Concepts de base

Nicolas Delestre, Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr

Modifié pour l'ENSICAEN par
Luc Brun
luc.brun@ensicaen.fr

Plan...

- Le formalisme utilisé
- Qu'est ce qu'une variable ?
- Qu'est ce qu'un type ?
- Qu'est ce qu'une expression ?
- Qu'est ce qu'une affectation
- Les entrées / sorties ?

Formalisme...

- Un algorithme doit être lisible et compréhensible par plusieurs personnes
- Il doit donc suivre des règles
- Il est composé d'une entête et d'un corps :
- l'entête, qui spécifie :
- le nom de l'algorithme **(Nom :)**
- son utilité **(Rôle :)**
- les données "en entrée", c'est-à-dire les éléments qui sont indispensables à son bon fonctionnement (**Entrée :**)
- les données "en sortie", c'est-à-dire les éléments calculés, produits, par l'algorithme (**Sortie :**)
- les données locales à l'algorithmique qui lui sont indispensables (**Déclaration :**)

Formalisme...

- le corps, qui est composé :
  - du mot clef **début**
  - d'une suite d'instructions **indentées**
  - du mot clef **fin**

Formalisme...

- Exemple de code :

Nom: ajoutDeuxEntiers
Role: Additionner deux entiers a et b et mettre le résultat dans c
Entrée: a,b : entier
Sortie: c : entier
Déclaration: -
début
    c $\leftarrow$a+b
fin

Qu'est ce qu'une variable...

- Une variable est une entité qui contient une information :
- une variable possède un nom, on parle **d'identifiant**
- une variable possède une valeur
- une variable possède un type qui caractérise l'ensemble des valeurs que peut prendre la variable
- L'ensemble des variables sont stockées dans la mémoire de l'ordinateur

Nommâge des variables

- Le nom d'une variable ne doit pas comporter d'espaces,
- Il doit être **significatif** (sauf pour les variables de boucle).
- Les noms de variables doit être construit en fonction de règles et être **systématique** :

- La règle : LowercaseMixedCapital.
- Premier mot qui compose le nom de la variable en minuscule,
- chaque mot suivant qui compose le nom de la variable prend une capitale.

Exemple : ajoutDeuxEntiers, estPremier...

- Autre règle :
- Tout en minuscule,
- mots séparés par des soulignés (_).

Exemple : ajout_deux_entiers, est_premier...

Qu'est ce qu'une variable...

- On peut faire l'analogie avec une armoire qui contiendrait des tiroirs étiquetés :

- l'armoire serait la mémoire de l'ordinateur
- les tiroirs seraient les variables (l'étiquette correspondrait à l'identifiant)
- le contenu d'un tiroir serait la valeur de la variable correspondante
- la couleur du tiroir serait le type de la variable (bleu pour les factures, rouge pour les bons de commande, etc.)

Qu'est ce qu'un type de données...

- Le type d'une variable caractérise :
- l'ensemble des valeurs que peut prendre la variable
- l'ensemble des actions que l'on peut effectuer sur une variable
- Lorsqu'une variable apparaît dans l'entête d'un algorithme on lui associe un type en utilisant la syntaxe suivante
- `Identifiant de la variable : Son type`
- Par exemple :
- `age : Naturel`
- `nom : Chaine de Caractères`
- Une fois qu'un type de données est associé à une variable, cette variable ne peut plus en changer
- Une fois qu'un type de données est associé à une variable le contenu de cette variable doit **obligatoirement** être du même type

Qu'est ce qu'un type de données...

- Par exemple, dans l'exemple précédent on a déclaré `a` et `b` comme des entiers

- `a` et `b` dans cet algorithme ne pourront pas stocker des réels

- `a` et `b` dans cet algorithme ne pourront pas changer de type

- Il y a deux grandes catégories de type :

- les types simples

- les types complexes (que nous verrons dans la suite du cours)

- Il y a deux grandes catégories de type simple :
  - Ceux dont le nombres d'éléments est fini, les **dénombrables**
  - Ceux dont le nombre d'éléments est infini, les **indénombrables**

Les types simples dénombrables...

- **booléen**, les variables ne peuvent prendre que les valeurs VRAI ou FAUX
- **intervalle**, les variables ne peuvent prendre que les valeurs entières définies dans cet intervalle, par exemple 1..10
- **énuméré**, les variables ne peuvent prendre que les valeurs explicitées, par exemple les jours de la semaine (du lundi au dimanche)
- Ce sont les seuls types simples qui peuvent être définis par l'informaticien
- **caractères**
- Exemples :

```
masculin : booleen
mois : 1..12
jour : JoursDeLaSemaine
```

Notez les règles pour le nom du type `JoursDeLaSemaine`.

Cas des énumérés...

- Si l'informaticien veut utiliser des énumérés, il doit définir le type dans l'entête de l'algorithme en explicitant toutes les valeurs de ce type de la façon suivante :
- *nom du type* = {*valeur1, valeur2, ..., valeurn*}
- Par exemple :
- JoursDeLaSemaine = {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche}

Les types simples indénombrables...

- **entier** (positifs et négatifs)
- **naturel** (entiers positifs)
- **réel**
- **chaîne de caractères**, par exemple 'cours' ou 'algorithmique'
- Exemples :

```
age : naturel
taille : reel
nom : chaine de caractères
```

Opérateur, opérande et expression...

- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs"
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type

Opérateur, opérande et expression...

- Par exemple dans `a+b` :
- `a` est l'opérande gauche
- `+` est l'opérateur
- `b` est l'opérande droite
- `a+b` est appelé une expression
- Si par exemple `a` vaut 2 et `b` 3, l'expression `a+b` vaut 5
- Si par exemple `a` et `b` sont des entiers, l'expression `a+b` est un entier

Opérateur...

- Un opérateur peut être unaire ou binaire :
- **Unaire**s'il n'admet qu'une seule opérande, par exemple l'opérateur `non`
- **Binaire**s'il admet deux opérandes, par exemple l'opérateur +
- Un opérateur est associé à *un* type de donnée et ne peut être utilisé qu'avec des variables, des constantes, ou des expressions de *ce* type
- Par exemple l'opérateur + ne peut être utilisé qu'avec les types arithmétiques (naturel, entier et réel) ou (exclusif) le type chaîne de caractères
- **On ne peut pas additionner un entier et un caractère**
- Toutefois *exceptionnellement* dans certains cas on accepte d'utiliser un opérateur avec deux opérandes de types différents, c'est par exemple le cas avec les types arithmétiques $(2+3.5)$

Opérateur...

- La signification d'un opérateur peut changer en fonction du type des opérandes
- Par exemple l'opérateur + avec des entiers aura pour sens l'addition, mais avec des chaînes de caractères aura pour sens la **concaténation**
- 2+3 vaut 5
- `"bonjour" + " tout le monde"` vaut `"bonjour tout le monde"`

Les opérateurs booléens...

- Pour les booléens nous avons les opérateurs `non`, `et`, `ou`, `ouExclusif`

- `non`

a	non a
Vrai	Faux
Faux	Vrai

- `et`

a	b	a et b
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Les opérateurs booléens...

- `ou`

a	b	a ou b
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

- `ouExclusif`

a	b	a ouExclusif b
Vrai	Vrai	Faux
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Les opérateurs sur les énumérés...

- Pour les énumérés nous avons trois opérateurs `succ`, `pred`, `ord` :

- `succ` permet d'obtenir le successeur, par exemple avec le type `JourDeLaSemaine` :

  - `succ Lundi vaut Mardi`
  - `succ Dimanche vaut Lundi`

- `pred` permet d'obtenir le prédécesseur, par exemple avec le type `JourDeLaSemaine` :

  - `pred Mardi vaut Lundi`
  - `pred Lundi vaut Dimanche`

- `ord` permet d'obtenir le naturel de l'énuméré spécifié dans la bijection du type énuméré vers les naturels, par exemple avec le type `JourDeLaSemaine` :

  - `ord Lundi vaut 0`
  - `ord Dimanche vaut 6`

- Pour les caractères on retrouve les trois opérateurs des énumérés avec en plus un quatrième opérateur nommé `car` qui est le dual de l'opérateur `ord` avec comme fonction de bijection la table de correspondance de la norme ASCII
- Cf. http ://www.commentcamarche.net/base/ascii.htm
- Par exemple
  - `ord A` vaut 65
  - `car 65` vaut `A`
  - `pred A` vaut `@`
- L'opérateur pour les chaînes de caractères
  - C'est l'opérateur de concaténation vu précédemment qui est `+`

Les opérateurs sur les naturels, entiers et réels...

- On retrouve tout naturellement `+`, `-`, `/`, `*`
- Avec en plus pour les naturels et les entiers `div` et `mod`, qui permettent respectivement de calculer une division entière et le reste de cette division, par exemple :

- `11 div 2` vaut 5
- `11 mod 2` vaut 1

L'opérateur d'égalité, d'inégalité, etc...

- L'opérateur d'égalité
  - C'est l'opérateur que l'on retrouve chez tous les types simples qui permet de savoir si les deux opérandes sont égales
  - Cet opérateur est représenté par le caractère $=$
  - Une expression contenant cet opérateur est un booléen
- On a aussi l'opérateur d'inégalité $\neq$
- Et pour les types possédant un ordre les opérateurs de comparaison $<, \leq, \geq, >$

Priorité des opérateurs...

- Tout comme en arithmétique les opérateurs ont des priorités
- Par exemple * et / sont prioritaires sur + et -
- Pour les booléens, la priorité des opérateurs est `non`, `et`, `ouExclusif` et `ou`
- Pour clarifier les choses (ou pour dans certains cas supprimer toutes ambiguités) on peut utiliser des parenthèses

Actions sur les variables...

- On ne peut faire que deux choses avec une variable :

1. Obtenir son contenu (*regarder le contenu du tiroir*)

- Cela s'effectue simplement en nommant la variable

2. Affecter un (nouveau) contenu (mettre une (nouvelle) information dans le tiroir)

- Cela s'effectue en utilisant l'opérateur d'affectation représenter par le symbole $\leftarrow$
- La syntaxe de cet opérateur est : `identifiant de la variable` $\leftarrow$ `expression sans opérateur d'affectation`

- Par exemple l'expression `c ← a + b` se comprend de la façon suivante :
- On prend la valeur contenue dans la variable `a`
- On prend la valeur contenue dans la variable `b`
- On additionne ces deux valeurs
- On met ce résultat dans la variable `c`
- Si `c` avait auparavant une valeur, cette dernière est perdue !

Les entrées/sorties...

- Un algorithme peut avoir des interactions avec l'utilisateur
- Il peut afficher un résultat (du texte ou le contenu d'une variable) et demander à l'utilisateur de saisir une information afin de la stocker dans une variable
- En tant qu' informaticien on raisonne en se mettant **"à la place de la machine"**, donc :

- Pour afficher une information on utilise la commande **écrire** suivie entre parenthèses de la chaîne de caractères entre guillemets et/ou des variables de type simple à afficher séparées par des virgules, par exemple :

- **écrire**("Le valeur de la variable a est", a)

- Pour donner la possibilité à l'utilisateur de saisir une information on utilise la commande **lire** suivie entre parenthèses de la variable de type simple qui va recevoir la valeur saisie par l'utilisateur, par exemple :

- **lire**(b)

Exemple d'algorithme...

Nom: euroVersFranc1
Role: Convertisseur des sommes en euros vers le franc, avec saisie de la somme en euro et affichage de la somme en franc
Entrée: -
Sortie: -
Déclaration: valeurEnEuro,valeurEnFranc,tauxConversion : Réel
début
    tauxConversion ←6.55957
    écrire("Votre valeur en euro :")
    lire(valeurEnEuro)
    valeurEnFranc ←valeurEnEuro * tauxConversion
    écrire(valeurEnEuro," euros = ",valeurEnFranc," Frs")
fin

Exemple d'algorithme...

Nom: euroVersFranc2
Role: Convertisseur des sommes en euros vers le franc
Entrée: valeurEnEuro : Réel
Sortie: valeurEnFranc : Réel
Déclaration: tauxConversion : Réel
début
    tauxConversion ←6.55957
    valeurEnFranc ←valeurEnEuro * tauxConversion
fin

Conditionnelles et itérations

Nicolas Delestre et Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr
Modifié pour l'ENSICAEN par :
Luc Brun
luc.brun@ensicaen.fr

Plan...

- Rappels sur la logique
- Les conditionnelles
- Les itérations

Rappels sur la logique booléenne...

- Valeurs possibles : `Vrai` ou `Faux`
- Opérateurs logiques : `non` `et` `ou`
- optionellement `ouExclusif` mais ce n'est qu'une combinaison de `non` , `et` et `ou`
- a `ouExclusif` b = (`non` a `et` b) `ou` (a `et` `non` b)
- Priorité sur les opérateurs : `non` , `et` , `ou`
- Associativité des opérateurs `et` et `ou`
- `a et (b et c) = (a et b) et c`
- Commutativité des opérateurs et et ou
- `a et b = b et a`
- `a ou b = b ou a`

Rappels sur la logique booléenne...

- Distributivité des opérateurs `et` et `ou`
  - `a ou (b et c)` = `(a ou b) et (a ou c)`
  - `a et (b ou c)` = `(a et b) ou (a et c)`
- Involution
  - `non non a = a`
- Loi de Morgan
  - `non (a ou b)` = `non a et non b`
  - `non (a et b)` = `non a ou non b`

Les conditionnelles...

- Jusqu'à présent les instructions d'un algorithme étaient **toutes** interprétées **séquentiellement**

- **Nom:** euroVersFranc2
  Role: Convertisseur des sommes en euros vers le franc
  Entrée: valeurEnEuro : Réel
  Sortie: valeurEnFranc : Réel
  Déclaration: tauxConversion : Réel
  début
  tauxConversion ← 6.55957
  valeurEnFranc ← valeurEnEuro * tauxConversion
  fin

- Mais il se peut que l'on veuille conditionner l'exécution d'un algorithme

- Par exemple la résolution d'une équation du second degré est conditionnée par le signe de Δ

L'instruction `si alors sinon`...

- L'instruction `si alors sinon` permet de conditionner l'exécution d'un algorithme à la valeur d'une expression booléenne

- Sa syntaxe est :
  si *expression booléenne* **alors**
      suite d'instructions exécutées si l'expression est vrai
  sinon
      suite d'instructions exécutées si l'expression est fausse
  finsi

- Le deuxième partie de l'instruction est optionnelle, on peut avoir la syntaxe suivante :
  si *expression booléenne* **alors**
      suite d'instructions exécutées si l'expression est vrai
  finsi

Exemple (1/3)

Nom: abs
Role: Calcule la valeur absolue d'un entier
Entrée: unEntier : **Entier**
Sortie: laValeurAbsolue : **Entier**
Déclaration: -
début
  si unEntier $\geq$ 0 **alors**
    laValeurAbsolue $\leftarrow$unEntier
  sinon
    laValeurAbsolue $\leftarrow$-unEntier
  finsi
fin

Exemple (2/3)

Nom: max
Role: Calcule le maximum de deux entiers
Entrée: lEntier1,lEntier2 : **Entier**
Sortie: leMaximum : **Entier**
Déclaration: -

```
début
    si lEntier1 < lEntier2 alors
        leMaximum ←lEntier2
    sinon
        leMaximum ←lEntier1
    finsi
fin
```

Exemple (3/3)

Nom: max
Role: Calcule le maximum de deux entiers
Entrée: lEntier1,lEntier2 : **Entier**
Sortie: leMaximum : **Entier**
Déclaration: -

```
début
    leMaximum ←lEntier1
    si lEntier2 > lEntier1 alors
        leMaximum ←lEntier2
    finsi
fin
```

L'instruction cas

- Lorsque l'on doit comparer une **même** variable avec plusieurs valeurs, comme par exemple :

si a=1 **alors**
    faire une chose
sinon
    si a=2 **alors**
        faire une autre chose
    sinon
        si a=4 **alors**
            faire une autre chose
        sinon
            ...
        finsi
    finsi
finsi

- On peut remplacer cette suite de `si` par l'instruction cas

L'instruction cas

- Sa syntaxe est :

cas où v **vaut**

  $v1$: $action_1$

  $v2_1, v2_2, \ldots, v2_m$: $action_2$

  $v3_1 \ldots v3_2$: $action_3$

  ...

  vn : $action_n$

  autre : *action*

fincas

- où :

- v1,...,vn sont des **constantes** de type **scalaire** (entier, naturel, énuméré, ou caractère)
- $action_i$ est exécutée si v = vi (on quitte ensuite l'instruction cas)
- *action* est exécutée si $\forall$ i, v $\neq$ vi

Exemple

Nom: moisA30Jours
Role: Détermine si un mois à 30 jours
Entrée: mois : **Entier**
Sortie: resultat : **Booléen**
Déclaration:
début
    cas où mois **vaut**
        4,6,9,11 : résultat $\leftarrow$Vrai
        autre : résultat $\leftarrow$Faux
    fincas
fin

Les itérations

- Lorsque l'on veut répéter plusieurs fois un même traitement, plutôt que de copier n fois la ou les instructions, on peut demander à l'ordinateur d'exécuter n fois un morceau de code

- Il existe deux grandes catégories d'itérations :

- Les itérations **déterministes** :

  le nombre de boucle est défini à l'entrée de la boucle

- les itérations **indéterministes** :

  l'exécution de la prochaine boucle est conditionnée par une expression booléenne

- Il existe une seule instruction permettant de faire des boucles déterministes, c'est l'instruction `pour`

- Sa syntaxe est :

  pour *identifiant d'une variable de type scalaire* ←*valeur de début* **à** *valeur de fin* **faire**
  instructions à exécuter à chaque boucle
  finpour

- dans ce cas la variable utilisée prend successivement les valeurs comprises entre *valeur de début* et *valeur de fin*

Exemple

Nom: somme
Role: Calculer la somme des n premiers entiers positifs, $s=0+1+2+\ldots+n$
Entrée: n : **Naturel**
Sortie: s : **Naturel**
Déclaration: i : **Naturel**
début
    s $\leftarrow$ 0
    pour i $\leftarrow$0 **à** n **faire**
        s $\leftarrow$ s+i
    finpour
fin

Les itérations indéterministes

- Il existe deux instructions permettant de faire des boucles indéterministes :
- L'instruction **tant que** :

tant que *expression booléenne* **faire**
    instructions
fintantque

- qui signifie que tant que l'expression booléenne est vraie on exécute les instructions
- L'instruction **répéter jusqu'à ce que** :

répéter
    instructions
jusqu'à ce que *expression booléenne*

- qui signifie que les instructions sont exécutées jusqu'à ce que l'expression booléenne soit vraie

Les itérations indéterministes

- À la différence de l'instruction *tant que*, dans l'instruction *répéter jusqu'à ce que* les instructions sont exécutées au moins une fois
- Si vous ne voulez pas que votre algorithme "tourne" indéfiniment, l'expression booléenne doit faire intervenir des variables dont le contenu doit être modifié par au moins une des instructions du corps de la boucle

Un exemple

Nom: invFact
Role: Détermine le plus grand entier e tel que e! $\leq$ n
Entrée: n : **Naturel**$\geq$ 1
Sortie: e : **Naturel**
Déclaration: fact : **Naturel**
début
    fact $\leftarrow$ 1
    e $\leftarrow$ 1
    tant que fact $\leq$ n **faire**
        e $\leftarrow$ e+1
        fact $\leftarrow$ fact*e
    fintantque
    e $\leftarrow$ e-1
fin

Explication avec n=10

Nom: invFact
Role: Détermine le plus grand entier e tel que e ! $\leq$ n
Entrée: n : **Naturel**$\geq$ 1
Sortie: e : **Naturel**
Déclaration: fact : **Naturel**
début
  fact $\leftarrow$ 1
  e $\leftarrow$ 1
  tant que fact $\leq$ n **faire**
    e $\leftarrow$ e+1
    fact $\leftarrow$ fact*e
  fintantque
  e $\leftarrow$ e-1
fin

	n	e	fact	fact $\leq$ n
fact $\leftarrow$1	10	?	1	vrai
e $\leftarrow$1	10	1	1	vrai
e $\leftarrow$e+1 fact $\leftarrow$fact*e	10	2	2	vrai
e $\leftarrow$e+1 fact $\leftarrow$fact*e	10	3	6	vrai
e $\leftarrow$e+1 fact $\leftarrow$fact*e	10	4	24	faux
e $\leftarrow$e-1	10	3	24	faux

Un autre exemple

Nom: calculerPGCD
Role: Calculer le pgcd(a,b) à l'aide de l'algorithme d'Euclide
Entrée: a,b : **Naturel** non nul
Sortie: pgcd : **Naturel**
Déclaration: reste : **Naturel**

début
    répéter
        reste $\leftarrow$ a mod b
        a $\leftarrow$ b
        b $\leftarrow$ reste
    jusqu'à ce que reste=0
    pgcd $\leftarrow$ a
fin

	a	b	reste	reste = 0
initialisation	21	14	?	?
itération 1	14	7	7	faux
itération 2	7	0	0	vrai

Variables (locales et globales), fonctions et procédures

Nicolas Delestre et Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr

Modifié pour l'ENSICAEN par :
Luc Brun
luc.brun@ensicaen.fr

Plan

- Rappels
- Les sous-programmes
- Variables locales et variables globales
- Structure d'un programme
- Les fonctions
- Les procédures

Vocabulaire

- Dans ce cours nous allons parler de “programme” et de “sous-programme”
- Il faut comprendre ces mots comme “programme algorithmique” indépendant de toute implantation

Rappels

- La méthodologie de base de l'informatique est :

1. Abstraire

- Retarder le plus longtemps possible l'instant du codage

2. **Décomposer**

- " diviser chacune des difficultés que j'examinerai en autant de parties qu'il se pourrait et qu'il serait requis pour les mieux résoudre." Descartes

3. Combiner

- Résoudre le problème par combinaison d'abstractions

Par exemple

- Résoudre le problème suivant :

  Écrire un programme qui affiche en ordre croissant les notes d'une promotion suivies de la note la plus faible, de la note la plus élevée et de la moyenne

- Revient à résoudre les problèmes suivants :
  - Remplir un tableau de naturels avec des notes saisies par l'utilisateur
  - Afficher un tableau de naturels
  - Trier un tableau de naturel en ordre croissant
  - Trouver le plus petit naturel d'un tableau
  - Trouver le plus grand naturel d'un tableau
  - Calculer la moyenne d'un tableau de naturels
- Chacun de ces sous-problèmes devient un nouveau problème à résoudre
- Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait "quasiment" résoudre le problème initial

Sous-programme

- Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial
- En algorithmique il existe deux types de sous-programmes :
  - Les fonctions,
  - Les procédures.
- Un sous-programme est obligatoirement caractérisé par un nom (un identifiant) unique
- Lorsqu'un sous programme a été explicité (on a donné l'algorithme), son nom devient une nouvelle instruction, qui peut être utilisé dans d'autres (sous-)programmes
- Le (sous-)programme qui utilise un sous-programme est appelé **(sous-)programme appelant**

Règle de nommage

- Nous savons maintenant que les variables, les constantes, les types définis par l'utilisateur (comme les énumérateurs) et que les sous-programmes possèdent un nom
- Ces noms doivent suivre certaines règles :
- Ils doivent être explicites (à part quelques cas particuliers, comme par exemple les variables i et j pour les boucles)
- Ils ne peuvent contenir que des lettres et des chiffres
- Ils commencent obligatoirement par une lettre
- Les variables et les sous-programmes commencent toujours par une minuscule
- Les types commencent toujours par une majuscule
- Les constantes ne sont composées que de majuscules
- Lorsqu'ils sont composés de plusieurs mots, on utilise les majuscules (sauf pour les constantes) pour séparer les mots (par exemple JourDeLaSemaine)

Les différents types de variable

- Définitions :

- La **portée** d'une variable est l'ensemble des sous-programmes où cette variable est connue (les instructions de ces sous-programmes peuvent utiliser cette variable)

- Une variable définie au niveau du programme principal (celui qui résout le problème initial, le problème de plus haut niveau) est appelée **variable globale**

- Sa portée est totale : **tout** sous-programme du programme principal peut utiliser cette variable

- Une variable définie au sein d'un sous programme est appelée **variable locale**

- La portée d'un variable locale est uniquement le sous-programme qui la déclare

- Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale est localement masquée

- Dans ce sous-programme la variable globale devient inaccessible

- Un programme doit suivre la structure suivante :

Programme *nom du programme*
    Définition des constantes
    Définition des types
    Déclaration des variables globales
    Définition des sous-programmes
début
    instructions du programme principal
fin

Les paramètres

- Un paramètre d'un sous-programme est une variable locale particulière qui est associée à une variable ou constante (numérique ou définie par le programmeur) du (sous-)programme appelant :
- Puisque qu'un paramètre est une variable locale, un paramètre admet un type
- Lorsque le (sous-)programme appelant appelle le sous-programme il doit indiquer la variable (ou la constante), de même type, qui est associée au paramètre
- Par exemple, si le sous-programme *sqr* permet de calculer la racine carrée d'un réel :
- Ce sous-programme admet un seul paramètre de type réel positif
- Le (sous-)programme qui utilise *sqr* doit donner le réel positif dont il veut calculer la racine carrée, cela peut être :
- une variable, par exemple *a*
- une constante, par exemple *5.25*

- Il existe trois types d'association (que l'on nomme **passage de paramètre**) entre le paramètre et la variable (ou la constante) du (sous-)programme appelant :

- Le **passage de paramètre en entrée**
- Le **passage de paramètre en sortie**
- Le **passage de paramètre en entrée/sortie**

Le passage de paramètres en entrée

- Les instructions du sous-programme ne peuvent pas modifier l'entité (variable ou constante) du (sous-)programme appelant

- En fait c'est la valeur de l'entité du (sous-) programme appelant qui est copiée dans le paramètre (à part cette copie il n'y a pas de relation entre le paramètre et l'entité du (sous-)programme appelant)

- C'est le `seul` passage de paramètre qui admet l'utilisation d'une constante

- Par exemple :

- le sous-programme *sqr* permettant de calculer la racine carrée d'un nombre admet un paramètre en entrée

- le sous-programme **écrire** qui permet d'afficher des informations admet n paramètres en entrée

- Les instructions du sous-programme affectent obligatoirement une valeur à ce paramètre (valeur qui est donc aussi affectée à la variable associée du (sous-)programme appelant)
- Il y a donc une liaison forte entre le paramètre et l'entité du (sous-) programme appelant
  - C'est pour cela qu'on ne peut pas utiliser de constante pour ce type de paramètre
- La valeur que pouvait posséder la variable associée du (sous-)programme appelant **n'est pas utilisée** par le sous-programme
- Par exemple :
  - le sous-programme **lire** qui permet de mettre dans des variables des valeurs saisies par l'utilisateur admet n paramètres en sortie

- Passage de paramètre qui combine les deux précédentes
- A utiliser lorsque le sous-programme doit utiliser et/ou modifier la valeur de la variable du (sous-)programme appelant
- Comme pour le passage de paramètre en sortie, on ne peut pas utiliser de constante
- Par exemple :
  - le sous-programme **échanger** qui permet d'échanger les valeurs de deux variables

Les fonctions

- Les fonctions sont des sous-programmes admettant des paramètres et retournant un **seul** résultat(comme les fonctions mathématiques y=f(x,y,. . .)
- les paramètres sont en nombre fixe (≥ 0)
- une fonction possède un seul type, qui est le type de la valeur retournée
- le passage de paramètre est **uniquement en entrée** : c'est pour cela qu'il n'est pas précisé
- lors de l'appel, on peut donc utiliser comme paramètre des variables, des constantes mais aussi des résultats de fonction
- la valeur de retour est spécifiée par l'instruction **retourner**
- Généralement le nom d'une fonction est soit un nom (par exemple *minimum*), soit une question (par exemple *estVide*)

- On déclare une fonction de la façon suivante :

  fonction *nom de la fonction* **(***paramètre(s) de la fonction***) :** *type de la valeur retournée*

      Déclaration *variable locale 1 : type 1 ; . . .*

  début

      instructions de la fonction avec au moins une fois l'instruction retourner

  fin

- On utilise une fonction en précisant son nom suivi des paramètres entre parenthèses

- Les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètre

Exemple de déclaration de fonction

fonction abs **(**unEntier : **Entier) : Entier**
début
    si unEntier $\geq$ 0 **alors**
        retourner unEntier
    finsi
    retourner -unEntier
fin

Remarque : Cette fonction est équivalente à :

fonction abs **(**unEntier : **Entier) : Entier**
    Déclaration tmp : **Entier**
début
    si unEntier $\geq$ 0 **alors**
        tmp $\leftarrow$ unEntier
    sinon
        tmp $\leftarrow$ -unEntier
    finsi
    retourner tmp
fin

Exemple de programme

```
Programme exemple1
    Déclaration a : Entier, b : Naturel
    fonction abs (unEntier : Entier) : Naturel
        Déclaration valeurAbsolue : Naturel
    début
        si unEntier ≥ 0 alors
            valeurAbsolue ← unEntier
        sinon
            valeurAbsolue ← -unEntier
        finsi
        retourner valeurAbsolue
    fin
début
    écrire("Entrez un entier :")
    lire(a)
    b ← abs(a)
    écrire("la valeur absolue de",a," est ",b)
fin
```

Lors de l'exécution de la fonction *abs*, la variable *a* et le paramètre *unEntier* sont associés par un passage de paramètre en entrée : La valeur de *a* est copiée dans *unEntier*

Un autre exemple

```
fonction minimum2 (a,b : Entier) : Entier
début
    si a ≥ b alors
        retourner b
    finsi
    retourner a
fin
fonction minimum3 (a,b,c : Entier) : Entier
début
    retourner minimum2(a,minimum2(b,c))
fin
```

- Les procédures sont des sous-programmes qui ne retournent **aucun** résultat
- Par contre elles admettent des paramètres avec des passages :
  - en entrée, préfixés par **Entrée** (ou **E**)
  - en sortie, préfixés par **Sortie** (ou **S**)
  - en entrée/sortie, préfixés par **Entrée/Sortie** (ou **E/S**)
- Généralement le nom d'une procédure est un verbe

- On déclare une procédure de la façon suivante :

procédure *nom de la procédure* **(** **E** *paramètre(s) en entrée ;* **S** *paramètre(s) en sortie ;* **E/S** *paramètre(s) en entrée/sortie* **)**
    Déclaration *variable(s) locale(s)*
début
    instructions de la procédure
fin

- Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses

Exemple de déclaration de procédure

procédure calculerMinMax3 **(** **E** a,b,c : **Entier** ; **S** m,M : **Entier** **)**
début
    m $\leftarrow$ minimum3(a,b,c)
    M $\leftarrow$ maximum3(a,b,c)
fin

Exemple de programme

```
Programme exemple2
    Déclaration a : Entier, b : Naturel
    procédure echanger ( E/S val1 Entier ; E/S val2 Entier ;)
        Déclaration temp : Entier
    début
        temp ← val1
        val1 ← val2
        val2 ← temp
    fin
début
    écrire("Entrez deux entiers :")
    lire(a,b)
    echanger(a,b)
    écrire("a=",a," et b = ",b)
fin
```

Autre exemple de programme

Programme *exemple3*

Déclaration entier1,entier2,entier3,min,max : **Entier**
fonction minimum2 **(**a,b : **Entier) : Entier**
. . .
fonction minimum3 **(**a,b,c : **Entier) : Entier**
. . .
procédure calculerMinMax3 **(E** a,b,c : **Entier** ; **S** min3,max3 : **Entier)**
début
min3 ← minimum3(a,b,c)
max3 ← maximum3(a,b,c)
fin

début
écrire("Entrez trois entiers :")
lire(entier1) ;
lire(entier2) ;
lire(entier3)
calculerMinMax3(entier1,entier2,entier3,min,max)
écrire("la valeur la plus petite est ",min," et la plus grande est ",max)
fin

Une fonction ou une procédure récursive est une fonction
qui s'appelle elle même.

Exemple :

fonction *factorielle* **(**n : Naturel**) :** Naturel
début
    si n = 0 **alors**
        retourner 1
    finsi
    retourner n*factorielle(n-1)
fin

Liste des appels

$$
\begin{array}{lcccc}
\text{factorielle}(4) & & 4*\text{factorielle}(3) & = & 4*3*2*1 \\
\downarrow & & \uparrow & & \\
\text{factorielle}(3) & & 3*\text{factorielle}(2) & = & 3*2*1 \\
\downarrow & & \uparrow & & \\
\text{factorielle}(2) & & 2*\text{factorielle}(1) & = & 2*1 \\
\downarrow & & \uparrow & & \\
\text{factorielle}(1) & & 1*\text{factorielle}(0) & = & 1*1 \\
\downarrow & & \uparrow & & \\
\text{factorielle}(0) & \rightarrow & 1 & &
\end{array}
$$

On peut caractériser un algorithme récursif par plusieurs propriétés :

- Le mode d'appel : direct/indirect
- Le nombre de paramètres sur lesquels porte la récursion : arité
- Le nombre d'appels récursifs : ordre de récursion
- Le genre de retour : terminal/non terminal.

Une fonction récursive s'appellant elle même a un mode d'appel **direct**(ex : factorielle). Si la récursivité est effectuée à travers plusieurs appels de fonctions différentes le mode d'appel est **indirect**.

fonction *pair* **(**n : Naturel**) :** Booléen
début
    si n = 0 **alors**
        retourner vrai
    finsi
    retourner imPair(n-1)
fin
	fonction *imPair* **(**n : Naturel**) :** Booléen
	début
	    si n = 0 **alors**
	        retourner faux
	    finsi
	    retourner pair(n-1)
	fin

- L'arité d'un algorithme est le nombre de paramètres d'entrée.
- Récursivité bien fondé :
  Une récursivité dans laquelle les paramètres de la fonction appelée sont ≪plus simple≫ que ceux de la fonction appelante. Par exemple `factorielle(n)` appelle `factorielle(n-1)`.
  Exemple de récursivité mal fondée :

  GNU : Gnu is not Unix

- L'ordre de récursion d'une fonction est le nombre d'appels récursifs lancés à chaque appel de fonction. Par exemple, factorielle(n) ne nécessite qu'un seul appel à la fonction factorielle avec comme paramètre $n - 1$. C'est donc une fonction récursive d'ordre 1.

  Par exemple, la fonction suivante basée sur la formule $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$, est d'ordre 2.

  fonction comb **(Entier** p, n**) : Entier**
  début
      si p = 0 ou n = p **alors**
          retourner 1
      sinon
          retourner comb(p, n - 1) + comb(p - 1, n - 1)
      finsi
  fin

Le genre d'un algorithme récursif est déterminé par le traitement effectué sur la valeur de retour.

- Si la valeur de retour est retrounée sans être modifié l'algorithme est dit **terminal**(Par exemple `pair/imPair` est terminal)
- Sinon l'algorithme est dit **non terminal**(ex factorielle qui multiplie la valeur de retour par n).

- La récursivité peut simplifier considérablement certains problèmes.
- Un appel de fonction/procédure à un coût non négligeable. Les programmes récursifs sont souvent plus coûteux que leurs homologues non récursifs.

Complexité

Luc Brun
luc.brun@ensicaen.fr

A partir de travaux de
Habib Abdulrab(Insa de Rouen)

Plan

- Notion de complexité
- Comment évaluer la complexité d'un algorithme
- Exemple de calculs de complexité

Notion de complexité (1)

- Comment évaluer les performances d'un algorithme
- différents algorithmes ont des coûts différents en termes de

1. temps d'exécution (nombre d'opérations effectuées par l'algorithme),
2. taille mémoire (taille nécessaire pour stocker les différentes structures de données pour l'exécution).

Ces deux concepts sont appelé la complexité en temps et en espace de l'algorithme.

Notion de complexité (2)

- La complexité algorithmique permet de mesurer les performances d'un algorithme et de le comparer avec d'autres algorithmes réalisant les même fonctionnalités.
- La complexité algorithmique est un concept fondamental pour tout informaticien, elle permet de déterminer si :
- un algorithme *a* et meilleur qu'un algorithme *b* et
- s'il est *optimal* ou
- s'il ne doit pas être utilisé...

Temps d'exécution (1)

Le temps d'exécution d'un programme dépend :

1. du nombre de données,
2. de la taille du code,
3. du type d'ordinateur utilisé(processeur,mémoire),
4. de la complexité en temps de l'algorithme «abstrait» sous-jacent.

Temps d'exécution (2)

Soit n la taille des données du problème et $T(n)$ le temps d'exécution de l'algorithme. On distingue :

- Le temps du plus mauvais cas $T_{max}(n)$:

  Correspond au temps maximum pris par l'algorithme pour un problème de taille n.

- Le temps moyen T_{moy} :

  Temps moyen d'exécution sur des données de taille n ($\Rightarrow$ suppositions sur la distribution des données).

$$T_{moy}(n) = \sum_{i=1}^{r} p_i T_{s_i}(n)$$

- p_i probabilité que l'instruction s_i soit exécutée,
- $T_{s_i}(n)$: temps requis pour l'exécution de s_i.

Temps d'exécution

Règles générales :

1. le temps d'exécution (t.e.) d'une affectation ou d'un test est considéré comme constant c,
2. Le temps d'une séquence d'instructions est la somme des $t.e.$ des instructions qui la composent,
3. le temps d'un branchement conditionnel est égal au t.e. du test plus le max des deux t.e. correspondant aux deux alternatives (dans le cas d'un temps max).
4. Le temps d'une boucle est égal à la somme du coût du test + du corps de la boucle + test de sortie de boucle.

- Soit l'algorithme suivant :

Nom: Calcul d'une somme de carrés
Role: Calculer la valeur moyenne d'un tableau
Entrée: n : entier
Sortie: somme : réel
Déclaration: i : Naturel
début
    somme $\leftarrow$ 0.0
    pour i $\leftarrow$0 **à** n-1 **faire**
        somme $\leftarrow$ somme+i*i
    finpour
fin

$$\begin{aligned} T_{moy}(n) = T_{max}(n) &= c_1 + c_1 + n(c_2 + c_3 + c_4 + c_5 + c_1) \\ &= 2c_1 + n\left(\sum_{i=1}^{5} c_i\right) \end{aligned}$$

avec c_1 : affectation, c_2 : incrémentation, c_3 test, c_4 addition, c_5 multiplication.

Trop précis $\Rightarrow$

1. Faux,
2. Inutilisable.

Notion de complexité : comportement asymptotique du t.e. :

$$T_{max}(n) = T_{moy}(n) \approx nC$$

Definition

Une fonction f est de l'ordre de g , écrit avec la notation Grand-O comme : f $= \mathcal{O}(g)$, s'il existe une constante c et un entier n_0 tels que : $f(n) \leq cg(n)$, pour tout $n \geq n_0$

selon la définition ci-dessus, on aura :

$$\begin{array}{lllll} f_1(n) = & 2n+3 & = \mathcal{O}(n^3), & 2n+3 & = \mathcal{O}(n^2), \quad 2n+3 = \mathcal{O}(n)\text{au plus jus} \\ f_2(n) = & 7n^2 & = \mathcal{O}(2^n), & 7n^2 & = \mathcal{O}(n^2) \quad \text{au plus juste} \end{array}$$

mais, $7n^2$ N'EST PAS $\mathcal{O}(n)$

Égalité de complexité

Definition

2 fonctions f et g sont d'égale complexité, ce qui s'écrit comme : O(f)= O(g) (ou $f = \theta(g)$), ssi f $= \mathcal{O}(g)$ et g $= \mathcal{O}(f)$

exemples :

- $n, 2n$, et $0,1n$ sont d'égale complexité : $\mathcal{O}(n) = \mathcal{O}(2n) = \mathcal{O}(0,1n)$
- $\mathcal{O}(n^2)$ et $\mathcal{O}(0,1n^2 + n)$ sont d'égale complexité : $\mathcal{O}(n^2) = \mathcal{O}(0,1n^2 + n)$
- par contre : $2n$ et n^3 se sont PAS d'égale complexité : $\mathcal{O}(2n) \neq \mathcal{O}(n^3)$

Definition

une fonction f est de de plus petite complexité que g, ce qui s'écrit comme : $\mathcal{O}(f) < \mathcal{O}(g)$, ssi f $= \mathcal{O}(g)$ mais g $\neq \mathcal{O}(f)$

exemples :

- $100n$ est de plus petite complexité que $0,01n^2$: $\mathcal{O}(100n) < \mathcal{O}(0,01n^2)$, $\log_2 n$ est de plus petite complexité que n : $\mathcal{O}(\log_2 n) < \mathcal{O}(n)$,

Propriétés

- Réflexivité :

$$g = \mathcal{O}(g)$$

- Transitivité :

$$\text{si } f = \mathcal{O}(g) \text{ et } g = \mathcal{O}(h) \text{ alors } f = \mathcal{O}(h)$$

- Produit par un scalaire :$\mathcal{O}(\lambda f) = \mathcal{O}(f), \lambda > 0$.
- Somme et produit de fonctions :
- $\mathcal{O}(f)$+$\mathcal{O}(g)$=$\mathcal{O}(max\{f, g\})$
- $\mathcal{O}(f)$.$\mathcal{O}(g)$=$\mathcal{O}(f.g)$

Complexité d'une boucle

pour i ←1 **à** n **faire**
    s
finpour
avec s=$\mathcal{O}(1)$.

- Temps de calcul de s : $T_s = C$
- Nombre d'appels de s : n
- Temps de calcul total : $T(n) = nT_s = \mathcal{O}(n)$
- Complexité : $\mathcal{O}(n)$

Théorème

La complexité de p boucles imbriquées de 1 à n ne contenant que des instructions élémentaires est en $\mathcal{O}(n^p)$.

Preuve:

- vrai pour $p = 1$,
- supposons la ppt vrai à l'ordre p. Soit :

  pour i $\leftarrow$1 **à** n **faire**
      instruction
  finpour

  où **instruction** contient p boucles imbriquées.
- Soit $T_{inst}(n)$ le temps d'exécution de **instruction** et $T(n)$ le temps total.
- $T(n) = nT_{inst}(n)$ avec $T_{inst}(n) \leq Cn^p$ pour $n \geq n_0$ (par hypothèse).
- $T(n) \leq Cn^{p+1} \Rightarrow T(n) = \mathcal{O}(n^{p+1})$

Boucles tant que

h $\leftarrow$ 1
tant que h $\leq$ n **faire**
    h $\leftarrow$ 2*h
fintantque

- Test, multiplication, affectation : $\mathcal{O}(1)$: $T = C$
- Nombre d'itérations : $\log_2(n)$.
- Temps de calcul : $T(n) = C\log_2(n) = \mathcal{O}(log_2(n))$

Complexité d'un algorithme récursif (1)

Soit l'algorithme :

fonction *factorielle* **(**n : Naturel**) :** Naturel
début
    si n = 0 **alors**
        retourner 1
    sinon
        retourner n*factorielle(n-1)
    finsi
fin

Complexité d'un algorithme récursif (2)

c_1 test, c_2 multiplication.

- $T(0)=c_1$
- $T(n)=c_1+c_2+T(n-1)$

$$\Rightarrow T(n) = nc_2 + (n+1)c_1$$

Soit $C = 2max\{c_1, c_2\}$

$$T(n) \leq Cn + c_1 = \mathcal{O}(n)$$

Complexité en $\mathcal{O}(n)$.

Les calculs de complexité d'algorithmes récursifs induisent naturellement des suites.

Algorithmes récursifs en $\mathcal{O}(\log_2(n))$ (1)

Théorème

Soient, debut, fin et $n = fin - debut$ trois entiers. La complexité de l'algorithme ci dessous est en $\mathcal{O}(\log_2(n))$.

procédure *f* (*debut,fin*)
    Déclaration Entier*milieu*
début
    milieu $\leftarrow \frac{debut+fin}{2}$
    si *fin-milieu* $>$ *1* et *milieu-debut* $>$ *1* alors
        s
    sinon
        si *test* alors
            f (debut,milieu)
        sinon
            f (milieu,fin)
        finsi
    finsi
fin

- Tests, s : $\mathcal{O}(1)$

$$
\begin{array}{rcl}
T(n) & = & T(\frac{n}{2}) + C \\
T(\frac{n}{2}) & = & T(\frac{n}{4}) + C \\
\vdots & = & \vdots \\
T(1) & = & T(0) + C \\
\hline
T(n) & = & T(0) + Clog_2(n)
\end{array}
$$

Algorithmes récursifs en $\mathcal{O}(n \log_2(n))$ (1)

procédure f **(**debut,fin**)**
    Déclaration Entier milieu
début
    milieu $\leftarrow \frac{debut+fin}{2}$
    si fin-milieu $>$ 1 et milieu-debut $>$ 1 **alors**
        s_1
    sinon
        pour i $\leftarrow$1 **à** n **faire**
            s_2
        finpour
        f(début,milieu)
        f(milieu,fin)
    finsi
fin

Algorithmes récursifs en $\mathcal{O}(n\log_2(n))$ (2)

- Tests, s : $\mathcal{O}(1)$
- Boucle : $\mathcal{O}(n)$.

$$
\begin{array}{lcl}
T(n) & = & n + 2T(\frac{n}{2}) \\
2T(\frac{n}{2}) & = & n + 4T(\frac{n}{4}) \\
\vdots & = & \vdots \\
2^{p-1}T(2) & = & n + 2^p T(1) \\
\hline
T(n) & = & n * p + 2^p * T(1)
\end{array}
$$

avec $p = [\log_2(n)]$.

On a de plus :

$$\mathcal{O}(n\log_2(n) + nT(1)) = \mathcal{O}(n\log_2(n))$$

Principales complexités

- $\mathcal{O}(1)$: temps constant,
- $\mathcal{O}(\log(n))$: complexité logarithmique *(Classe L)*,
- $\mathcal{O}(n)$: complexité linaire *(Classe P)*,
- $\mathcal{O}(n\log(n))$
- $\mathcal{O}(n^2)$,$\mathcal{O}(n^3)$,$\mathcal{O}(n^p)$: quadratique, cubique,polynomiale *(Classe P)*,
- $\mathcal{O}(p^n)$ complexité exponentielle *(Classe EXPTIME)*,,.
- $\mathcal{O}(n!)$ complexité factorielle.

Influence de la complexité

$n \to 2n$.

$\mathcal{O}(1)$	$\mathcal{O}(\log_2(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n\log_2(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$
t	$t+1$	$2t$	$2t+2n$	$4t$	$8t$	t^2

	1	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	2^n
$n=10^2$	$1\mu s$	$6\mu s$	$0.1ms$	$0.6ms$	$10ms$	$1s$	$4\times10^{16}a$
$n=10^3$	$1\mu s$	$10\mu s$	$1ms$	$10ms$	$1s$	$16.6min$	∞
$n=10^4$	$1\mu s$	$13\mu s$	$10ms$	$0.1s$	$100s$	$11,5j$	∞
$n=10^5$	$1\mu s$	$17\mu s$	$0.1s$	$1.6s$	$2.7h$	$32a$	∞
$n=10^6$	$1\mu s$	$20\mu s$	$1s$	$19.9s$	$11,5j$	32×10^3a	∞

- La complexité est un résultat asymptotique : un algorithme en $\mathcal{O}(Cn^2)$ peut être plus efficace qu'un algorithme en $\mathcal{O}(C'n)$ pour de petites valeurs de n si $C \ll C'$,
- Les traitements ne sont pas toujours linéaires $\Rightarrow$ Il ne faut pas supposer d'ordres de grandeur entre les différentes constantes.

Conseils

Qualités d'un algorithme :

1. Maintenable (facile à comprendre, coder, déboguer),
2. Rapide

Conseils :

1. Privilégier le point 2 sur le point 1 uniquement si on gagne en complexité.
2. ≪ce que fait≫ l'algorithme doit se lire lors d'une lecture rapide : Une idée par ligne. Indenter le programme.
3. Faire également attention à la précision, la stabilité et la sécurité.

La rapidité d'un algorithme est un élément d'un tout définissant les qualités de celui-ci.

Les tableaux

Nicolas Delestre et Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr
Adapté pour l'ENSICAEN par
Luc Brun
luc.brun@ensicaen.fr

Plan

- Pourquoi les tableaux ?
- Les tableaux à une dimension
- Les tableaux à deux dimensions
- Les tableaux à n dimensions

Pourquoi les tableaux ?

- Imaginons que l'on veuille calculer la moyenne des notes d'une promotion, quel algorithme allons nous utiliser ?
- Pour l'instant on pourrait avoir l'algorithme suivant :

Nom: moyenne
Role: Affichage de la moyenne des notes d'une promo saisies par le prof
Entrée: -
Sortie: -
Déclaration: somme, nbEleves, uneNote, i : Naturel
début

```
    somme ← 0.0
    écrire(Nombre d'élèves :)
    lire(nbEleves)
    pour i ←0 à nbEleves faire
        écrire("Note de l'élève numéro",i," :")
        lire(note)
        somme ← somme+note
    finpour
```

fin
écrire("Moyenne des notes :",somme/nbEleves)

Pourquoi les tableaux ?

- Imaginons que l'on veuille toujours calculer la moyenne des notes d'une promotion mais en gardant en mémoire toutes les notes des étudiants (pour par exemple faire d'autres calculs tels que l'écart type, la note minimale, la note maximale, etc.)

- Il faudrait alors déclarer autant de variables qu'il y a d'étudiants, par exemple en supposant qu'il y ait 3 étudiants, on aurait l'algorithme suivant :

procédure moyenne **()**
    Déclaration somme, note1, note2, note3 : Naturel
début
    écrire("Les notes des trois étudiants :")
    lire(note1) ;
    lire(note2) ;
    lire(note3) ;
    somme $\leftarrow$ note1+note2+note3
    écrire("La moyenne est de :",somme/3)

Pourquoi les tableaux ?

- Le problème est que cet algorithme ne fonctionne que pour 3 étudiants

- Si on en a 10, il faut déclarer 10 variables

- Si on en a n, il faut déclarer n variables
  … ce n'est pas réaliste

- Il faudrait pouvoir par l'intermédiaire d'une seule variable stocker plusieurs valeurs de même type
  … c'est le rôle des tableaux

Les tableaux à une dimension

- C'est ce que l'on nomme un type complexe (en opposition aux types simples vus précédemment)
- Le type défini par un tableau est fonction :
- du nombre d'éléments maximal que peut contenir le tableau
- du type des éléments que peut contenir le tableau
- Par exemple un tableau d'entiers de taille 10 et un tableau d'entiers de taille 20 sont `deux` types différents
- On peut utiliser directement des variables de type tableau, ou définir de nouveau type à partir du type tableau
- On utilise un type tableau via la syntaxe suivante :
- **Tableau[***intervalle***] de** *type des éléments stockés par le tableau*
  où `intervalle` est un intervalle sur un type simple **dénombrable** avec des bornes **constantes**

Par exemple :

- **Type** Notes = **Tableau[**1..26**] de** Naturel
  - défini un nouveau type appelé Notes, qui est un tableau de 26 naturels
- a : Notes, déclare une variable de type Notes
- b : **Tableau[**1..26**] de** Naturel
  - déclare une variable de type tableau de 26 Naturels
  - a et b sont de même type
- c : **Tableau[**'a'..'z'**] d'**Entier
  - déclare une variable de type tableau de 26 entiers
  - a et c sont de types différents

Les tableaux à une dimension

- Ainsi l'extrait suivant d'algorithme :
  tab : **Tableau[**'a'..'c'**] de** Réel
  tab['a']←2.5
  tab['b']←-3.0
  tab['c']←4.2

- . . . peut être présentée graphiquement par :

tab :	2.5	-3.0	4.2
	a	*b*	*c*

Les tableaux à une dimension

- On accède (en lecture ou en écriture) à la i $^{\text{ème}}$ valeur d'un tableau en utilisant la syntaxe suivante :
- *nom de la variable***[***indice***]**
- Par exemple si *tab* est un tableau de 10 entiers (tab : **Tableau[**1..10**] d'**Entier)
- tab[2] $\leftarrow$-5, met la valeur -5 dans la 2 $^{\text{ème}}$ case du tableau
- En considérant le cas où *a* est une variable de type Entier,
- a $\leftarrow$tab[2], met la valeur de la 2 $^{\text{ème}}$ case du tableau tab dans a, c'est-à-dire 5
- **lire**(tab[1]) met l'entier saisi par l'utilisateur dans la première case du tableau
- **ecrire**(tab[1]) affiche la valeur de la première case du tableau

Exemple

Nom: moyenne
Role: Affichage de la moyenne des notes d'une promo saisies par le prof
Entrée: -
Sortie: -
Déclaration: somme, nbEleves, i : Naturel, lesNotes : **Tableau[**1..100**] de** Naturel

```
début
    somme ← 0
    répéter
        écrire("Nombre d'eleves (maximum 100) :")
        lire(nbEleves)
    jusqu'à ce que nbEleves¿0 et nbEleves ≤ 100
    pour i ←1 à nbEleves faire
        écrire("Note de l'eleve numero ",i," :")
        lire(lesNotes[i])
    finpour
    pour i ←1 à nbEleves faire
        somme ← somme + lesNotes[i]
    finpour
    écrire("La moyenne est de :",somme/nbEleves)
```

Remarques

- Un tableau possède un nombre maximal d'éléments défini lors de l'écriture de l'algorithme (les bornes sont des constantes explicites, par exemple 10, ou implicites, par exemple MAX)
  - ce nombre d'éléments ne peut être fonction d'une variable
- Par défaut si aucune initialisation n'a été effectuée les cases d'un tableau possèdent des valeurs aléatoires.
- Le nombre d'éléments maximal d'un tableau est différent du nombre d'éléments significatifs dans un tableau
  - Dans l'exemple précédent le nombre maximal d'éléments est de 100 mais le nombre significatif d'éléments est référencé par la variable nbEleves
- L'accès aux éléments d'un tableau est direct (temps d'accès constant)
- Il n'y a pas conservation de l'information d'une exécution du programme à une autre

- On peut aussi avoir des tableaux à deux dimensions (permettant ainsi de représenter par exemple des matrices à deux dimensions)
- On déclare une matrice à deux dimensions de la façon suivante :
- **Tableau[***intervallePremièreDimension***][***intervalleDeuxièmeDimension***] de** *type des éléments*
- On accède (en lecture ou en écriture) à la i ème ,j ème valeur d'un tableau en utilisant la syntaxe suivante :
- *nom de la variable***[***i***][***j***]**

- Par exemple si *tab* est défini par tab : **Tableau[**1..3**][**1..2**] de** Réel)
- tab[2][1] $\leftarrow$-1.2
- met la valeur -1.2 dans la case 2,1 du tableau
- En considérant le cas où *a* est une variable de type Réel, a $\leftarrow$tab[2][1]
- met -1.2 dans a

	1	*2*
1	7.2	5.4
2	-1.2	2
3	4	-8.5

- Attention, le sens que vous donnez à chaque dimension est important et il ne faut pas en changer lors de l'utilisation du tableau

- Par exemple, le tableau tab défini de la façon suivante :
  tab : **Tableau[**1..3**][**1..2**] de** Réel
  tab[1][1]←2.0 ;tab[2][1]←-1.2 ;tab[3][1]←3.4
  tab[1][2]←2.6 ; tab[2][2]←-2.9 ; tab[3][2]←0.5
  ... peut permettre de représenter l'une des deux matrices suivantes :
  $\begin{pmatrix} 2.0 & -1.2 & 3.4 \\ 2.6 & -2.9 & 0.5 \end{pmatrix} \begin{pmatrix} 2.0 & 2.6 \\ -1.2 & -2.9 \\ 3.4 & 0.5 \end{pmatrix}$

Les tableaux à n dimensions

- Par extension, on peut aussi utiliser des tableaux à plus grande dimension
- Leur déclaration est à l'image des tableaux à deux dimensions, c'est-à-dire :
- **tableau [***intervalle1***][***intervalle2***]**. . . **[***intervallen***] de** *type des valeurs*
- Par exemple :
- tab : **tableau[**1..10**][**0..9**][**'a'..'e'**] d'**Entier
- Ainsi que leur utilisation :
- tab[2][1]['b'] ←10
- a ←tab[2][1]['b']

Les algorithmes de tri

Nicolas Delestre et Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr
Adapté pour l'ENSICAEN par
Luc Brun
luc.brun@ensicaen.fr

Plan

- Les algortihmes de tri
  - Définition d'un algorithme de tri,
  - Le tri par minimum successifs,
  - Le tri a bulles,
  - Le tri rapide.
- Les algorithmes de recherche.
  - Recherche séquentielle non triée
  - Recherche séquentielle triée,
  - Recherche dichotomique.

Définition d'un algorithme de Tri

- Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité,
- Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger en ordre croissant ou décroissant,
- Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant
- Dans ce cours on ne fera que des tris en ordre croissant
- Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur.
- Examinons tout d'abord : *le tri par minimum successif*

La procédure échanger

Tous les algorithmes de tri utilisent une procédure qui permet d'échanger (de permuter) la valeur de deux variables Dans le cas où les variables sont entières, la procédure échanger est la suivante :

procédure échanger **(**E/S a,b : Entier**)**
  Déclaration temp : Entier
début
  temp $\leftarrow$ a
  a $\leftarrow$ b
  b $\leftarrow$ temp
fin

Tri par minimum successif

- Principe
- Le tri par minimum successif est un tri par sélection :
- Pour une place donnée, on sélectionne l'élément qui doit y être positionné
- De ce fait, si on parcourt la tableau de gauche à droite, on positionne à chaque fois le plus petit élément qui se trouve dans le sous tableau droit
- Ou plus généralement : Pour trier le sous-tableau t[i..nbElements] il suffit de positionner au rang i le plus petit élément de ce sous-tableau et de trier le sous-tableau t[i+1..nbElements]

Tri par minimum successif

Par exemple, pour trier <101, 115, 30, 63, 47, 20>, on va avoir les boucles suivantes :

- i=1 <101, 115, 30, 63, 47, 20>
- i=2 <20, 115, 30, 63, 47, 101>
- i=3 <20, 30, 115, 63, 47, 101>
- i=4 <20, 30, 47, 63, 115, 101>
- i=5 <20,30, 47, 63, 115, 101>
- Donc en sortie : <20, 30, 47, 63, 101, 155>

Il nous faut donc une fonction qui pour soit capable de déterminer le plus petit élément (en fait l'indice du plus petit élément) d'un tableau à partir d'un certain rang

Fonction indiceDuMinimum

fonction indiceDuMinimum **(**t : Tableau[1..MAX] d'Entier ; rang, nbElements : Naturel**) :** Naturel

    Déclaration i, indiceCherche : Naturel

début

    indiceCherche $\leftarrow$ rang

    pour i $\leftarrow$rang+1 **à** nbElements **faire**

        si t[i]¡t[indiceCherche] **alors**

            indiceCherche $\leftarrow$ i

        finsi

    finpour

    retourner indiceCherche

fin

- L'algorithme de tri est donc :

procédure effectuerTriParMimimumSuccessif **(**E/S t : Tableau[1..MAX] d'Entier ; E nbElements : Naturel**)**
    Déclaration i,indice : Naturel
début
    pour i $\leftarrow$1 **à** nbElements-1 **faire**
        indice $\leftarrow$ indiceDuMinimum(t,i,nbElements)
        si i $\neq$ indice **alors**
            echanger(t[i],t[indice])
        finsi
    finpour
fin

- Recherche du minimum sur un tableau de taille n
  $\rightarrow$ Parcours du tableau.

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

Complexité en $\mathcal{O}(n^2)$.

- Principe de la méthode : Sélectionner le minimum du tableau en parcourant le tableau de la fin au début et en échangeant tout couple d'éléments consécutifs non ordonnés.

Tri à bulles : Exemple

Par exemple, pour trier <101, 115, 30, 63, 47, 20>, on va avoir les boucles suivantes :

- i=1 <101, 115, 30, 63, 47, 20>
  - <101, 115, 30, 63, 20, 47>
  - <101, 115, 30, 20, 63, 47>
  - <101, 115, 20, 30, 63, 47>
  - <101, 20, 115, 30, 63, 47>
- i=2 <20, 101, 115, 30, 63, 47>
- i=3 <20, 30,101, 115, 47, 63>
- i=4 <20, 30,47,101, 115, 63>
- i=4 <20, 30, 47, 63, 101, 115>
- Donc en sortie : <20, 30, 47, 63, 101, 155>

Tri à bulles : l'algorithme

```
procédure TriBulles (E/S t : Tableau[1..MAX] d'Entiers,nbElements :
Naturel)
   Déclaration i,k : Naturel
début
   pour i ←0 à nbElements-1 faire
      pour k ←nbElements-1 à i+1 faire
         si t[k]<t[k-1] alors
            echanger(t[k],t[k-1])
         finsi
      finpour
   finpour
fin
```

- Nombre de tests(moyenne et pire des cas) :

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

Compléxité en $\mathcal{O}(n^2)$.

- Nombre d'échanges (pire des cas) :

$$E(n) = n - 1 + n - 2 + \cdots + 1 \rightarrow \mathcal{O}(n^2)$$

- Nombre d'échange (en moyenne)$\mathcal{O}(n^2)$ (calcul plus compliqué)

En résumé : complexité en $\mathcal{O}(n^2)$.

Le tri rapide

- Principe de la méthode
- Choisir un élément du tableau appelé *pivot*,
- Ordonner les éléments du tableau par rapport au pivot
- Appeler récursivement le tri sur les parties du tableau
- à gauche et
- à droite du pivot.

La partition

procédure partition **(**E/S t : Tableau[1..MAX] d'**Entier** ; E :premier, dernier : **Naturel**, S : indPivot : **Naturel)**

```
    Déclaration compteur, i : Naturel,pivot : Entier
début
    compteur ← premier
    pivot ← t[premier]
    pour i ←premier+1 à dernier faire
        si t[i] < pivot alors
            compteur ← compteur+1
            echange(t[i],t[compteur]) ;
        finsi
    finpour
    echanger(T,compteur,premier)
    indPivot ← compteur
fin
```

Exemple de partition

$6^{(c)}$	$3^{(i)}$	0	9	1	7	8	2	5	4
6	$3^{(i,c)}$	0	9	1	7	8	2	5	4
6	3	$0^{(i,c)}$	9	1	7	8	2	5	4
6	3	$0^{(c)}$	$9^{(i)}$	1	7	8	2	5	4
6	3	$0^{(c)}$	9	$1^{(i)}$	7	8	2	5	4
6	3	0	$\color{blue}{1^{(c)}}$	$\color{blue}{9^{(i)}}$	7	8	2	5	4
6	3	0	$1^{(c)}$	9	7	8	$2^{(i)}$	5	4
6	3	0	1	$2^{(c)}$	7	8	$9^{(i)}$	5	4
6	3	0	1	2	$5^{(c)}$	8	9	$7^{(i)}$	4
6	3	0	1	2	5	$4^{(c)}$	9	7	$8^{(i)}$

4	3	0	1	2	5	$6(c)$	9	7	8

Le tri rapide

- Algorithme :

procédure triRapide **(**E/S t : Tableau[1..MAX] d'**Entier** ; gauche,droit : **Naturel)**

    Déclaration pivot : **Naturel**
début
    si gauche$<$droite **alors**
        partition(t,gauche,droite,pivot)
        triRapide(t,gauche,pivot-1)
        triRapide(t,pivot+1,droite)
    finsi
fin

Exemple

Dans l'exemple précédent on passe de :$<6,3,0,9,1,7,8,2,5,4>$ à $<4,3,0,1,2,6,8,9,7>$ et on se relance sur :

- $<4,3,0,1,2>$ et
- $<8,9,7>$

- Le tri par rapport au pivot nécessite de parcourir le tableau. On relance ensuite le processus sur les deux sous tableaux à gauche et à droite du pivot.

$$T(n) = n + T(p) + T(q)$$

p, q taille des sous tableaux gauche et droits.
Dans le meilleur des cas $p = q$ et :

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

Posons $n = 2^p$. On obtient :

$$\begin{array}{rcl} T(p) & = & 2^p + 2T(p-1) \\ & = & p2^p + 2^p \end{array}$$

En repassant en n : $T(n) = \log_2(n).n + n$. La complexité est donc en $\mathcal{O}(n \log_2(n))$ (dans le meilleur des cas).

- Recherche dans un tableau non trié.

fonction rechercheNonTrie **(**tab : Tableau[0..MAX] d'Éléments, x : Élément**) :** Naturel
    Déclaration i : Naturel
début
    i $\leftarrow$ 0
    tant que (i$\leq$MAX) et (tab[i] $\neq$ x) **faire**
        i $\leftarrow$ i+1
    fintantque
    si i=MAX+1 **alors**
        retourner MAX+1
    finsi
    retourner i
fin

- Recherche séquentielle dans un tableau trié.

fonction rechercheSeqTrie **(**tab : Tableau[0..MAX+1] d'Éléments, x : Élément**) :** Naturel

    Déclaration i : Naturel
début
    si x<tab[0] **alors**
        retourner MAX + 1
    finsi
    i ← 0
    tab[MAX+1] ← x
    tant que x>tab[i] **faire**
        i ← i+1
    fintantque
    si x=tab[i] **alors**
        retourner i
    finsi
    retourner MAX + 1

fonction rechercheDicoTrie **(**tab : Tableau[0..MAX] d'Éléments, x : Élément**) :** Naturel
    Déclaration gauche,droit,milieu : Naturel
début
    gauche←0 ;droit←MAX
    tant que gauche ≤ droit **faire**
        milieu ← (gauche+droit) div 2
        si x=tab[milieu] **alors** **retourner** milieu **finsi**
        si x<tab[milieu] **alors**
            droit ← milieu-1
        sinon
            gauche ← milieu+1
        finsi
    fintantque
    retourner MAX+1
fin

Exemple

- On cherche 101 dans <20, 30, 47, 63, 101, 115>.
- i=1 <20(g), 30, 47(m), 63, 101, 115(d)>.
- i=2 <20, 30, 47, 63(g), 101(m), 115(d)>.
- et on renvoi l'indice de 101.

Types Abstraits de Données (TAD)

Nicolas Delestre et Michel Mainguenaud
{Nicolas.Delestre,Michel.Mainguenaud}@insa-rouen.fr
Adapté pour l'ENSICAEN par
Luc Brun
luc.brun@ensicaen.fr

Plan

- Définition d'un type abstrait
- Définition des enregistrements
- Bestiaire des types abstraits
- Les ensembles,
- Les listes,
- Les files,
- Les piles,
- Les arbres,
- Les arbres binaires,
- Les arbres binaires de recherche,
- Les arbres parfaits et les tas

Pourquoi les types abstraits

- Un Type abstrait de données (TAD) est :

1. un ensemble de données organisé et
2. d'opérations sur ces données.

- Il est défini d'une manière indépendante de la représentation des données en mémoire. On étudie donc le concept en termes de fonctionnalités.
- Cette notion permet l'élaboration des algorithmes :

1. en faisant appel aux données et aux opérations abstraites du TAD (couche supérieure),
2. suivi d'un choix de représentation du TAD en mémoire (couche inférieure).

Décomposition en couches

- Le codage de la couche supérieure donne des programmes abstraits, compréhensibles et réutilisables.
- Le codage de la couche inférieure implante un choix de la représentation du TAD en mémoire.

☺ La couche supérieure reste inchangée si on change cette couche inférieure.

Le codage de types abstraits dans la couche inférieure nécessite souvent des types complexes.

- En plus des types élémentaires : Entiers, Réels, Caractères..., on peut définir des nouveaux types (des types composites) grâce à la notion d'enregistrement.

- Remarque :
  La notion de Classe (beaucoup plus riche)
  dans les langages à Objets remplace avantageusement la notion d'enregistrement.

Définition d'un enregistrement

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

Type <nomEnregistrement> **= Enregistrement**
début
    <Attribut1> : <type de Attribut1> ;
    ⋮
    <Attributn> :<type de Attributn> ;
fin

Exemple

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

Type Personne **= Enregistrement**
début
    nom : ChaîneDeCharactères ;
    prénom : Chaîne ;
    age : Entier ;
fin

Enregistrements Imbriqués

- Il est possible d'imbriquer sans limitation des enregistrements les uns dans les autres.
- Exemple : On peut définir l'adresse (par un numéro, une rue, une ville et un code postal) comme un nouvel enregistrement, et l'ajouter comme un attribut supplémentaire à l'enregistrement Personne.

Exemple

Type Adresse = **Enregistrement**
début
    numero : Entier ;
    codePostal : Entier ;
    rue,ville : ChaîneDeCharactères ;
fin
Type Personne = **Enregistrement**
début
    nom,prenom : ChaîneDeCharactères ;
    age : Entier ;
    adresse : Adresse ;
fin

Accès aux attributs

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

  Var p , p1 : personne ;
  p.nom :='Dupont' ;
  p.prenom :='Jean' ;
  p.adresse.rue :='Place Colbert' ;

Les principaux types abstraits de données

- Classification suivant :

1. La présence d'un ordre parmis les éléments du TAD,
2. L'existence d'une méthode spécifique d'insertion/suppression.

On distingue notamment :

- Les ensembles : Pas de notion d'ordre.
- Les listes : Ensemble ordonné, sans notion de priorité d'entrée et de sortie.
- Les piles (FILO : First in Last Out).
- Les files (FIFO : First in First out)

Opérations :

- creer() : Ensemble.
- vide(e : Ensemble) : Booléen
- ajouter(x : Élement,e : Ensemble) : Booléen
- supprimer(x : Élément,e : Ensemble) : Booléen
- appartient(x,Élément, e : Ensemble) : Booléen

Plus en option :

- tête(e : Ensemble).
  Positionne un index sur un élément de l'ensemble,
- suivant(e : Ensemble) : Booléen.
  Choisi aléatoirement un élément non préalablement parcouru.
- courant(e : Ensemble) : Élément.
  Renvoi l'élément courant.

Différence

Calcule la différence de deux ensembles. Le paramètre de sortie diff est supposé vide.

```
procédure difference (E e1,e2 : Ensemble, S diff : Ensemble)
    Déclaration x : Element
début
    si vide(e1) alors
        retourner diff
    finsi
    tête(e1)
    répéter
        x ← courant(e1)
        si non appartient(x,e2) alors
            ajouter(x,diff)
        finsi
    jusqu'à ce que  non suivant(e1)
fin
```

Un exemple de codage d'ensemble

- Un tableau de taille MAX,
- un index sur l'élément courant,
- un index sur le dernier élément.

Type Ensemble = **Enregistrement**
début
    courant : Naturel ;
    dernier : Entier ;
    élément : Tableau[0..MAX] d'Éléments ;
fin

Un début de codage (1)

fonction créer **() :** Ensemble
  Déclaration ens : Ensemble
début
  ens.courant $\leftarrow$ 0
  ens.dernier $\leftarrow$ -1
  retourner ens
fin
fonction vide **(**e : Ensemble**) :** Booléen
  Déclaration -
début
  retourner e.dernier$=$-1
fin

Un début de codage (2)

```
fonction ajouter (x : Élément,e : Ensemble) : Booléen
   Déclaration -
début
   si e.dernier = MAX alors
      retourner faux
   finsi
   e.dernier ← e.dernier+1
   élément[e.dernier] ← x
   retourner vrai
fin
```

Un début de codage (3)

```
fonction supprimer (x : Élément, e : Ensemble) : Booléen
   Déclaration i : Naturel
début
   si vide(e) alors
      retourner faux
   finsi
   i ← 0
   tant que i ¡ e.dernier faire
      si e.élément[i]=x alors
         si i≠e.dernier alors
            echanger(e.élement[i],e.element[e.dernier])
         finsi
         e.dernier ← e.dernier-1
      sinon
         i ← i+1
      finsi
   fintantque
```

Les listes

Un ensemble ordonné d'éléments (notion de premier, second...).

Opérations :

- créer() : Liste,
- vide(l : Liste) : Booléen,
- ajouter(x : Élément, l : Liste) : Booléen (à préciser).
- supprimer(x : Élément, l : liste) : Booléen

Opérations de parcourt :

- tête(l : Liste)
- courant(l : Liste) : Élément
- suivant(l : Liste) : Booléen (faux si vide)
- ajouterCourant(x : Élément, l : Liste)
- pré conditions :
  - tête(l), courant(l), suivant(l) définis ssi vide(l)=faux

Exemple

```
fonction appartient (x : Élément, l : Liste) : Booléen
    Déclaration current : Élément
début
    si vide(l) alors
        retourner faux
    finsi
    tête(l)
    répéter
        current ← courant(l)
        si current=x alors
            retourner vrai
        finsi
    jusqu'à ce que non suivant(l)
    retourner faux
fin
```

Représentation par tableau

Identique à celle vue pour les ensembles.

Type Ensemble = **Enregistrement**
début
    courant : Naturel ;
    dernier : Entier ;
    élément : Tableau[0..MAX] d'Éléments ;
fin

La différence apparaît dans les opérations qui doivent maintenir l'ordre des éléments dans la liste.

Représentation par tableau

- Avantages :
  - Parcours et accès faciles au i[e]élément (accès direct).
  - Possibilité de recherche efficace si la liste est triée (par exemple, recherche dichotomique).
- Inconvénients :
  - Réservation, lors de la compilation de la taille maximale.

    Manque de souplesse et d'économie.
  - Inefficacité de la suppression et de l'insertion pour les listes :

    Obligation de décaler tous les éléments entre l'élément inséré ou supprimé et le dernier élément.

Les Pointeurs

- Une variable de type pointeur est une variable qui contient une adresse.
- Syntaxe (en pseudo langage algorithmique) :
- nom : $^\wedge$¡type¿ ;

  nom ici est une variable de type ¡type¿
  Exemple :

  x : $^\wedge$Entier
- nom$^\wedge$ désigne la valeur (de type ¡type¿) sur laquelle pointe la variable nom.
- NIL est le pointeur vide : une variable x de type pointeur initialisée à NIL signifie que x ne pointe sur rien.
- allouer(nom, ¡type¿)

  permet d'allouer dynamiquement de l'espace mémoire (de type ¡type¿) et fait pointer nom sur elle.
- libérer(nom)

  permet de libérer l'espace mémoire alloué ci-dessus.

Illustration

x : ^Entier
allouer(x,Entier) ;
x^ $\leftarrow$ 1

	1	
x$\rightarrow$	2	4
	3	
x^$\rightarrow$	4	1

Soit l'enregistrement :

Type Personne **= Enregistrement**

début

nom : ChaîneDeCharactères ;

prénom : Chaîne ;

age : Entier ;

fin

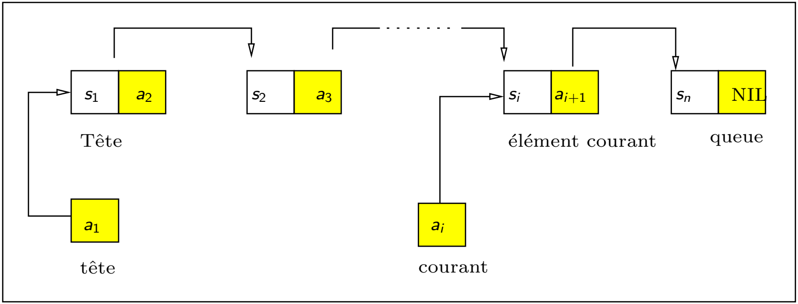
- Un pointeur sur enregistrement se défini par :

x : ^Personne

- On accède à un attribut comme suit :

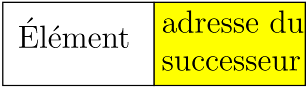
x^.age ← 10

Représentation de listes par cellules



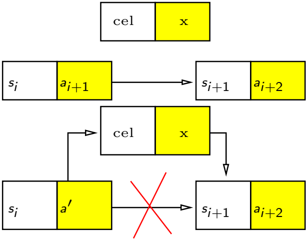
Codage par cellules

Type PointeurCellule = ^Cellule
Type Cellule = **Enregistrement**
début
    contenu : Élément ;
    suivant : PointeurCellule
fin
Type Liste = **Enregistrement**
début
    tête : PointeurCellule ;
    courant : PointeurCellule
fin



Liste vide, Ajout d'une cellule

procédure ajouter **(**E : x Élément, E/S l : Liste**)**
    Déclaration cel : PointeurCellule
début
    allouer(cel,cellule)
    cel^.contenu ← x
    si non vide(l) **alors**
        cel^.suivant ← l.courant^.suivant
        l.courant^.suivant ← cell
    sinon
        cel^.suivant ← NIL
        l.courant ← cell
        l.tete ← cell
    finsi
fin



fonction vide **(**l : Liste**)** **:** Booléen
    Déclaration -
début
    retourner l.tete=NIL
fin

Suppression d'une cellule

procédure supprimer **(**E/S l : Liste**)**

  Déclaration cell : PointeurCellule

début

  si vide(l) **alors**

    retourner

  finsi

  cell ← l.courant.suivant

  si cell=NIL **alors**

    si l.courant=l.tête **alors**

      libérer(l.tete)

      l.tete ← NIL

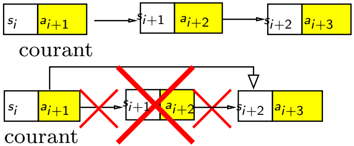
      l.courant ← NIL

    finsi

  sinon

    l.courant.suivant ← cell.suivant

    libérer(cell)



Représentation des listes par cellules

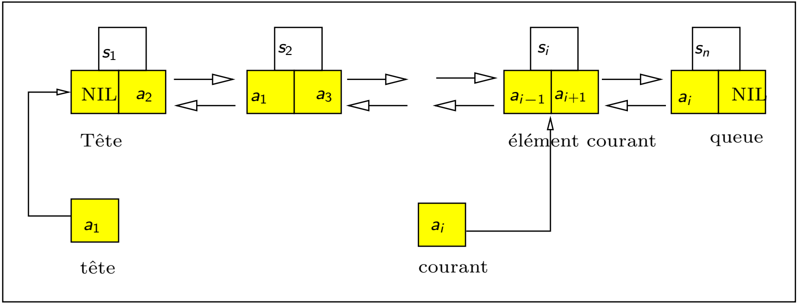
- Avantages :
  - La quantité de mémoire

    utilisée est exactement ajustée aux besoins.
  - Insertion et suppression

    plus aisées et efficaces que dans le cas de la représentation par des tableaux.
- Inconvénients :
  - Accès uniquement séquentiel.

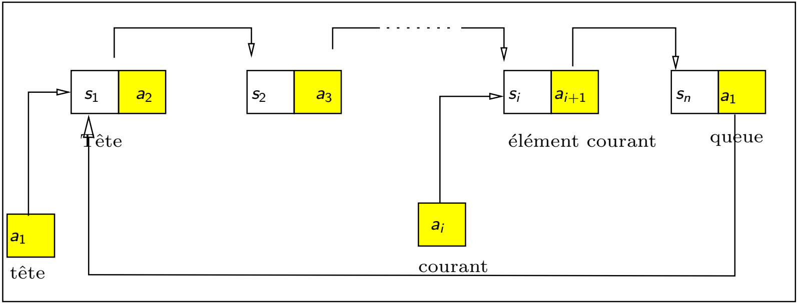
- Les listes doublement chaînées.

Type Cellule = **Enregistrement**
début
    contenu : Élément ;
    suivant : PointeurCellule
    précédent : PointeurCellule
fin



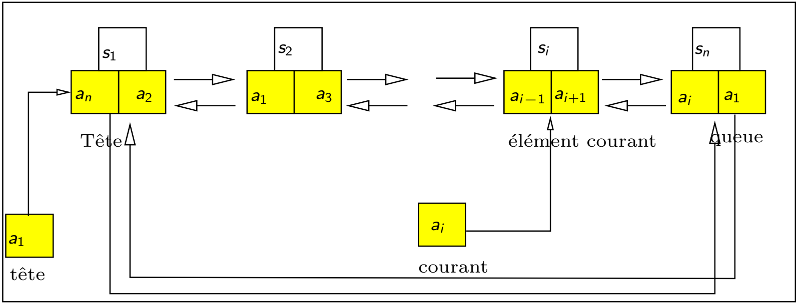
- Les listes circulaires :Le pointeur de queue pointe sur la tête.

- Avantages : Gestion plus aisée des suppressions, insertions.



Différents types de listes

- Liste doublements chaînées circulaires.



- Correspondent comme les liste à une suite d'éléments ordonnés.
- Différences :
  - La notion d'élément courant n'existe pas.
  - On ajoute et enlève les éléments de la structure dans un ordre précis.

- Structure de type FIFO (files d'attentes).
- On utilise généralement la notion de file lorsque l'on à un mécanisme d'attente où le premier arrivé est le plus prioritaire (supermarché, imprimante. . .)

$\leftarrow$	10	20	30	$\leftarrow$

Files : Opérations

- Opérations :
  - créer() : File,
  - vide(f : file) : Booléen,
  - enfiler(x : Élément,f : File),
  - défiler(f :File),
  - tête(f : File) : Élément,
- Pré conditions :
  - défiler(f) est défini ssi vide(f)=faux
  - tête(f) est défini ssi vide(f)=faux

Type File **= Enregistrement**
début
  début : Naturel ;
  nbÉléments :Naturel ;
  tab : Tableau[1..MAX] d'Éléments
fin
fonction créer **() :** File
  Déclaration f : File
début
  f.debut ← 0
  nbÉléments ← 0
fin

```
fonction plein (f : File) : Booléen
   Déclaration -
début
   retourner  nbÉléments=MAX
fin
procédure enfiler (x : Élément, f : File)
   Déclaration fin : Naturel
début
   si plein(f) alors
      erreur("File pleine")
   finsi
   fin ← (f.début+nbÉléments) mod MAX
   f.tab[fin] ← x
   nbÉléments ← nbÉléments+1
fin
```

```
fonction vide (f : File) : Booléen
   Déclaration -
début
   retourner nbÉléments=0
fin
procédure défiler (f : File)
   Déclaration -
début
   si vide(d) alors
      erreur("File Vide")
   finsi
   f.debut ← (f.debut+1) mod MAX
   nbÉléments ← nbÉléments-1
fin
```

Exemple

-créer()

début	:	0
nbÉlt	:	0

0	1	2	3	4

-enfiler(10,20,30,40,50)

début	:	0
nbÉlt	:	5
fin	:	0

0	1	2	3	4
10	20	30	40	50

-défiler()

début	:	1
nbÉlt	:	4
fin	:	0

0	1	2	3	4
	20	30	40	50

-enfiler(60)

début	:	1
fin	:	1

0	1	2	3	4
60	20	30	40	50

```
Type File = Enregistrement
début
    début :PointeurCellule ;
    fin : PointeurCellule ;
fin
fonction créer () : File
    Déclaration f : File
début
    f.début ← NIL
    f.fin ← NIL
fin
```

Enfiler

```
procédure enfiler (x : Élément, f : File)
    Déclaration cell : PointeurCellule
début
    allouer(cell,Cellule)
    cell^.contenu ← x
    cell^.suivant ← NIL
    si vide(f) alors
        f.debut ← cell
    sinon
        f.fin.suivant ← cell
    finsi
    f.fin ← cell
fin
```

Vide, défiler

```
procédure défiler (f : File)
    Déclaration cell : PointeurCellule
début
    si vide(f) alors
        erreur("File Vide")
    finsi
    si f.debut=f.fin alors
        libérer(f.debut)
        f.debut ← NIL
        f.fin ← NIL
    sinon
        cell ← f.début
        f.début ← f.debut.suivant
        liberer(cell)
    finsi
fin
fonction vide (f : File) :
Booléen
```

Les Files circulaires

- Permet de se passer du pointeur de début.
- Le début de la liste est la cellule suivante celle pointé par fin.

Les Piles

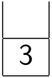
- Structure de type LIFO (Last In, First Out).
- Dans une pile l'information importante est la dernière entrée.
- Permet de mémoriser :
  - La dernière action effectuée,
  - La dernière tache en cours...

Penser à une pile de papier, de dossier...

Pile : opérations

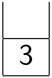
- Opérations
  - créer() : Pile,
  - empiler(x : Élément, p : Pile),
  - dépiler(p : Pile),
  - sommet(p : Pile) :Élément,
  - vide(p : Pile) : Booléen.
- Pré conditions :
  - dépiler(p) défini ssi : vide(p)=faux,
  - sommet(p) défini ssi : vide(p)=faux,

Exemple

- p=creer() ; 
- empiler(3,p) ; 

- empiler(4,p) ; 

- x ←tête(p) 

→ x=4.
- dépiler(p) 

- dépiler(p) 

Type Pile **= Enregistrement**
début
    nbÉléments : Naturel
    tab : Tableau[0..MAX] de Naturel
fin
fonction créer **() :** Pile
    Déclaration p : Pile
début
    p.nbÉléments $\leftarrow$ 0
fin

```
fonction vide (p : Pile) : Booléen
début
    retourner p.nbÉléments=0
fin
fonction sommet (E p : Pile) : Élément
    Déclaration -
début
    si vide(p) alors
        erreur("Pile vide")
    finsi
    retourner   p.tab[p.nbÉléments-1]
fin
```

Empiler, dépiler

```
procédure empiler (E x : Élément,E/S p : Pile)
   Déclaration -
début
   si p.nbÉléments=MAX+1 alors
      erreur("Pile pleine")
   finsi
   p.tab[p.nbÉléments] ← x
   p.nbÉléments ← p.nbÉléments+1
fin
procédure dépiler (E/S p : Pile)
   Déclaration -
début
   si vide(p) alors
      erreur("Pile Vide")
   finsi
   p.nbÉléments ← p.nbÉléments-1
fin
```

Représentation par liste (chaînée)

Type Pile = **Enregistrement**
début
    tête : PointeurCellule
fin
fonction créer **() :** Pile
    Déclaration p : Pile
début
    p.tête $\leftarrow$ NIL
fin
fonction vide **(**p :Pile**) :** Booléen
    Déclaration -
début
    retourner p.tête=NIL
fin

Sommet, Empiler

```
fonction sommet (p : pile) : Élément
    Déclaration -
début
    si vide(p) alors
        erreur("Pile Vide")
    finsi
    retourner p.tête^.contenu
fin
procédure empiler (E x : Élément,E/S p : Pile)
    Déclaration cell : PointeurCellule
début
    allouer(cell,Cellule)
    cell.contenu ← x
    cell.suivant ←  p.tête
    p.tête ←  cell
fin
```

Dépiler

```
procédure dépiler (E/S p : Pile)
    Déclaration cell : PointeurCellule
début
    si vide(p) alors
        erreur("Pile Vide")
    finsi
    cell ← p.tête
    p.tête ← p.tête^.suivant
    libérer(cell)
fin
```

Les graphes

- Un graphe $G = (V, E)$ est défini par :
  - Un ensemble de sommets V,
  - un ensemble d'arêtes E.
- Exemples de graphes :
  - (villes, routes),
  - (machines, réseaux),
  - (Personnes, relations)

$$\vdots$$

Quelques définitions

- Un chemin :
  - Suite $(v_1, \ldots, v_n)$ de sommets tels que :

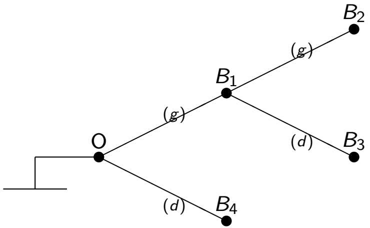
$$\forall i \in \{1, \ldots, n-1\}\ (v_i, v_{i+1}) \in E$$

- Un chemin simple :
  - chaque v_i apparaît une seule fois.
- Un cycle :
  - Un chemin fermé $(v_1 = v_n)$
- Un graphe connexe :
  - Un graphe tel que tout couple de sommets peut être relié par un chemin.

Arbre

- Un arbre est un graphe G,
  simple non orienté satisfaisant une des condition (équivalentes) suivante :

  - G est connexe sans cycle (si non connexe on parle de forêt i.e. un ensemble d'arbres),
  - G est sans cycle et l'ajout d'une arête quelconque crée un cycle,
  - G est connexe et n'est plus connecté si l'on retire n'importe qu'elle arête,
  - Tout couples de sommets de G sont reliés par un chemin unique.



Arbres enraciné

- Arbre enraciné :

  arbre dans lequel on distingue un sommet particulier appelé *racine* (induit une orientation naturelle des arêtes).

- Soient n et n' deux sommets adjacents.
- n appartient au chemin de n' à la racine
  $\Rightarrow$ n est le père de n' & n' le fils de n ou
- n' appartient au chemin de n à la racine
  $\Rightarrow$ n' est le père de n & n le fils de n'.
- Un noeud possédant au moins un fils est appelé un *noeud interne*
- Un noeud ne possédant pas de fils est appelé une *feuille*.
- n_i est un *ancêtre* de n_j si il existe p tel que $\text{père}^{(p)}(n_j) = n_i$. La racine est l'ancêtre commun de tous les noeuds.
- n_j est un *descendant* de n_i si n_i est un ancêtre de n_j

Type PointeurNoeud = $^\wedge$Arbre
Type Arbre = **Enregistrement**
début
    contenu : Élément ;
    fils : Liste de PointeurNoeud ;
fin

- Liste de PointeurNoeud : liste où le champ contenu est un pointeur sur un noeud.

- La *taille* d'un arbre est son nombre de noeuds.
- taille(arbre vide)=0

$$taille(n) = 1 + \sum_{n' \in Fils(n)} taille(n')$$

- La *hauteur*(profondeur ou niveau) d'un noeud n est le nombre de noeuds entre celui-ci et la racine.
- hauteur(n)=0 si n racine, sinon :

$$hauteur(n) = 1 + hauteur(\text{père}(n))$$

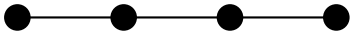
- La hauteur (ou profondeur) d'un arbre est la profondeur maximum de ses noeuds.

$$hauteur(B) = max_{n \in B} hauteur(n)$$

- Tout noeud d'un *arbre binaire* a au plus 2 fils.
- $B = (O, B_1, B_2)$,
  - O : racine,
  - B_1 sous arbre gauche,
  - B_2 sous arbre droit de B.
- *Fils gauche*(resp. droit) : racine du sous arbre gauche (resp. droit)
  - fils_gauche(O)=a
  - fils_droit(O)=d

Arbres binaires particuliers

- Un arbre binaire *dégénéré* ou *filiforme* est un arbre formé de noeuds n'ayant qu'un seul fils.

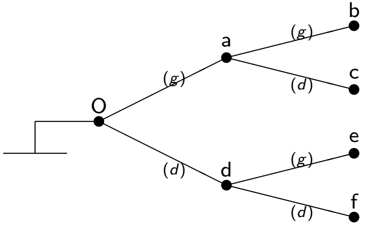


- Un arbre binaire est dit *complet*

  si toutes les feuilles ont un même niveau h et si chaque sommet interne a exactement deux fils. Le nombre de noeuds au niveau h est donc égal à 2^h. Le nombre total de noeuds est égal à :

$$\sum_{i=0}^{h} 2^i = 2^{i+1} - 1$$

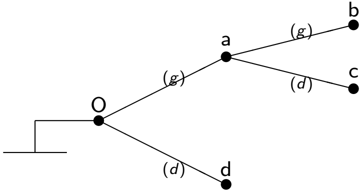
  avec h hauteur de l'arbre



Un arbre complet

- Un arbre binaire de niveau h est parfait

  si l'ensemble des noeuds de niveau $h-1$ forme un arbre binaire complet et si les noeuds de niveau h sont situé le plus à gauche possible.

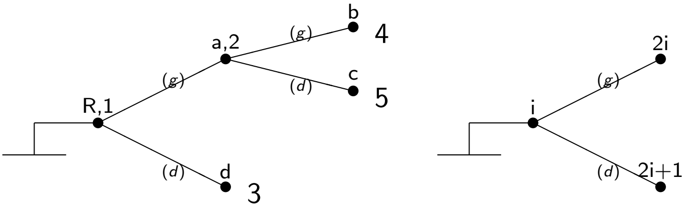


Un arbre parfait.

- Un arbre complet est parfait. On parle d'arbre parfait complet et incomplet.

Numérotation hiérarchique

- On numérote tous les noeuds à partir de la racine, en largeur d'abord de gauche à droite.



- Codage implicite de la structure.

1	2	3	4	5
R	a	d	b	c

Encadrements de la hauteur d'un arbre binaire

- Hauteurs min et max d'un arbre de n noeuds :

- hauteur max : arbre dégénéré (h=n-1).

- hauteur min : arbre parfait : ($h = \log_2(n)$)

$$\Rightarrow \log_2(n) \leq h \leq n - 1$$

Arbre binaires :Opérations-préconditions

- Opérations :
  - créer() : ArbreBinaire
  - vide(a : ArbreBinaire) : Booléen
  - contenu(a : ArbreBinaire) : Élément
  - filsG(a : ArbreBinaire) : ArbreBinaire
  - filsD(a : ArbreBinaire) : ArbreBinaire
- Pré conditions
  - contenu(a) est défini ssi vide(a)=faux
  - filsG(a) est défini ssi vide(a)=faux
  - filsD(a) est défini ssi vide(a)=faux

```
Type Fils = ^Noeud
Type Noeud = Enregistrement
début
    contenu : Élément ;
    filsGauche : Fils ;
    filsDroit : Fils ;
fin
Type ArbreBinaire = Enregistrement
début
    racine : ^Noeud ;
fin
```

Arbre binaire : créer, vide

fonction créer **() :** ArbreBinaire
  Déclaration a : ArbreBinaire
début
  a.racine $\leftarrow$ NIL
fin
fonction vide **(**a : ArbreBinaire**) :** Booléen
  Déclaration -
début
  retourner a.racine=NIL
fin

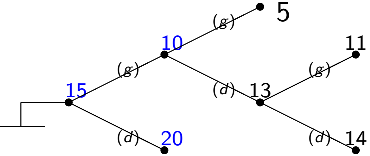
Arbre binaire : contenu, filsG

fonction contenu **(**a : ArbreBinaire**) :** Élément
  Déclaration -
début
  retourner a.racine$^\wedge$.contenu
fin
fonction filsG **(**a : ArbreBinaire**) :** ArbreBinaire
  Déclaration fils : ArbreBinaire
début
  fils.racine $\leftarrow$ a.racine$^\wedge$.filsGauche
  retourner fils
fin

- Pour tout noeud *n*

- tous les noeuds du sous arbre de gauche ont une valeur inférieure ou égale à celle de *n*,

- tous les noeuds du sous arbre de droite ont une valeur supérieure à celle de *n*,



```
fonction cherche (E x : Élément, E a : Arbre) : Booléen
   Déclaration -
début
   si a.racine=NIL alors
      retourner faux
   finsi
   si a.racine^.contenu=x alors
      retourner vrai
   finsi
   si  x< a.racine^.contenu alors
      retourner cherche(x,filsG(a))
   finsi
   retourner cherche(x,filsD(a))
fin
```

Arbres binaires de recherche : ajout feuilles

procédure ajoutFeuille **(**E x : Élément, E/S a : ArbreBinaire**)**
  Déclaration -
début
  si x$<$ a.racine$^\wedge$.contenu **alors**
    si a.racine.filsGauche=NIL **alors**
      a.racine.filsGauche $\leftarrow$ créerNoeud(x)
    sinon
      ajoutFeuille(x,a.racine.filsGauche)
    finsi
  sinon
    si a.racine.filsDroit=NIL **alors**
      a.racine.filsDroit $\leftarrow$ créerNoeud(x)
    sinon
      ajoutFeuille(x,a.racine.filsDroit)
    finsi
  finsi
fin

fonction ajoutRacine **(**E x : Élément, E a : ArbreBinaire**) :** ArbreBinaire
    Déclaration arbreMisAJour : ArbreBinaire,n : $^\wedge$Noeud
début
    allouer(n,Noeud)
    n$^\wedge$.contenu $\leftarrow$ x
    arbreMisAJour.racine $\leftarrow$ n
    couper(x,a,n$^\wedge$.filsG,n$^\wedge$.filsD)
fin

procédure couper **(**E x : Élément, E/S a : ArbreBinaire,E/S filsG,filsD : Noeud**)**
début
    si vide(a) **alors**
        retourner
    finsi
    si x>a.racine^.contenu **alors**
        filsG ← a.racine
        couper(x,filsD(a),filsG^.filsDroit,filsD)
    sinon
        filsD ← a.racine
        couper(x,filsG(a),filsG,filsD^.filsGauche)
    finsi
fin

Arbres binaires de recherche : Suppression (1)

- Si l'élément à supprimer n'existe pas on ne fait rien.
- Si l'élément à supprimer n'a pas de fils gauche, on le remplace par son fils droit.
- Si l'élément à supprimer n'a pas de fils droit, on le remplace par son fils gauche.
- Si l'élément à supprimer à deux fils, on le remplace par le plus grand (resp. petit) élément de son sous arbre gauche (resp. droit).

- Suppression du Max.

procédure supMax **(**E/S max : Élément, E/S f : Fils**)**
début
    si f^.filsDroit=NIL **alors**
        max ← f^.contenu
        f ← f^.filsGauche
    sinon
        supMax(max,f^.filsDroit)
    finsi
fin

```
procédure supprimer (E x :Élément,
E/S f : Fils)
   Déclaration max : Élément
début
   si f=NIL alors retourner finsi
   si f^.contenu=x alors
      si f.filsGauche=NIL alors
         f ← f^.filsDroit
         retourner
      finsi
      si f.filsDroit=NIL alors
         f ← f^.filsGauche
         retourner
      finsi
      suppmax( max,f^.filsGauche)
      f^.contenu ← max
   finsi
   si x<f^.contenu alors
      supprimer(x,f^.filsGauche)
   sinon
      supprimer(x,f^.filsDroit)
   finsi
fin
```

- Implémentation par tableaux.

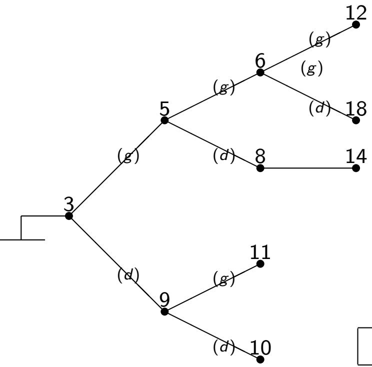
Type ArbreBinaireParfait = **Enregistrement**
début
    tab : ^Tableau[1..MAX] d'Éléments
    indice : [1..MAX]
    nbÉlémentsTotal : [1..MAX]
fin

fonction créer **() :** ArbreBinaireParfait
    Déclaration a : ArbreBinaireParfait
début
    allouer(a.tab,Tableau[1..MAX] d'Éléments)
    a.nbÉlémentsTotal $\leftarrow$ 0
    a.indice $\leftarrow$ 1
    a.racine $\leftarrow$ NIL
fin
fonction vide **(**a : ArbreBinaireParfait**) :** Booléen
    Déclaration -
début
    retourner a.nbÉlémentsTotal=0
fin

fonction contenu **(**a : ArbreBinaireParfait**) :** Élément
  Déclaration -
début
  retourner a.tab^[a.indice]
fin
fonction filsG **(**a : ArbreBinaireParfait**) :** ArbreBinaireParfait
  Déclaration fils : ArbreBinaireParfait
début
  si 2*a.indice$\geq$ a.nbÉlémentsTotal **alors**
    erreur("Pas de fils Gauche")
  finsi
  fils.tab $\leftarrow$ a.tab
  fils.indice $\leftarrow$ 2*a.indice
  fils.nbÉlémentsTotal $\leftarrow$ a.nbÉlémentsTotal
  retourner fils
fin

Les tas

- Un tas est un arbre binaire parfait tel que le contenu de chaque noeud est inférieur à celui de ses fils.
- Le minimum de l'arbre se trouve donc à la racine.



1	2	3	4	5	6	7	8	9	10
3	5	9	6	8	11	10	12	18	14

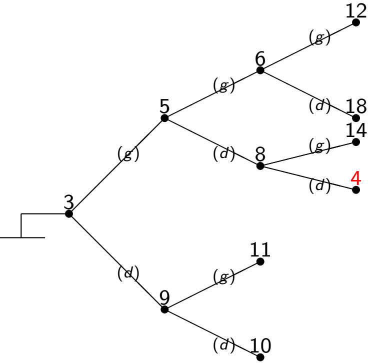
Ajout d'un élément dans le tas

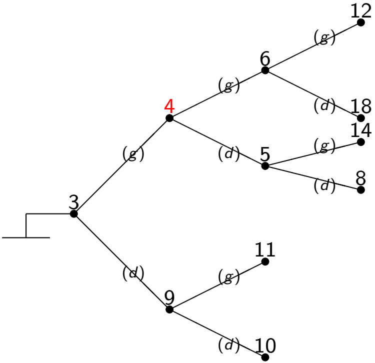
- On met l'élément ajouté dans la feuille la plus à droite.
- On le permute avec son père jusqu'à ce qu'il trouve sa place.
- p : taille du tas.

procédure ajouter **(**E/S t : Tableau d'Éléments, p : Naturel, x : Élément**)**

```
    Déclaration i : Naturel
début
    p ← p+1
    t[p] ← x
    i ← p
    tant que (i>1) et (t[i]<t[i div 2]) faire
        échanger(t[i],t[i div 2])
        i ← i div 2
    fintantque
fin
```

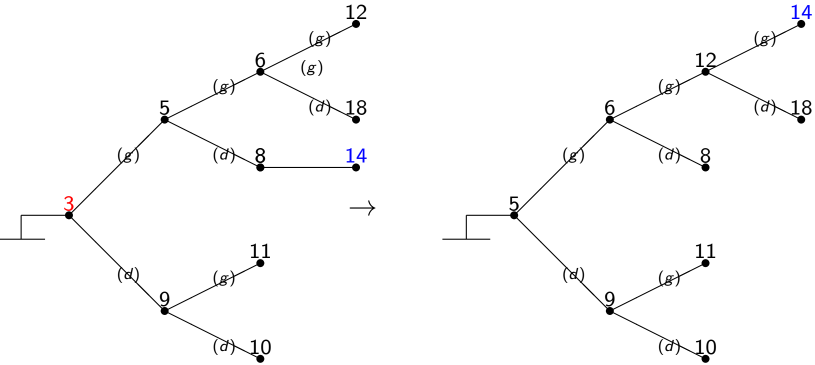
Exemple





Suppression du minimum

- On remplace la racine (le minimum) par le dernier élément.
- On redescend ce dernier élément jusqu'à ce qu'il trouve sa place.



L'algorithme détruire

```
procédure détruire (E/S t : Tableau d'Éléments, p : Naturel, min :
Élément)
   Déclaration i,j : Naturel
début
   min←t[1] ;t[1]←t[p] ;p←p-1 ;i←1
   tant que i≤(p div 2) faire
      si (2*i=p) ou (t[2*i]<t[2*i+1]) alors
         j←2*i sinon j←2*i+1
      finsi
      si t[i]≤t[j] alors
         retourner
      finsi
      échanger(t[i],t[j])
      i ← j
   fintantque
fin
```

Explications

- Le sommet le plus à droite a 1 fils ssi p est pair.
- Demonstration :

$$p = \sum_{j=0}^{h-1} 2^j + 2(i-1) + nb = \sum_{j=1}^{h-1} 2^j + 2(i-1) + nb + 1$$

  i : indice au niveau h-1 du dernier sommet avoir nb=1 ou nb=2 fils.

- Si p est pair, p div 2 correspond au dernier sommet de niveau $h-1$ avec 1 fils.
- Demonstration : $p = \sum_{j=0}^{h-1} 2^j + 2(i-1) + 1 = 2\left(\sum_{j=0}^{h-2} 2^j + i\right)$

  $p\ div\ 2 = \sum_{j=0}^{h-2} 2^j + i$: indice de i dans le tableau t.

- Si p est impair, p div 2 correspond au dernier sommet de niveau $h-1$ avec 2 fils.
- Conclusion : p div 2 +1 est le premier sommet sans fils.

Complexités et Tri par tas (heapsort)

- Complexités liées à la hauteur de l'arbre ($log_2(p)$)
  - Ajout : $log_2(p)$
  - Détruire : $log_2(p)$
- Tri basé sur un algorithme de sélection
  - On ajoute 1 à 1 les éléments du tableau dans le tas (n fois).
  - On retire le minimum du tas pour le mettre dans le tableau (n fois).

procédure triParTas **(**t :Tableau[1..n] d'Entiers**)**
  Déclaration p,min : Entiers
début
  p $\leftarrow$ 0
  tant que p$<$n **faire**
    ajouter(t,p,t[p$+$1])
    Remarque : *p incrémenté par ajouter*
  fintantque
  tant que p$>$1 **faire**
    détruire(t,p,min)
    Remarque : *p décrémenté par détruire*
    t[p$+$1] $\leftarrow$ min
  fintantque
fin

Tris par tas : Exemple (1/2)

‖ 10	8	11	15	2	3		Tableau initial
10 ‖	8	11	15	2	3		ajout de 10
8	10 ‖	11	15	2	3		ajout de 8
8	10	11 ‖	15	2	3		ajout de 11
8	10	11	15 ‖	2	3		ajout de 15
2	8	11	15	10 ‖	3		ajout de 2
2	8	3	15	10	11 ‖		ajout de 3

Tris par tas : Exemple (2/2)

2 8 3 15 10 11 ‖	Tas initial
3 8 11 15 10 ‖ 2	suppression de 2
8 10 11 15 ‖ 3 2	suppression de 3
10 15 11 ‖ 8 3 2	suppression de 8
11 15 ‖ 10 8 3 2	suppression de 10
15 ‖ 11 10 8 3 2	suppression de 11
‖ 15 11 10 8 3 2	Fin du tri.

- On fait n ajouter et n détruire.

$$\begin{aligned} T_n &= 2 \quad \sum_{i=1}^{n} \log_2(i) \\ &= \quad 2log_2(n!) \end{aligned}$$

- Utilisation de la formule de Stirling :

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- Donc complexité en $\mathcal{O}(n\log_2(n))$

Du récursif à l'itératif

Luc Brun
luc.brun@ensicaen.fr

Plan

- Un seul appel récursif
- Deux appels récursif
  - Cas de base
  - Avec tests séparés

Notations

- recn : fonction récursive.
- itern : version itérative de recn
- A(x), B(x),C(x),D(x) : fonctions auxilières
- f(x), g(x) : transformation de x avant l'appel récursif.

Un seul appel : Cas 1

procédure rec1 **(**E x : élément**)**
début
  si C(x) **alors**
    A(x)
    rec1(f(x))
  finsi
fin

- Si $C(x) : x = 0$ et $f(x) : x - 1$,
- rec1(4) implique les appels de A(4),A(3)A(2),A(1).

procédure iter1 **(**E x : élément**)**
début
  tant que C(x) **faire**
    A(x)
    x $\leftarrow$ f(x)
  fintantque
fin

Un seul appel : Cas 2

procédure rec2 **(**E x : élément**)**
début
  si C(x) **alors**
    A(x)
    rec2(f(x))
    B(x)
  finsi
fin

- Si $C(x)$: $x = 0$ et $f(x)$: $x - 1$,
- rec2(4) implique les appels de A(4),A(3)A(2),A(1),B(1),B(2),B(3),B(4).

Un seul appel : Cas 2

```
procédure iter2 (E x : élément)
    Déclaration p : Pile
début

    p ← creer_pile()
    tant que C(x) faire
        A(x)
        empiler(x,p)
        x ← f(x)
    fintantque

    tant que non vide(p) faire
        x ← sommet(p)
        depiler(p)
        B(x)
    fintantque

fin
```

Cas 2 : Un exemple

- Affichage a l'envers des éléments d'une liste après l'élément courrant.

procédure afficheListe **(**E l : Liste**)**
    Déclaration elt : élément
début
    si suivant(l) **alors**
        elt $\leftarrow$ courrant(l)
        afficheListe(l)
        écrire(elt)
    finsi
fin

Cas 2 : Un exemple

- Affichage a l'envers des éléments d'une liste après l'élément courrant.

procédure afficheListe2 **(**E l : Liste**)**
    Déclaration elt : élément,p : Pile
début

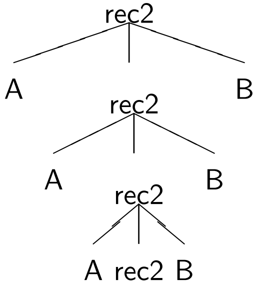
    p $\leftarrow$ creer_pile()
    tant que suivant(l) **faire**
        empiler(courrant(l),p)
    fintantque

    tant que non vide(p) **faire**
        elt $\leftarrow$ sommet(p)
        depiler(p)
        écrire(elt)
    fintantque

fin

Cas 2 : Représentation par arbre d'exécution.

- Exemple avec 3 appels récursif.



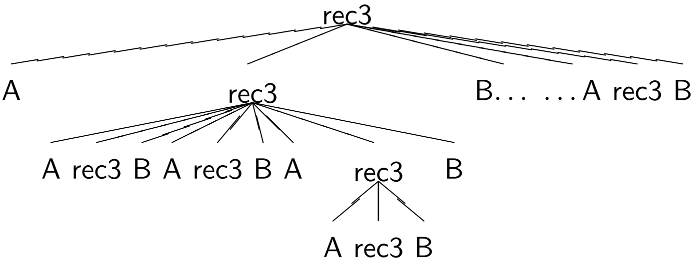
- Ordre des Appels :
rec2(x),A(x),rec2(f(x)),A(x),rec2(f(x)),A(x),rec2(f(x)),B(x),B(x),B(x).

Un seul appel : Cas 3

```
procédure rec3 (E x : élément)
début
    tant que C(x) faire
        A(x)
        rec3(f(x))
        B(x)
    fintantque
fin
```

- Si $C(x) : x > 0$, $f(x) : x - 2$ et $B(x) : x \leftarrow x - 1$
- rec3(4) implique les appels de A(4),A(2),A(1),A(3),A(1),A(2),A(1).

Cas 3 : arbre d'exécution



Un seul appel : Cas 3

```
procédure iter3 (E x : élément)
   Déclaration p : Pile
début

   p ← creer_pile()
   tant que C(x) et non vide(p)
   faire
      tant que C(x) faire
         A(x)
         empiler(x,p)
         x ← f(x)
      fintantque
      x ← sommet(p)
      depiler(p)
      B(x)
   fintantque

fin
```

Deux appels : Cas 4

- Deux appels récursifs avec deux transformations f et g.

```
procédure rec4 (E x : élément)
début
    si C(x) alors
        A(x)
        rec4(f(x))
        rec4(g(x))
    finsi
fin
```

Deux appels : Cas 4

- Soit A'(x)=A(x)REC4(f(x)). On se retrouve alors dans le cas 1.

procédure rec4 **(**E x : élément**)**
début
    tant que C(x) **faire**
        A(x)
        rec4(f(x))
        x $\leftarrow$ g(x)
    fintantque
fin

Deux appels : Cas 4

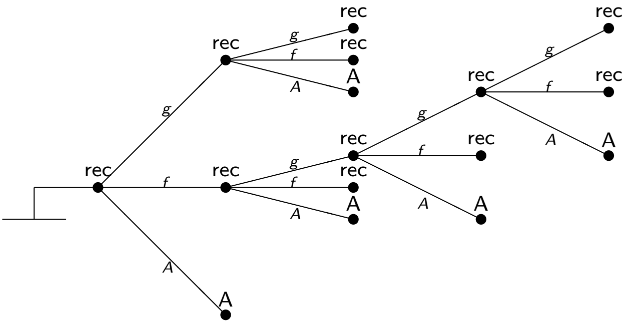
- Soit B'(x)=$x \leftarrow g(x)$. On se retrouve alors dans le cas 3.

procédure iter4 **(**E x : élément**)**
  Déclaration p : Pile
début

  p ← creer_pile()
  tant que C(x) et non vide(p)
  faire
    tant que C(x) **faire**
      A(x)
      empiler(x,p)
      x ← f(x)
    fintantque
    x ← sommet(p)
    depiler(p)
    x ← g(x)
  fintantque

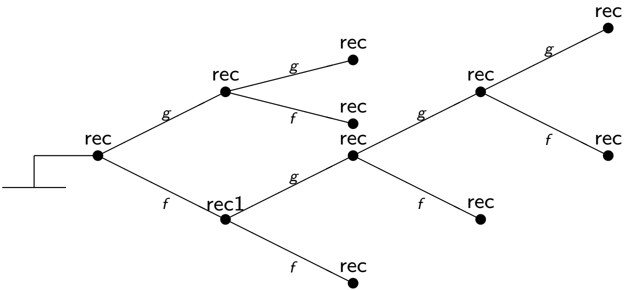
fin

Cas 4 : arbre d'exécution



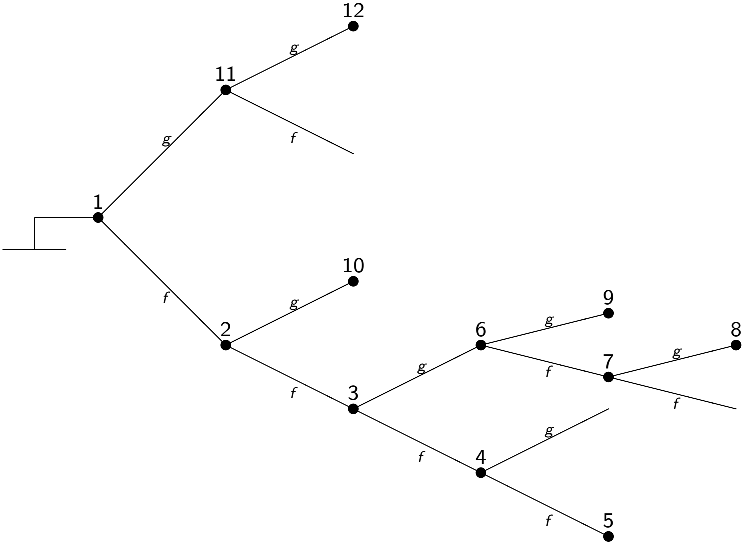
Cas 4 : arbre d'exécution

- On oublie les A



- Les parcours des branches gauches sont codés par la boucle intérieure (en f(x))
- Les déplacements droite sont effectués en sortant de la boucle intérieure.

Cas 4 : Un exemple de parcours



Cas 4 bis

- On empile x même lorsque C(x) est faux : inutile

```
procédure iter4 (E x : élément)
    Déclaration p : Pile
début
    p ← creer_pile()
    empiler(x,p)
    tant que non vide(p) faire
        x ← sommet(p)
        depiler(p)
        tant que C(x) faire
            A(x)
            si C(g(x)) alors
                empiler(g(x),p)
            finsi
            x ← f(x)
        fintantque
    fintantque
fin
```

```
procédure rec5 (E x : élément)
début
    si C(x) alors
        A(x)
        rec5(f(x))
        B(x)
        rec5(g(x))
    finsi
fin
```

Cas 5

- Le retour de récursion s'effectue lorsque l'on dépile.

```
procédure iter5 (E x : élément)
    Déclaration p : Pile
début

    p ← creer_pile()
    tant que C(x) et non vide(p)
    faire
        tant que C(x) faire
            A(x)
            empiler(x,p)
            x ← f(x)
        fintantque
        x ← sommet(p)
        depiler(p)
        B(x)
        x ← g(x)
    fintantque

fin
```

Variante à deux tests

```
procédure rec6 (E x : élément)
début
    A(x)
    si C1(x) alors
        rec5(f(x))
    finsi
    si C2(x) alors
        rec5(g(x))
    finsi
fin
```

Cas 6

```
procédure iter6 (E x : élément)
    Déclaration p : Pile
début

    p ← creer_pile()
    empiler(x,p)
    tant que non vide(p) faire
        x ← sommet(p)
        depiler(p)
        répéter
            A(x)
            si C2(g(x)) alors
                empiler(g(x),p)
            finsi
            filsGauche ← C1(x)
            x ← f(x)
        jusqu'à ce que non fils gauche

    fintantque

fin
```

Attention, le test $C1$ est sur x et non sur $f(x)$ (d'où le répéter... jusqu'a.

Une dernière Variante

```
procédure rec7 (E x : élément)
début
    A(x)
    si C1(x) alors
        rec5(f(x))
    finsi
    B(x)
    si C2(x) alors
        rec5(g(x))
    finsi
fin
```

procédure iter7 **(**E x : élément**)**
Déclaration p : Pile,filsGauche : booléen
début

```
p ← creer_pile()
empiler(x,p)
tant que non vide(p) faire
    x ← sommet(p)
    depiler(p)
    répéter
        A(x)
        empiler(x,p)
        filsGauche ← C1(x)
        x ← f(x)
    jusqu'à ce que non filsGauche
    répéter
        x ← sommet(p)
        depiler(p)
        B(x)
    jusqu'à ce que C2(x) ou vide(p)
    si C2(x) alors
        empiler(g(x),p)
    finsi
fintantque
```

fin

Évaluation d'expressions

Luc Brun
luc.brun@ensicaen.fr

Plan

- Les différents types d'expressions
  - Expression complètement parenthésée (ECP),
  - Expression préfixée (EPRE),
  - Expression postfixée (EPOST),
  - Expression infixée (EINF).
- Évaluation d'expressions
  - postfixée,
  - complètement parenthésée,
- Conversion d'expressions
  - ECP à postfixée,
  - infixée à postfixée.

Définition d'une expression

- Définition : Une expression est une suite de variables combinées par un ensemble d'opérateurs avec éventuellement un jeux de parenthèses ouvrantes et fermantes.

- Exemple :

$$X * (A + B) * (C + D)$$

- NB : On ne fait pas intervenir de constantes ou de fonctions (petite simplification).

- L'évaluation d'une expression implique la définition d'un *environnement* (affectation de chaque variable à une valeur).

Les opérateurs

- Opérateurs binaires
  - opérateurs arithmétiques
  $$\{+, -, *, /\},$$
  - opérateurs logiques
  $$\{<, >, \leq, \geq, \neq, =, \mathit{et}, \mathit{ou}\},$$
- Opérateurs unaires
  - opérateurs arithmétiques
  $$\{\oplus, \ominus\}$$
  NB : $\oplus \equiv +$, $\ominus \equiv -$.
  - opérateurs logiques
  $$\mathit{non}.$$

Expression complètement parenthésée

- Une expression complètement parenthésée se construit à partir des règles suivantes :

1. Une variable est une ECP,
2. si x et y sont des ECP et β un *opérateur binaire*, alors

   $(x \ \beta \ y)$ est une ECP,
3. si x est une ECP et α un *opérateur unaire* alors

   $(\alpha \ x)$ est une ECP,
4. Il n'y a pas d'autre ECP que celles formées par les 3 règles précédentes.

Grammaire des ECP

- Grammaire BNF (Backus-Naur Form)

$$
\begin{array}{lcl}
<\textbf{ecp}> & \rightarrow & (\ <\textbf{ecp}><\textbf{optbin}><\textbf{ecp}>)\ \text{—} \\
 & & (<\textbf{optun}><\textbf{ecp}>)\ \text{—} \\
 & & <\textbf{variable}> \\
<\textbf{optbin}> & \rightarrow & +|-|*|/|<|\leq|=|\geq|>|\neq|et|ou \\
<\textbf{optun}> & \rightarrow & \oplus|\ominus|non \\
<\textbf{variable}> & \rightarrow & A|B|C|\ldots|Z
\end{array}
$$

Exemples d'ECP

- ECP conformes :
- ((A+B)*C)
- (((A/B)=C)et(E¡F))
- ($\ominus$ A)
- ECP non conformes :
- (A)
- ((A+B))
- A $\oplus$ B

Expressions préfixées

Les expressions préfixées (EPRE) se construisent à partir des 4 règles suivantes :

1. une variable est une EPRE,
2. si x et y sont des EPRE et β un opérateur binaire alors
   $\beta\ x\ y$ est une EPRE,
3. Si x est une EPRE et α un opérateur unaire, alors
   $\alpha\ x$ est une EPRE,
4. il n'y a pas d'autres EPRE que celles formées à partir des 3 règles précédentes.

Grammaire des EPRE

- Règles :

$$
\begin{array}{lcl}
<\textbf{epre}> & \rightarrow & <\textbf{optbin}>\ <\textbf{epre}>\ <\textbf{epre}> \text{ — } \\
 & & <\textbf{optun}><\textbf{epre}> \text{ — } \\
 & & <\textbf{variable}> \\
<\textbf{optbin}> & \rightarrow & +|-|*|/|<|\leq|\models|\geq|>|\neq|et|ou \\
<\textbf{optun}> & \rightarrow & \oplus|\ominus|non \\
<\textbf{variable}> & \rightarrow & A|B|C|\ldots|Z
\end{array}
$$

- NB : Seule la première règle a été changée.

Exemples d'EPRE

- Expressions correctes :

EPRE	ECP
+ - A B C	((A-B)+C)
et non < A B C	((non(A<B))et C)
non = + A B C	(non ((A+B)=C))

- Expressions incorrectes

- A - + B C
- non B C

Expressions postfixées

- Expressions utilisées par les calculatrices HP (cf TP Web) et données par les 4 règles :

1. une variable est une EPOST,
2. si x et y sont des EPOST et β un opérateur binaire alors
   x y β est une EPOST,
3. Si x est une EPOST et α un opérateur unaire, alors
   x α est une EPOST,
4. il n'y a pas d'autres EPOST que celles formées à partir des 3 règles précédentes.

Grammaire des EPOST

- Règles :

$$
\begin{array}{lcl}
<\textbf{epost}> & \rightarrow & <\textbf{epost}> <\textbf{epost}> <\textbf{optbin}> \text{ — } \\
 & & <\textbf{epost}> <\textbf{optun}> \text{ — } \\
 & & <\textbf{variable}> \\
<\textbf{optbin}> & \rightarrow & + | - | * | / | < | \leq | = | \geq | > | \neq | et | ou \\
<\textbf{optun}> & \rightarrow & \oplus | \ominus | non \\
<\textbf{variable}> & \rightarrow & A | B | C | \ldots | Z
\end{array}
$$

- NB : Seule la première règle a été changée.

Exemples d'EPOST

ECP	EPRE	EPOST
(((A+B)-C)/D)	/ - + A B C D	A B + C - D /
((A < B) et (non C))	et < A B non C	A B < C non et
((non (A < B)) et (C>D))	et non < A B > C D	A B < non C D > et

Expressions Infixées

- Écriture ECP sans ambiguï té mais parfois un peu lourde.
- Suppressions de certaines parenthèses par 2 données implicites :

1. Priorité des opérateurs

$$A * B + C \Leftrightarrow ((A * B) + C)$$

2. Priorité à gauche pour les opérateurs de même priorité

$$A + B - C \Leftrightarrow ((A + B) - C)$$

- Les parenthèses imposent des priorités

$$A - B - C \Leftrightarrow ((A - B) - C) \neq A - (B - C) \Leftrightarrow (A - (B - C))$$

Priorité des opérateurs

Opérateur	Priorité
(	0
$<, \leq, =, \geq, <, \neq, >$	1
ou, $+$, -	2
et, *, $/$	3
$\ominus, \oplus$, non	4

Grammaire des EINF

- $\{expr\}$: expression éventuellement présente un nombre quelconque de fois.

$$
\begin{array}{lcl}
<\textbf{einf}> & \rightarrow & <\textbf{esimple}>\ \{<\textbf{op1}>\ <\textbf{esimple}>\} \\
<\textbf{esimple}> & \rightarrow & <\textbf{terme}>\{<\textbf{op2}><\textbf{terme}>\} \\
<\textbf{terme}> & \rightarrow & <\textbf{facteur}>\{<\textbf{op3}><\textbf{facteur}>\} \\
<\textbf{facteur}> & \rightarrow & <\textbf{variable}> \text{ — } \\
 & & <\textbf{op4}><\textbf{facteur}> \text{ — } \\
 & & (<\textbf{einf}>) \\
<\textbf{op1}> & \rightarrow & < \mid = \mid > \mid \leq \mid \neq \mid \geq \\
<\textbf{op2}> & \rightarrow & + \mid - \mid \mathit{ou} \\
<\textbf{op3}> & \rightarrow & * \mid / \mid \mathit{et} \\
<\textbf{op4}> & \rightarrow & \ominus \mid \oplus \mid \mathit{non} \\
<\textbf{variable}> & \rightarrow & A \mid B \mid C \mid \ldots \mid Z
\end{array}
$$

Exemples d'EINF

EINF	ECP
A*B+C-D	(((A * B) +C)-D)
A+B*C	(A+(B * C))
A=B et C>D	((A=(B et C))> D)
A et B ou C et D	((A et B) ou (C et D))
A et (B ou C) et D	((A et (B ou C)) et D)
non A et B ou C et non D	(((non A) et B) ou (C et (non D)))

Évaluation d'expression

- L'évaluation s'effectue par rapport à un environnement (affectation des variables).
- Idée : Ordonnancer l'évaluation de façon à ce que l'évaluation d'une opération s'effectue sur
  - des variables ou
  - des expressions déjà évaluées.
- Soit S appliquant β sur x et y.

$$\begin{aligned}\text{evaluation(S)} \quad = \quad & \text{opération}(\beta, \\ & \text{si variable(x) alors valeur(x) sinon evaluation(x)}, \\ & \text{si variable(y) alors valeur(y) sinon evaluation(y)})\end{aligned}$$

Exemple d'évaluation

- Soit à évaluer S=(A+(B*C)) avec $A = 2$, $B = 3$, $C = 4$.

$$
\begin{aligned}
\text{evaluation(S)} &= \text{opération}(+,\text{A},\text{évaluation}(\text{B*C})) \\
&= \text{opération}(+,\text{A},\text{opération}(*,\text{B},\text{C})) \\
&= \text{opération}(+,2,\text{opération}(*,3,4)) \\
&= \text{opération}(+,2,12) \\
&= 14
\end{aligned}
$$

Évaluation d'une expression postfixée

- Les opérandes apparaissent avant l'opération.
- L'évaluation d'une opération fournie soit le résultat soit un nouvel opérande.

$$
\begin{aligned}
10\ 3 + 5\ 6\ \text{-}\ \text{-} &= 13\ 5\ 6\ \text{-}\ \text{-} \\
&= 13\ \text{-}1\ \text{-} \\
&= 14
\end{aligned}
$$

- Idée :

- empiler les opérandes,
- dépiler pour effectuer une opération.

- évaluons A B * C D + / avec A=2,B=3,C=2,D=1

(pile vide)	2	3 2	6
pile vide	empiler A	empiler B	evaluer *
AB*CD+/#	*A*B*CD+/# ↑	A*B**CD+/# ↑	AB∗CD+/# ↑
2 6	1 2 6	3 6	2
empiler C	empiler D	évaluer +	evaluer /
AB**C*D+/# ↑	AB*C*D*+/# ↑	AB*CD+/# ↑	AB*CD+/# ↑

Évaluation d'EPOST : Énumération des cas

Soit epost[1..MAX] une chaîne de caractères contenant une EPOST et i une position dans la chaîne.

- Si epost[i]='#' : l'évaluation est terminé. Résultat en sommet de pile.
- Si epost[i]≠'#'
  - Si epost[i] est une variable : empiler sa valeur
  - Si epost[i] est un opérateur :
    1. on dépile son (ou ses 2 opérandes),
    2. on effectue l'opération et
    3. on empile le résultat.

Évaluation d'EPOST : l'algorithme

```
fonction evalPost (epost : Tableau[1...MAX] deCaratère ) : Réel
   Déclaration i :Entier,valG,valD :Réel,p : Pile
début

   p ← creer()
   i ← 1
   tant que epost[i]≠'#' faire
      si variable(epost[i]) alors
         empiler(epost[i],p)
      sinon
         si  unaire(epost[i]) alors
            valD ← dépiler(p)
            valD ←
            oper1(epost[i],valD)
            empiler(valD,p)
         sinon
            valD ← dépiler(p)
            valG ← dépiler(p)
            valD ←
            oper2(valG,epost[i],valD)

            empiler(valD,p)
         finsi
      finsi
      i ← i+1
   fintantque
   retourner sommet(p)
fin
```

Évaluation d'ECP

- Utilisation d'une pile à travers trois types d'actions :

1. Si le symbole lu est un opérateur : on l'empile
2. si c'est une variable : on empile sa valeur,
3. Si c'est une parenthèse droite :

   1. on dépile une sous expression et
   2. on empile le résultat.

Évaluation d'ECP : Exemple

- évaluons (A * (B+C)) avec A=2,B=3,C=1.

			3
	*	*	*
2	2	2	2
(A*(B+C))#	(A*(B+C))#	(A*(B+C))#	(A*(B+C))#

	1		
+	+		
3	3	4	
*	*	*	
2	2	2	8
(A*(B+C))#	(A*(B+C))#	(A*(B+C))#	(A*(B+C))#

Évaluation d'ECP : l'algorithme

```
fonction evalEcp (ecp : Tableau[1...MAX] deCaratère ) : Réel
    Déclaration i :Entier,op :Caratère,valG,valD :Réel,p : Pile
début

    p ← creer()
    i ← 1
    tant que ecp[i]≠'#' faire
        si variable(ecp[i]) alors
            empiler(valeur(ecp[i]),p)
            continuer
        finsi
        si operateur(ecp[i]) alors
            empiler(ecp[i],p)
            continuer
        finsi
        si  ecp[i]=')' alors
            valD ← dépiler(p)
            op ← dépiler(p)
            si unaire(op) alors

                empiler(oper1(op,valD),p)

            sinon
                valG ← dépiler(p)

                emp.(op2(valG,op,valD),p)

            finsi
        finsi
        i ← i+1
    fintantque
    retourner sommet(p)
```

Conversion ECP à EPOST

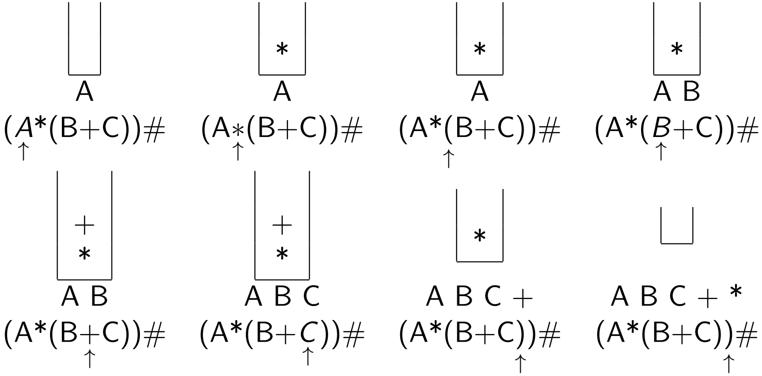
- ECP lisibles, tapé par exemple dans un code source,
- EPOST directement interprétable par une machine.
- Nécessité de convertir.
- Remarque : Lors de la lecture d'une ECP on rencontre les opérandes dans le même ordre que dans l'EPOST. Seul l'opérateur est placé à la fin.

$$(A + B) \to AB+$$

- Idée : Empiler les opérateurs et les dépiler sur les parenthèses fermantes.
  - Opérateur : empiler,
  - variable : écrire dans chaî ne résultat.
  - parenthèse fermante : dépiler un opérateur
  - parenthèse ouvrante : ne rien faire.

Exemple de conversion

- Convertissons (A * (B+C))



Conversion ECP à EPOST : l'algorithme

procédure convEcpEpost **(** **E** ecp : **Tableau[**$1 \dots MAX$**] deCaratère** , **S** epost : **Tableau[**$1 \dots MAX$**] deCaratère** **)**

Déclaration indecp,indepost :**Entier**,p : Pile

début

```
p ← creer()
indecp ← 1
indepost ← 1
tant que ecp[indecp]≠'#' faire
    si operateur(ecp[i]) alors
        empiler(ecp[i],p)
        continuer
    finsi
    si variable(ecp[i]) alors
        epost[indepost] ← ecp[indecp]
        indepost ← indepost+1
        continuer
    finsi
    si ecp[i]=')' alors
        epost[indepost] ← dépiler(p)
        indepost ← indepost+1
    finsi
    indecp ← indecp+1
fintantque
epost[indepost] ← #
```

Conversion ECP/EPOST : Amélioration

- Détection d'erreurs de syntaxe dans l'ECP.
- La pile ne doit pas être vide avant la fin (trop de parenthèses fermantes) et vide à la fin (pas assez).
- Un caractère d'ECP est soit :
- un opérateur,
- une variable,
- une parenthèse fermante,
- une parenthèse ouvrante.

Conversion EINF à EPOST

- Problème compliqué par l'absence de parenthèses. On applique les règles suivantes :

- variable : empiler.

- opérateur : empiler.

  Il faut dépiler auparavant tous les opérateurs de priorité supérieure ou égale.

- parenthèse gauche : empiler.

  Délimite une sous expression.

- parenthèse droite : dépiler

  jusqu'à une parenthèse gauche.

- Convertissons A*B+C/(D+E)

	A	*A**B+C/(D+E)#
*	A	A*B+C/(D+E)#
*	A B	A**B*+C/(D+E)#
+	A B*	A*B+C/(D+
+	A B *C	A*B+*C*/(D+E)#
/ +	A B *C	A*B+C/(D+E)#
(/ +	A B* C +	A*B+C/(D+E)#
(/ +	A B* C D	A*B+C/(*D*+

Exemple de conversion (2/2)

- A*B+C/(D+E)

(
/
+

A B *C D
A*B+C/(*D*+E)#
↑

+
(
/
+

A B *C D
A*B+C/(D+E)#
↑

+
(
/
+

A B* C D E
A*B+C/(D+*E*)#
↑

/
+

A B* C D E +
A*B+C/(D+E)#
↑

A B* C D E + / +
A*B+C/(D+E)#

Conversion EINF/EPOST : Points délicats

- Trois points délicats :

- Empilement d'une opération,
- Rencontre d'une parenthèse fermante,
- Fin de l'algorithme et vidage de la pile.

Empilement d'une opération

procédure traiterOpt **(** **E** op :**Caratère** , **S** epost : **Tableau[**$1 \ldots MAX$**]** **deCaratère** **E/S** p : Pile,indpost : **Naturel)**

  Déclaration dépil : **Booléen**,élément : **Caratère**

début

  dépil ← non vide(p)

  tant que dépil **faire**

    si priorité(sommet(p))≥priorité(op) **alors**

      élément ← depiler(p)

      epost[indpost] ← élément

      indpost ← indpost+1

      dépil ← non vide(p)

    sinon

      dépil ← faux

    finsi

  fintantque

  empiler(op)

fin

procédure traiterPF **(S** epost : **Tableau[**$1 \ldots MAX$**] deCaratère** , **E/S** indpost : **Naturel**,p : Pile **)**

  Déclaration élément : **Caratère**

début

  élément $\leftarrow$ dépiler(p)

  tant que élément $\neq$ '(' **faire**

    epost[indpost] $\leftarrow$ élément

    indpost $\leftarrow$ indpost$+1$

    élément $\leftarrow$ depiler(p)

  fintantque

fin

Vidage de la pile

procédure traiterFin **(S** epost : **Tableau[**$1 \ldots MAX$**] deCaratère** , **E/S** indpost : **Naturel**,p : Pile **)**

  Déclaration élément : **Caratère**

début

  tant que non vide(p) **faire**

    epost[indpost] $\leftarrow$ depiler(p)

    indpost $\leftarrow$ indpost$+1$

  fintantque

  epost[indpost] $\leftarrow$ '$\#$'

fin

procédure convEinfEpost **(** **E** einf : **Tableau[**$1 \ldots MAX$**] deCaratère** , **S** epost : **Tableau[**$1 \ldots MAX$**] deCaratère** **)**

Déclaration indinf,indpost :**Entier**,p : Pile

début

```
p ← creer()
indinf ← 1
indpost ← 1
tant que einf[indinf]≠'#' faire
    si operateur(einf[indinf]) alors
        traiterOpt(einf[indinf],epost,indpost,p)
        continuer
    finsi
    si variable(einf[indinf]) alors
        epost[indpost] ←
        einf[indinf]
        indpost ← indpost+1
        continuer
    finsi
    si einf[i]=')' alors
        traiterPF(epost,indpost,p)
    sinon
        empiler('(');
    finsi
    indinf ← indinf+1
fintantque
traiterFin(epost,indpost,p)
```