

# Algorithmes sur les listes chaînées et les types de fonction et procédure

## I3 - Algorithmique et programmation

Nicolas Delestre, Nicolas Malandain

# Plan

- 1 Rappel
- 2 Algorithmes sur les listes chaînées
  - Afficher
  - Obtenir le nombre d'éléments
  - Insérer un élément
- 3 Le jeu du serpent
  - Analyse
  - Conception

# Liste chaînée 1 / 2

## Définition

Une liste chaînée est soit :

- une liste vide
- un élément suivi d'une liste chaînée

## Conception

**Type** ListeChaine =  $\hat{}$ Noeud

**Type** Noeud = **Structure**

element : Element

listeSuiivante : ListeChaine

**finstructure**

## Liste chaînée 2 / 2

## Utilisation de fonctions et procédures

- **fonction** listeVide () : ListeChaînee
- **fonction** estVide (uneListe : ListeChaînee) : **Booleen**
- **procédure** ajouter (**E/S** uneListe : ListeChaînee, **E** element : Element)
- **fonction** obtenirElement (uneListe : ListeChaînee) : Element  
     |précondition(s) *non(estVide(uneListe))*
- **fonction** obtenirListeSuivante (uneListe : ListeChaînee) : ListeChaînee  
     |précondition(s) *non(estVide(uneListe))*
- **procédure** fixerListeSuivante (**E/S** uneListe : ListeChaînee, **E** nelleSuite : ListeChaînee)  
     |précondition(s) *non(estVide(uneListe))*
- **procédure** supprimerTete (**E/S** l : ListeChaînee)  
     |précondition(s) *non estVide(l)*
- **procédure** supprimer (**E/S** uneListe : ListeChaînee)

# Algorithmes

- Afficher les éléments d'une liste chaînée
- Obtenir le nombre éléments d'une liste chaînée
- Insérer un élément à la  $i$ ème position :
  - si  $i > \text{longueur}(l)$  alors l'insertion se fait en fin de liste
  - si  $i \leq 1$  alors l'insertion se fait en début de liste
- À chaque fois, nous allons avoir un algorithme :
  - itératif utilisant directement la structure de liste chaînée (utilisation des pointeurs, des champs de structure et des procédures d'allocation et libération dynamique de la mémoire)
  - itératif utilisant les fonctions et procédures d'encapsulation
  - récursif utilisant les fonctions et procédures d'encapsulation

## Version itérative (sans les fonctions et procédures)

**procédure** afficher (**E** l : ListeChaînee)**debut****tant que** l  $\neq$  NIL **faire**

afficherElement(l^.element)

    l  $\leftarrow$  l^.listeSuivante**fin tant que****fin**

## Version itérative (avec les fonctions et procédures)

```
procédure afficher (E l : ListeChainee)
debut
  tant que non estVide(l) faire
    afficherElement(obtenirElement(l))
    l ← obtenirListeSuivante(l)
  fin tant que
fin
```

## Version récursive (avec les fonctions et procédures)

```
procédure afficher (E l : ListeChaine)  
debut  
  si non estVide(l) alors  
    afficherElement(obtenirElement(l))  
    afficher(obtenirListeSuivante(l))  
  finsi  
fin
```



## Nombre d'éléments 1 / 3

## Version itérative (sans les fonctions et procédures)

**fonction** nbElements (l : ListeChaînee) : Naturel

Déclaration res : Naturel

**debut**

res ← 0

**tant que** l ≠ NIL **faire**

res ← res+1

l ← l^.listeSuivante

**fintantque****retourner** res**fin**

## Nombre d'éléments 2 / 3

## Version itérative (avec les fonctions et procédures)

**fonction** nbElements (l : ListeChaınee) : Naturel**Déclaration** res : Naturel**debut**

res ← 0

**tant que** non estVide(l) **faire**

res ← res+1

l ← obtenirListeSuivante(l)

**fintantque****retourner** res**fin**

## Nombre d'éléments 3 / 3

Version récursive (avec les fonctions et procédures)

**fonction** nbElements (l : ListeChaine) : **Naturel**

**debut**

**si** estVide(l) **alors**

**retourner** 0

**sinon**

**retourner** 1+nbElements(obtenirListeSuivante(l))

**finsi**

**fin**

## Insérer 1 / 3

## Version itérative (sans les fonctions et procédures)

**procédure** inserer (**E/S** l : ListeChaînee, **E** i : NaturelNonNul, e : Element)

**Déclaration** g,d,temp : ListeChaîne

**debut**

si l = NIL ou  $i \leq 1$  alors

d ← l

allouer(l)

l^.element ← e

l^.listeSuiVante ← d

**sinon**

g ← l

d ← l^.listeSuiVante

i ← i-1

**tant que** i ≠ 1 et d ≠ NIL **faire**

i ← i-1

g ← d

d ← d^.listeSuiVante

**fintantque**

allouer(temp)

temp^.element ← e

temp^.listeSuiVante ← d

g^.listeSuiVante ← temp

**finsi**

**fin**

## Insérer 2 / 3

## Version itérative (avec les fonctions et procédures)

**procédure** inserer (**E/S** l : ListeChaine, **E** i : NaturelNonNul, e : Element)

**Déclaration** g,d,temp : ListeChaine

**debut**

**si** estVide(l) ou  $i \leq 1$  **alors**

ajouter(l,e)

**sinon**

g ← l

d ← obtenirListeSuivante(l)

i ← i-1

**tant que**  $i \neq 1$  et non estVide(d) **faire**

i ← i-1

g ← d

d ← obtenirListeSuivante(d)

**fintantque**

ajouter(d,e)

fixerListeSuivante(g,d)

**finsi**

**fin**

## Version récursive (avec les fonctions et procédures)

**procédure** inserer (**E/S** l : ListeChaínee, **E** i : NaturelNonNul, e : Element)

**Déclaration** temp : ListeChaínee

**debut**

**si** estVide(l) ou  $i \leq 1$  **alors**

ajouter(l,e)

**sinon**

temp  $\leftarrow$  obtenirListeSuivante(l)

inserer(temp,i-1,e)

fixerListeSuivante(l,temp)

**finsi**

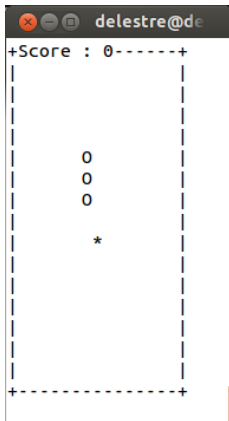
**fin**

# Jeu du serpent

## Principe

L'objectif global de cet exercice est de concevoir une partie du jeu du serpent : jeu des années 80, que l'on retrouve sur tous les vieux téléphones, qui consiste à guider un serpent de façon à ce qu'il mange des proies sans qu'il se mange lui même.

Le serpent avance constamment dans une direction. On peut le faire changer de direction. Lorsqu'il arrive au bord de l'aire de jeu, il réapparaît de l'autre côté. Enfin lorsqu'il mange une proie, il grandit pendant un certain temps (seule sa tête avance). La partie se termine lorsque le serpent mange une partie de son corps.



# Analyse 1 / 2

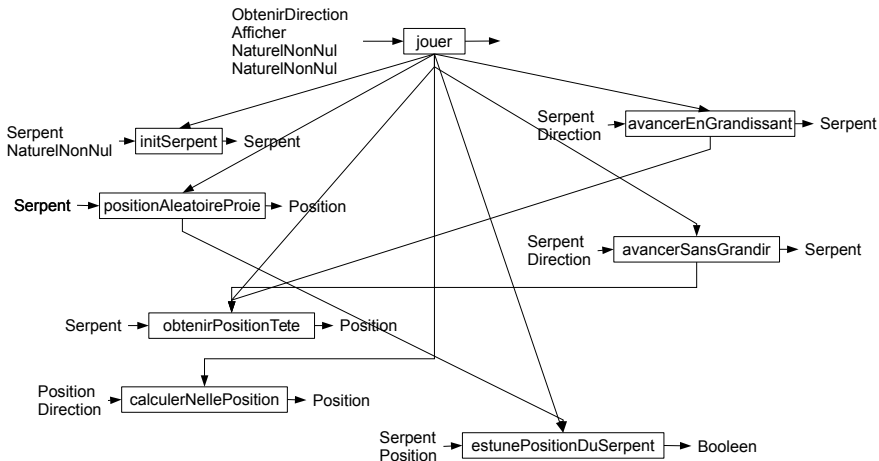
## Types

- On suppose posséder les types suivants :
  - Pour les entités du jeu : Serpent, Position, Direction
  - Pour la séparation de l'IHM et de la logique métier :
    - ObtenirDirection
    - Afficher



## Analyse 2 / 2

## Analyse descendante



## L'air de jeu 1 / 2

## Conception

- Soit les types `Direction` et `Position` définis de la façon suivante :

**Type** `Direction` = {nord,est,ouest,sud}

**Type** `Position` = **Structure**

`x` : **NaturelNonNul**

`y` : **NaturelNonNul**

**finstructure**

- L'air de jeu est caractérisée par quatre constantes `XMIN`, `XMAX`, `YMIN`, `YMAX` (`(XMIN, YMIN)` en haut à gauche (`XMAX, YMAX`) en bas à droite)

## On doit donner l'algorithme de :

- **fonction** `calculerNouvellePosition` (`p` : `Position`, `d` : `Direction`) : `Position`

## L'air de jeu 2 / 2

## Fonction nouvellePosition

**fonction** nouvellePosition (p : Position, d : Direction) : Position

**Déclaration** resultat : Position

**debut**

resultat ← p

**cas où d vaut**

*est:*

resultat.x ← resultat.x + 1

**si** resultat.x > XMAX **alors**

resultat.x ← XMIN

**finsi**

*ouest:*

resultat.x ← resultat.x - 1

**si** resultat.x < XMIN **alors**

resultat.x ← XMAX

**finsi**

*nord:*

resultat.y ← resultat.y - 1

**si** resultat.y < YMIN **alors**

resultat.y ← YMAX

**finsi**

*sud:*

resultat.y ← resultat.y + 1

**si** resultat.y > YMAX **alors**

resultat.y ← YMIN

**finsi**

**fincas**

**retourner** resultat

**fin**

## Le serpent 1 / 7

## Conception

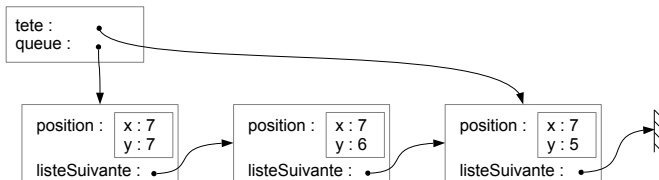
On se propose de représenter le serpent à l'aide d'une liste chaînée de positions, tel que la tête de la liste représentera la position de la queue du serpent et le dernier élément de la liste représentera la position de la tête du serpent. Ainsi on représente le type `Serpent` de la façon suivante :

**Type** Serpent = **Structure**

tete : ListeChainePosition

queue : ListeChainePosition

**finstructure**



## Le serpent 2 / 7

On doit donner l'algorithme de :

- **fonction** obtenirPositionTete (s : Serpent) : Position
- **fonction** obtenirPositionQueue (s : Serpent) : Position
- **procédure** avancerEnGrandissant (**E/S** s : Serpent, **E** d : Direction)
- **procédure** avancerSansGrandir (**E/S** s : Serpent, **E** d : Direction)
- **procédure** initSerpent (**E/S** s : Serpent, **E** tailleInitiale : NaturelNonNul)
  - └ **précondition**(s) tailleInitiale > 1 et  
tailleInitiale ≤ (YMAX-YMIN) div 2
- **fonction** estUnePositionDuSerpent (s : Serpent, p : Position) : Booleen

## Le serpent 3 / 7

## Fonction obtenirPositionTete

```
fonction obtenirPositionTete (s : Serpent) : Position
debut
    retourner obtenirPosition(s.tete)
fin
```

## Fonction obtenirPositionQueue

```
fonction obtenirPositionQueue (s : Serpent) : Position
debut
    retourner obtenirPosition(s.queue)
fin
```

## Le serpent 4 / 7

## Fonction avancerEnGrandissant

**procédure** avancerEnGrandissant (**E/S** s : Serpent, **E** d : Direction)

**Déclaration** temp : ListeChainePosition

**debut**

temp ← listeVide()

ajouter(temp, nouvellePosition(obtenirPositionTete(s), d))

fixerListeSuivante(s.tete, temp)

s.tete ← temp

**fin**

## Le serpent 5 / 7

## Fonction avancerSansGrandir

**procédure** avancerSansGrandir (**E/S** s : Serpent, **E** d : Direction)**debut**

avancerEnGrandissant(s,d)

supprimerTete(s.queue)

**fin**



## Le serpent 6 / 7

Fonction `initSerpent`

**procédure** `initSerpent` (**E/S** `s` : Serpent, **E** `tailleInitiale` : **NaturelNonNul**)

**[précondition(s)]** `tailleInitiale > 1` et `tailleInitiale ≤ (YMAX-YMIN) div 2`

**Déclaration** `p` : Position  
`i` : **Naturel**

**debut**

`p.x` ←  $(XMAX+XMIN) \text{ div } 2$

`p.y` ←  $(YMAX+YMIN) \text{ div } 2$

`s.tete` ← `listeVide()`

`ajouter(s.tete,p)`

`s.queue` ← `s.tete`

**pour** `i` ← -1 à `tailleInitiale-1` **faire**  
    `avancerEnGrandissant(s,nord)`

**finpour**

**fin**

## Le serpent 7 / 7

## Fonction estUnePositionDuSerpent

**fonction** estUnePositionDuSerpent (s : Serpent, p : Position) : **Booleen**

**Déclaration** temp : ListeChainePosition  
trouve : **Booleen**

**debut**

temp ← s.queue

trouve ← FAUX

**tant que** non estVide(temp) et non trouve **faire**

**si** obtenirPosition(temp)=p **alors**

    trouve ← VRAI

**sinon**

    temp ← obtenirListeSuiivante(temp)

**finsi**

**fintantque**

**retourner** trouve

**fin**

## Les proies 1 / 2

## Conception

On possède la fonction suivante qui permet d'obtenir un naturel aléatoire compris entre deux bornes.

- **fonction** naturelAleatoire (borneInf, borneSup : **Naturel**) : **Naturel**

└ **précondition(s)** borneInf < borneSup

On doit donner l'algorithme de :

- **fonction** positionAleatoireProie (s : Serpent) : Position

## Les proies 2 / 2

## Fonction positionAleatoireProie

**fonction** positionAleatoireProie (s : Serpent) : Position

**Déclaration** resultat : Position

**debut**

**repete**

resultat.x  $\leftarrow$  naturelAleatoire(XMIN,XMAX)

resultat.y  $\leftarrow$  naturelAleatoire(YMIN,YMAX)

**jusqu'à ce que** non estUnePositionDuSerpent(s,resultat)

**retourner** resultat

**fin**

## Le jeu 1 / 3

## Séparation IHM/Logique métier

On suppose posséder :

- **Type** ObtenirDirection = **fonction**(d : Direction) : Direction
- **Type** Afficher = **procédure**(E s : Serpent ; positionProie : Position ; score : **Naturel**)

## Procédure jouer

**procédure** jouer (oDir : ObtenirDirection, aff : Afficher, tailleInitiale, tailleAgrandissementSiProieMange : **NaturelNonNul**)

  | **précondition**(s)  tailleInitiale > 1 et tailleInitiale ≤ (YMAX-YMIN) div 2

**Déclaration**  s : Serpent  
                   pProie, p : Position  
                   tailleAgrandissementRestant, score : **Naturel**  
                   dir : Direction  
                   jeuFini : **Booleen**

**debut**

  initSerpent(s, tailleInitiale)  
   tailleAgrandissementRestant ← 0  
   score ← 0  
   dir ← nord  
   pProie ← positionAleatoireProie(s)  
   jeuFini ← FAUX  
   *voir boucle principale (transparent suivant)*  
   supprimerSerpent(s)

**fin**

## Le jeu 2 / 3

```

tant que non jeuFini faire
  aff(s,pProie,score)
  dir ← oDir(dir)
  p ← calculerNellePosition(obtenirPositionTete(s),dir)
  si estUnePositionDuSerpent(s,p) alors
    jeuFini ← VRAI
  sinon
    si p = pProie alors
      pProie ← positionAleatoireProie(s)
      tailleAgrandissementRestant ← tailleAgrandissementRestant + tailleAgrandissementSiProieMange
      score ← score + 1
    finsi
    si tailleAgrandissementRestant > 0 alors
      avancerEnGrandissant(s,dir)
      tailleAgrandissementRestant ← tailleAgrandissementRestant - 1
    sinon
      avancerSansGrandir(s,dir)
    finsi
  finsi
fintantque

```

# Le jeu 3 / 3

## Ce qu'il reste à faire

- Écrire les fonctions et procédures de l'IHM :
  - pour obtenir la prochaine direction
  - pour afficher l'état du jeu
- Le programme principal

## Ce qui pourrait être amélioré

- Avoir un type AireDeJeu (pour éviter les constantes)