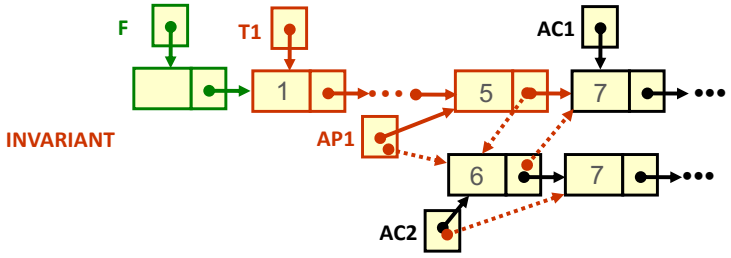
- AC1 et AC2 : adresses courantes
- de T1 à AP1 : interclassement des éléments de L1 et L2 déjà traités
- AP1 ≠ F ⇒ ((AP1↑.E ≤ AC1↑.E)
  et (AC2 ≠ Nil ⇒ AP1↑.E ≤ AC2↑.E))
- AP1↑.Suc = AC1

c) Interclassement de deux listes triées
Réalisation

Cas où AC1↑.E ≤ AC2↑.E : Avance sur L1



Cas où AC2↑.E < AC1↑.E : AC2↑ est placé entre AP1 et AC1



4.5. Séquences de séquences
Séquences et chaînage

4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

Partition d'un ensemble de mots

Séquence de séquences

Polycopié E4.7

Soit E un ensemble de mots distincts deux à deux
et soit P la partition des mots de E selon leur initiale.
Construire P à partir de E - Construire E à partir de P

Analyse des informations

- Un mot est une séquence non vide de lettres
- E est un ensemble de mots
- P est un ensemble d'ensembles non vides de mots
  - Chaque classe de P est caractérisée par une lettre (initiale commune à tous les mots de la classe)

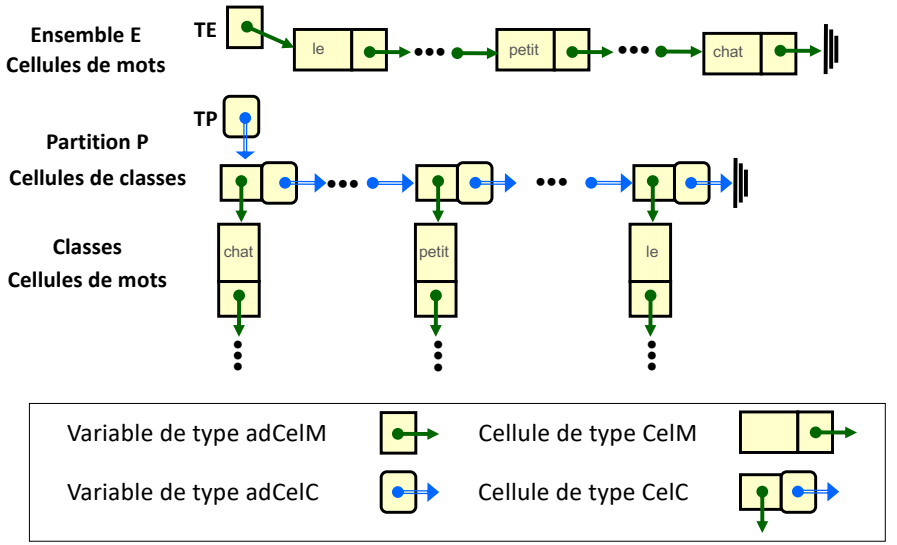
Représentations

Tout ensemble est représenté par une séquence elle-même sous forme chaînée

- E est une liste chaînée de mots
- P est une liste chaînée de classes
- Chaque classe est une liste chaînée non vide de mots

Partition d'un ensemble de mots

Illustration de la représentation

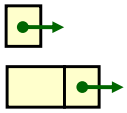


Partition d'un ensemble de mots

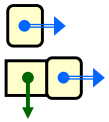
Lexique

{Mots}
Lettre **: type caractère** *{restreint aux lettres de l'alphabet}*
Mot **: type** séquence de lettres *{on fait abstraction de la représentation}*

{Listes de mots}
adCelM **: type pointeur de** CelM
CelM **: type <**M **:** Mot**,** Msuiv **:** adCelM**>**



{Listes de Classes (ensembles de mots de même initiale)}
adCelC **: type pointeur de** CelC
CelC **: type <**T **:** adCelM**,** Csuiv **:** adCelC**>**

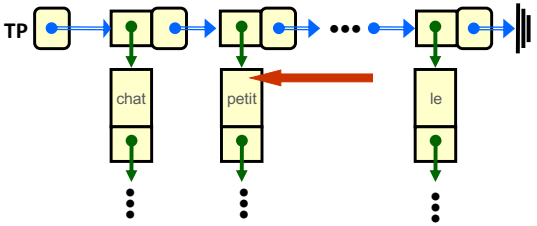


{L'ensemble E et la partition P}
TE **:** adCelM *{tête de la liste de mots représentant E}*
TP **:** adCelC *{tête de la liste de classes représentant P}*

Partition d'un ensemble de mots

Expressions de chemins

Compléter l'instruction ***Écrire (……)*** *pour afficher la lettre initiale des mots de la deuxième classe de la partition.*



Adresse du 2° élément de la liste représentant P TP↑.Csuiv

Adresse de tête de la liste représentant la 2° classe de P (TP↑.Csuiv)↑.T

Premier mot de la 2° classe de P ((TP↑.Csuiv)↑.T)↑.M

Initiale des mots de la 2° classe de P Premier(((TP↑.Csuiv)↑.T)↑.M)

Partition d'un ensemble de mots

Construction de la partition : principe

Réaliser un algorithme qui pour E donné crée la partition P des mots de E selon leur initiale

La liste représentant E n'est pas modifiée.

Principe

Pour chaque mot de E (parcours),

Recopier le mot dans une nouvelle cellule de mot

S'il existe dans P (recherche) une classe correspondant à son initiale,

Ajouter le mot en tête de cette classe (aucun ordre imposé)

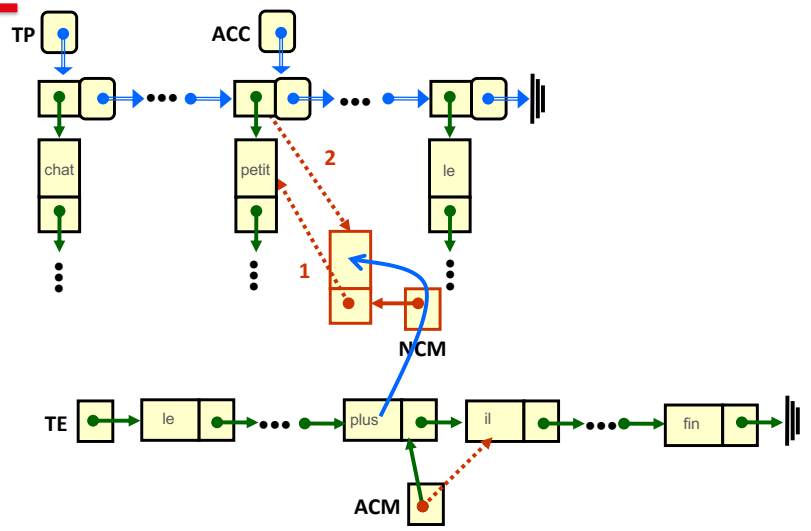
sinon, créer une nouvelle classe avec ce mot

par ajout en tête de la liste des classes (aucun ordre imposé).

Variante : P est construite en utilisant les cellules de E (à l'état final, la liste représentant E est vide)

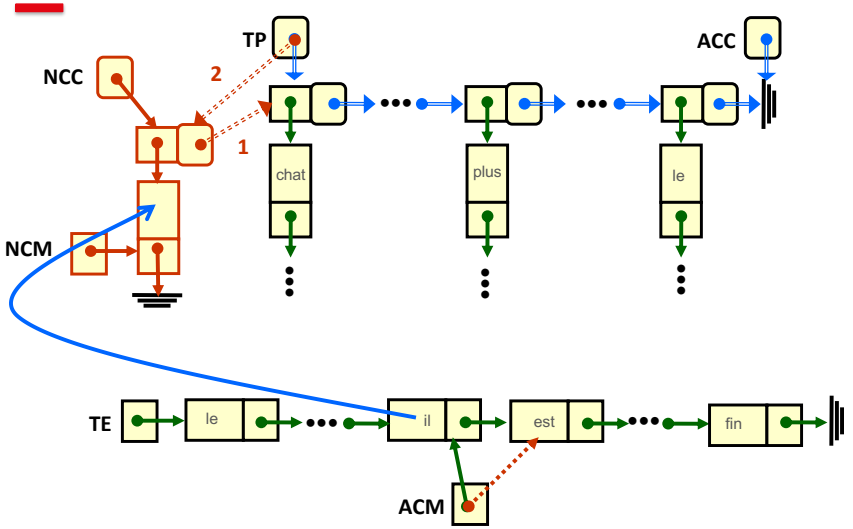
Partition d'un ensemble de mots

Construction de P : ajout dans une classe existante



Partition d'un ensemble de mots

Construction de P : création d'une nouvelle classe



Partition d'un ensemble de mots

Construction de P : algorithme

```
ACM : adCelM                                   {pour le parcours de E.}
ACC : adCelC                                   {pour la recherche dans P.}
NCM : adCelM                                   {pour la création de cellules de mots.}
NCC : adCelC                                   {pour la création de cellules de classes.}
TP ← Nil                                       {créer la partition vide}
ACM ← TE
tant que ACM ≠ Nil
    Allouer(NCM) ; NCM↑ ← <ACM↑.M, Nil>        {créer une cellule de mot}
    {recherche de la classe de l'élément courant}
    ACC ← TP
    tant que ACC ≠ Nil et puis premier(ACM↑.M) ≠ premier((ACC↑.T)↑.M)
                ACC ← ACC↑.Csuiv
    si ACC ≠Nil alors {insertion en tête de classe}
                NCM↑.Msuiv ← ACC↑.T ; ACC↑.T ← NCM
    sinon {création d'une nouvelle classe}
                Allouer(NCC) ; NCC↑ ← <NCM, TP> ; TP ← NCC
    ACM ← ACM↑.Msuiv
```

4.6. Gestion du chaînage dans un tableau

Séquences et chaînage

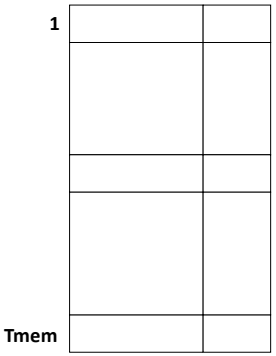
4.1. Principes de représentation chaînée
4.2. Traitement des listes chaînées
4.3. Variantes pour la représentation
4.4. Interclassement de séquences triées
4.5. Séquences de séquences
4.6. Gestion du chaînage dans un tableau

a) Gestion mémoire à l'aide d'un tableau

Principe

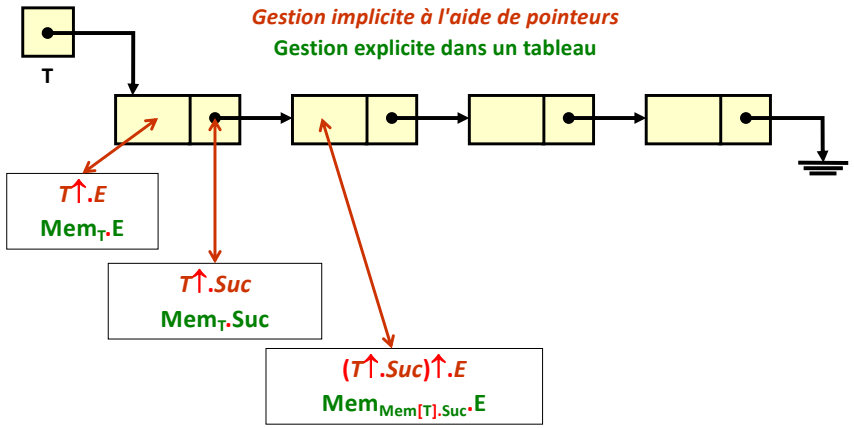
- L'espace mémoire est représenté par un tableau
  couples <élément, adresse du successeur>
- Une adresse est un indice dans ce tableau.

Elément **: type**
Tmem **: constante de type entier > 0**
Cel **: type <**E **:** Elément**,** Suc **:** adCel**>**
adCel **: type entier sur [0…**Tmem**]**
Nil **: constante 0 de type** adCel
Mem **: tableau sur [1…**Tmem**] de** Cel



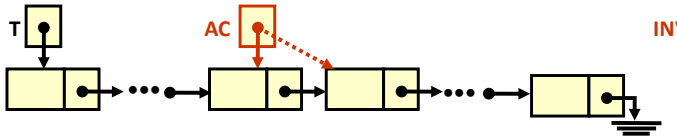
b) Accès aux valeurs

Expression de chemin d'accès



Nombre d'éléments positifs

Parcours d'une liste chaînée



Parcours de la séquence des adresses, marquée par Nil

```
Nb : entier ≥ 0
AC : adCelEnt                          {adresse courante}
AC ← T
Nb ← 0
tant que AC ≠ Nil
    si MEM_AC.E > 0 alors Nb ← Nb + 1
       AC ← MEM_AC.Suc
```

c) Gestion de l'espace mémoire

Liste libre

Liste chaînée de toutes les cellules libres (ordre non pertinent)

TL : adCel

Allocation

- On alloue la cellule de tête de la liste libre
- Déconnexion en tête de la liste libre

Allouer(X) : X ← TL ; TL ← Mem_{TL}.Suc

Libération

- Connexion en tête de la liste libre de la cellule restituée.

Libérer(X) : Mem_X.Suc ← TL ; TL ← X

Conclusion du chapitre 4

Éléments pour guider le travail hors séance

- Connaître les définitions

  Comprendre les notions sous-jacentes, savoir les utiliser
  - adresse, indirection, lien explicite de chaînage, liste chaînée
  - interface de gestion mémoire (allocation, libération de cellules)
  - conventions de représentation d'une liste chaînée à l'aide de dessins
- Connaître la représentation chaînée des séquences

  Principes, lexiques généraux, schémas d'opérations
  - représentation standard et variantes
- S'approprier les notations concernant les pointeurs
- Savoir
  - Construire une solution par abstractions successives
    - *ensemble → séquence → liste chaînée*
    - *principe → schéma à appliquer → codage à partir d'un dessin*
  - Appliquer les schémas
    - *parcours, recherche, création de la liste vide, insertion, suppression*
  - et toujours
    - *exprimer un principe de solution, vérifier un algorithme (types, invariants, etc.)*

Vocabulaire

Savoir donner une définition ,un exemple

- adresse, chaînage
- accès direct, accès indirect, indirection
- gestion mémoire, allocation mémoire, libération mémoire

- représentation chaînée d'une séquence, liste chaînée
- tête de liste, queue de liste
- élément fictif, liste circulaire, liste doublement chaînée

- pointeur, déréférençage

- principe d'interclassement