



Université Paris XI
I.U.T. d'Orsay
Département Informatique
Année scolaire 2003-2004

Algorithmique : Volume 4

- Listes chaînées
- Piles et files

Retour au type tableau

- Avantages
 - Accès direct à un élément quelconque du tableau par un indice
 - Souplesse dans l'écriture des algorithmes
 - Algorithmes de recherche très performants quand les éléments sont ordonnés

Retour au type tableau (suite)

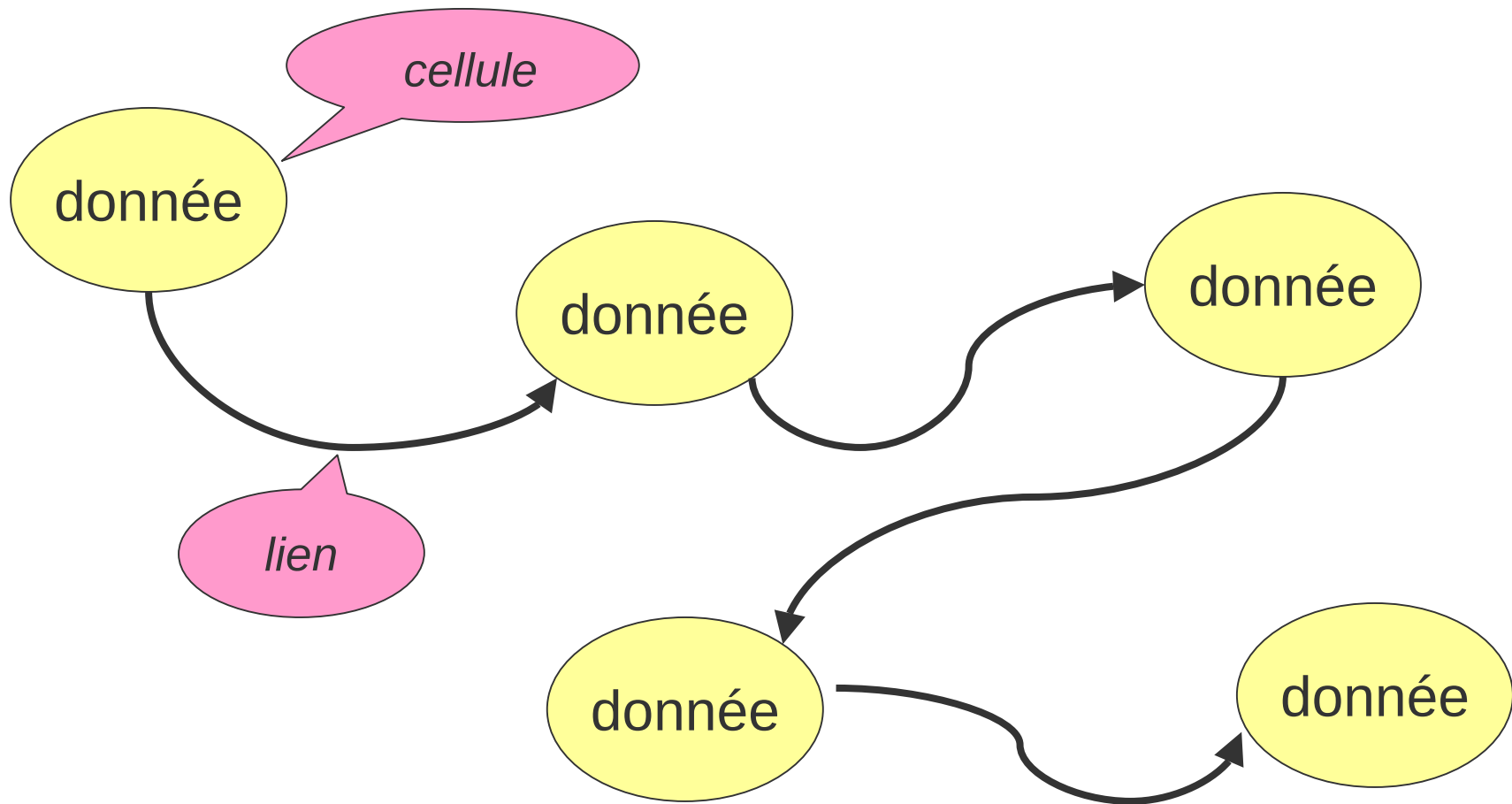
- Inconvénients

- Nécessité de définir la dimension maximale allouée dès la déclaration, c'est à dire de façon statique
- Lourdeur dans la gestion optimale de l'espace occupé par les données
- Nécessité d'effectuer des contrôles de débordement à chaque étape du traitement

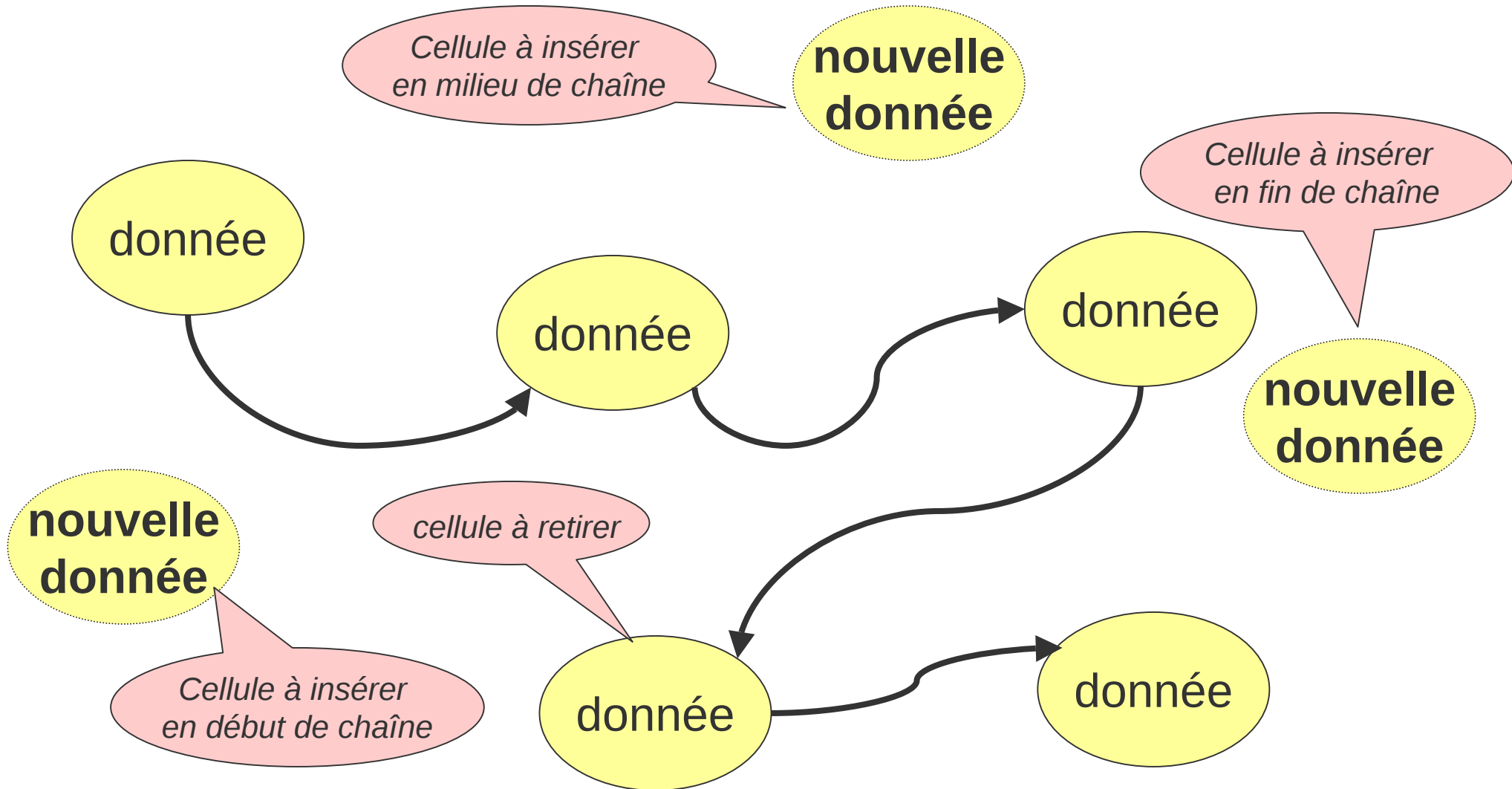
Cahier des charges d'une représentation moins contrainte

- Permettre l'allocation de mémoire en fonction des besoins, de façon dynamique
- Faciliter la gestion de la mémoire occupée en cas d'insertion ou de suppression de nouvelles données
- Simulation de phénomènes du monde physique mal représentés par la structure en tableaux
- Exemples de situations :
 - File d'attente à un guichet
 - Urgences d'un hôpital
 - Distribution de cartes à une table de joueurs
 - Gestion des dossiers empilés sur un bureau

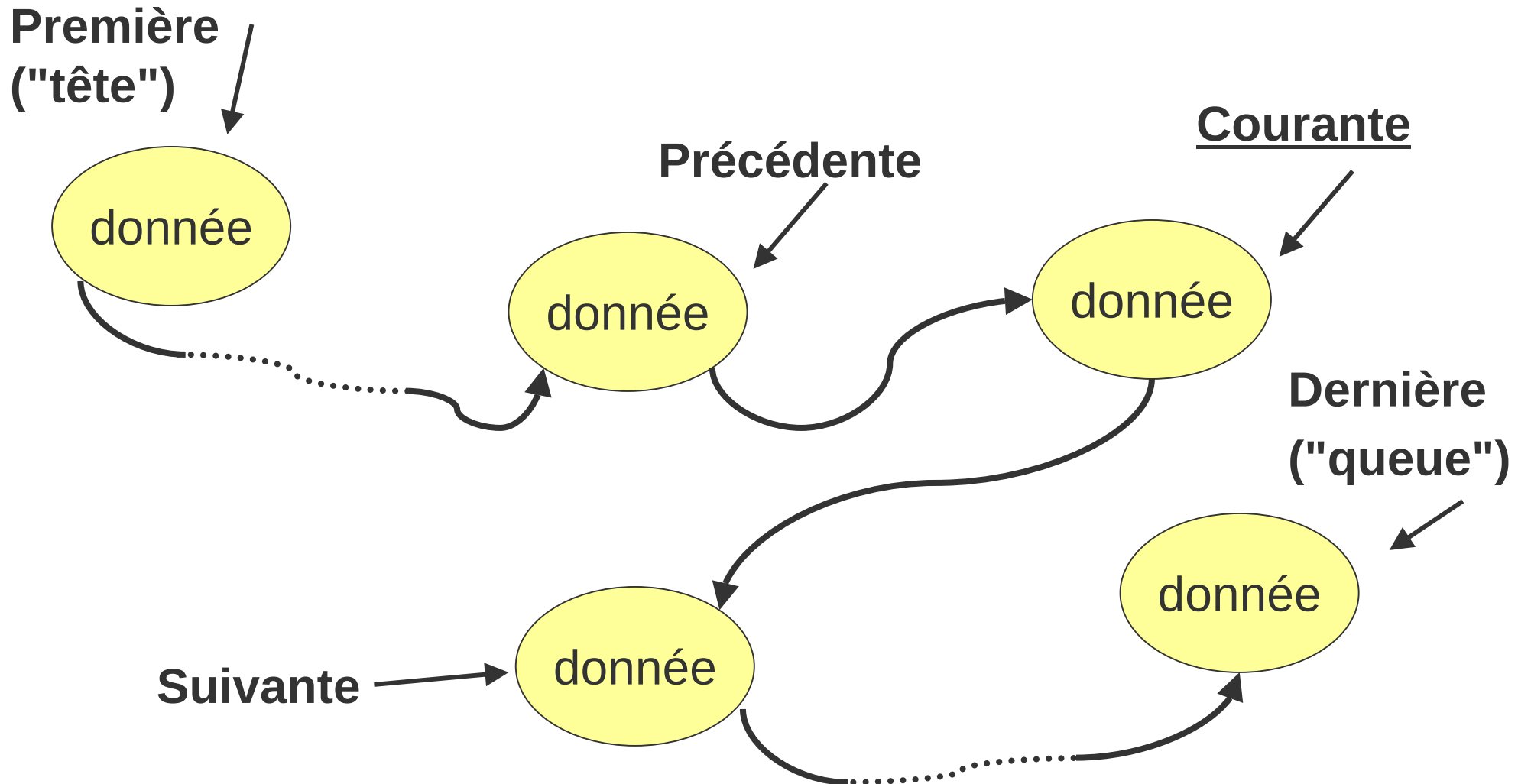
La notion de liste chaînée



Insertion/suppression d'une donnée



Repérage des cellules d'une liste chaînée



Exemples de traitements opérés sur les listes (1)

- Traitements relatifs à la composition structurelle de la liste :
 - Positionnement sur la première cellule de la structure
 - Positionnement sur la dernière cellule de la structure
 - Calcul de la longueur d'une liste (nombre de cellules)
 - Reconnaissance d'une liste vide
 - Déplacement du positionnement courant sur la cellule suivante

Exemples de traitements opérés sur les listes (2)

- Traitements relatifs à l'information enregistrée dans une liste :
 - Enregistrement de données jusqu'à épuisement du flot de données
 - Visualisation de l'information enregistrée dans une cellule, quelle que soit sa place dans la liste
 - Visualisation de l'ensemble des informations enregistrées dans la liste
 - Suppression d'une cellule; ajout d'une cellule

Cahier des charges pour une classe Liste

- Les attributs de la classe Liste doivent permettre:
 - le positionnement sur les différentes cellules de la liste
 - la définition du type d'information enregistrée dans la cellule
- Les méthodes de la classe doivent permettre tous les traitements décrits précédemment (et plus...)

Définition de la classe Liste

classe Liste

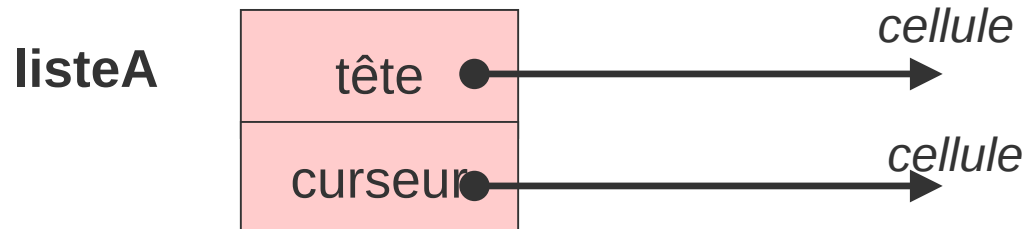
Attributs :

tête : référence {référence à la cellule tête de liste}

curseur : référence {référence à la cellule courante}

référence : type dont le domaine de définition est l'ensemble des adresses mémoire.

- Valeur de l'attribut **tête** (attribut **curseur**) : adresse de la case mémoire où est stockée la cellule de tête de liste (cellule courante)
- Représentation graphique : une flèche



Définition d'une cellule

type Cellule = agrégat

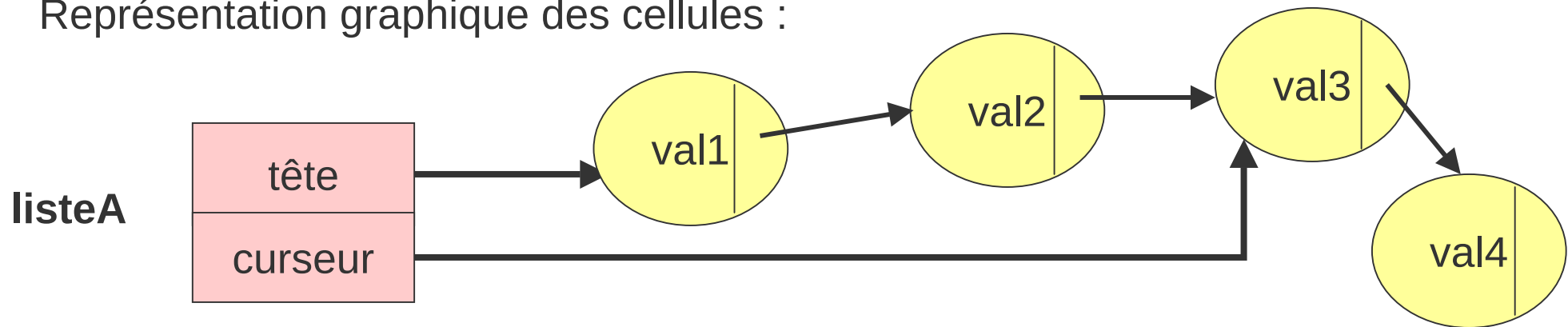
val : Info *{information stockée dans une cellule}*

suivant : référence *{référence à une autre cellule}*

fin

Info : le type de l'information stockée; peut-être un type de base (par exemple, un entier) ou bien un type complexe (un agrégat)

Représentation graphique des cellules :



Méthodes de la classe Liste

MProcédure **suivant()**

{place le curseur sur la cellule qui suit la cellule courante.

Si le curseur était sur la dernière cellule, il devient hors liste. Erreur si la liste est vide ou si le curseur est déjà hors liste.}

paramètre (D/R) cible : Liste

exemple :

liste initiale **L1**

après **L1.suivant()**

Méthodes (suite)

MProcédure **premier()**

{place le curseur sur la première cellule de la liste.

Si la liste est vide, le curseur reste hors liste.}

paramètre (D/R) cible : Liste

MProcédure **dernier()**

{place le curseur sur la dernière cellule de la liste.

Si la liste est vide, le curseur reste hors liste.}

paramètre (D/R) cible : Liste

MFonction **vide()** retourne booléen

{retourne vrai si la liste est vide, faux sinon}

paramètre (D) cible : Liste

Méthodes (suite)

MFonction horsListe() retourne booléen

{retourne vrai si le curseur est placé hors liste ou si la liste est vide, faux sinon.}

paramètre (D) cible : Liste

MFonction info() retourne Info

*{retourne la valeur enregistrée dans la cellule courante.
Erreur si la liste est vide ou si le curseur est hors liste.}*

paramètre (D) cible : Liste

MProcédure affecter(val)

*{affecte la valeur val à la cellule courante.
Erreur si la liste est vide ou si le curseur est hors liste.}*

paramètres (D/R) cible : Liste
(D) val : Info

Méthodes (suite)

MProcédure **insérerAvant(val)**

{crée une nouvelle cellule, y affecte la valeur val, et l'insère avant la cellule courante. Le curseur est alors placé sur cette nouvelle cellule qui devient ainsi la nouvelle cellule courante. Si le curseur était placé sur la tête, la nouvelle cellule devient la nouvelle tête. Si la liste était vide, elle contient maintenant l'unique cellule qui vient d'être créée.

Erreur si la liste était non vide et le curseur hors liste.}

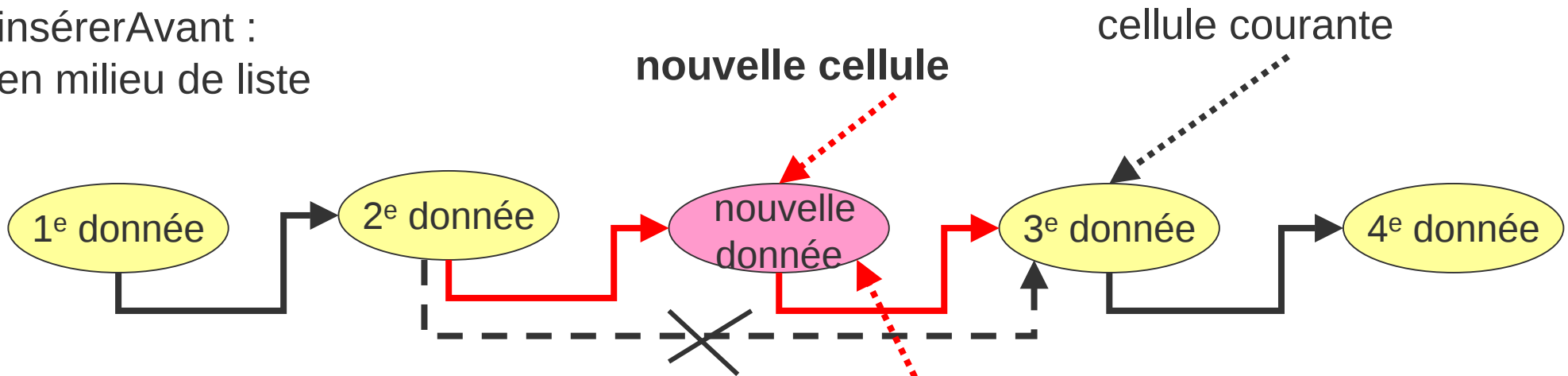
paramètres (D/R) cible : Liste
(D) val : Info

Remarque :

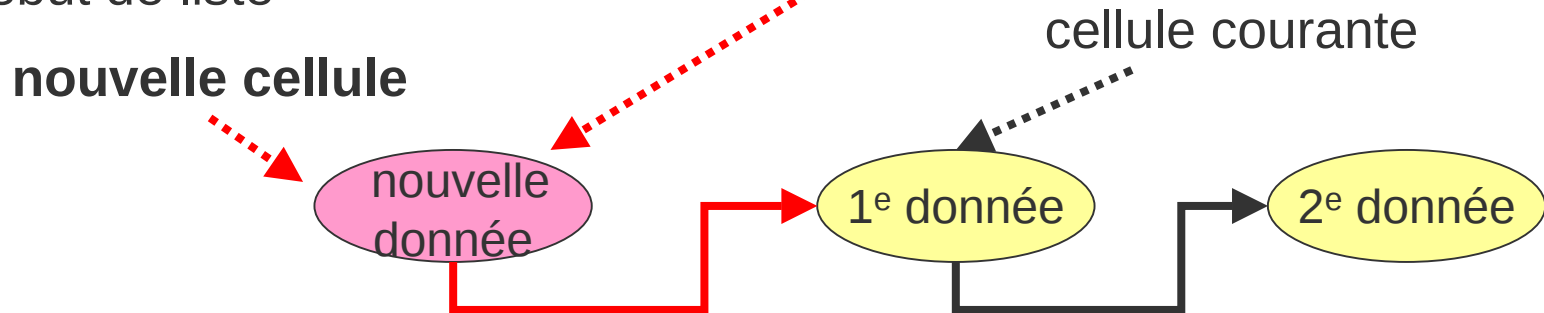
premier() suivi de **insérerAvant(val)** $\Rightarrow$ ajouter en tête

Insertion d'une cellule : insérerAvant

insérerAvant :
en milieu de liste



insérerAvant :
en début de liste



Méthodes (suite)

MProcédure **insérerAprès(val)**

{crée une nouvelle cellule, y affecte la valeur val, et l'insère après la cellule courante. Le curseur est alors placé sur cette nouvelle cellule qui devient ainsi la nouvelle cellule courante. Si liste était vide, elle contient maintenant l'unique cellule qui vient d'être créée. Erreur si la liste était non vide et le curseur hors liste.}

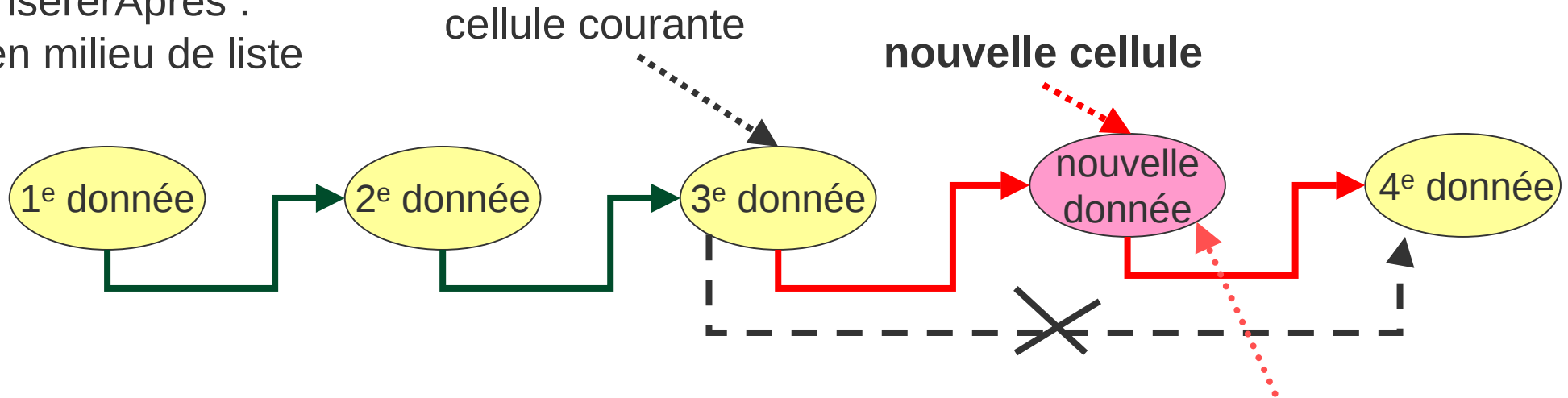
paramètres (D/R) cible : Liste
(D) val : Info

Remarque :

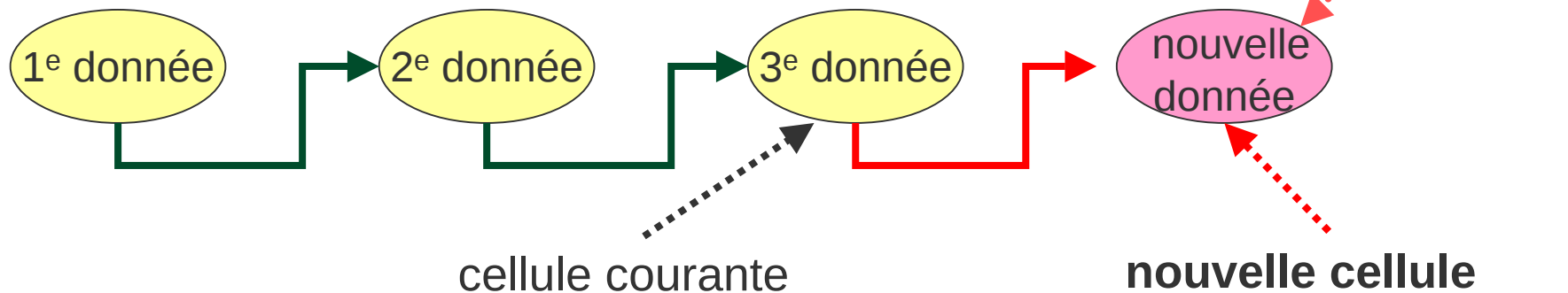
dernier() suivi de **insérerAprès(val)** $\Rightarrow$ ajouter en queue

Insertion d'une cellule : insérerAprès

insérerAprès :
en milieu de liste



insérerAprès :
en fin de liste



Méthodes (suite)

MProcédure **supprimer()**

{supprime la cellule courante. Le curseur est alors placé sur la cellule suivante qui devient ainsi la nouvelle cellule courante. Si la cellule à supprimer est la dernière cellule, le curseur devient hors liste. Si la cellule à supprimer est la tête, la cellule suivante devient alors la nouvelle tête. Si la liste ne contenait qu'une seule cellule, la liste devient vide. Erreur si la liste est vide ou si le curseur est hors liste.}

paramètre (D/R) cible : Liste

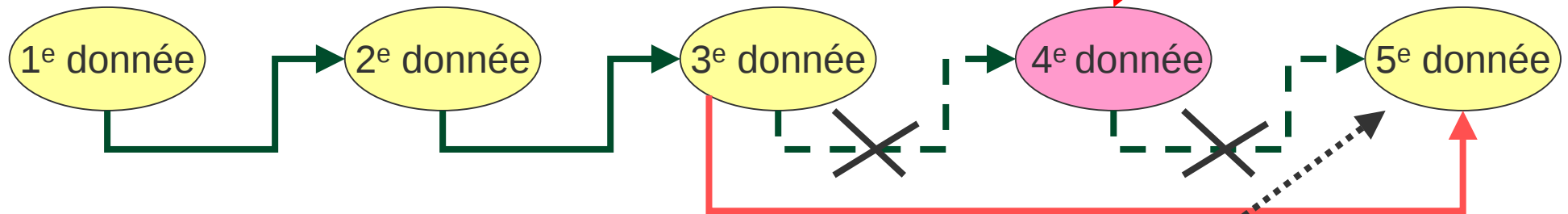
Remarque

premier() suivi de **supprimer()** $\Rightarrow$ supprimer en tête

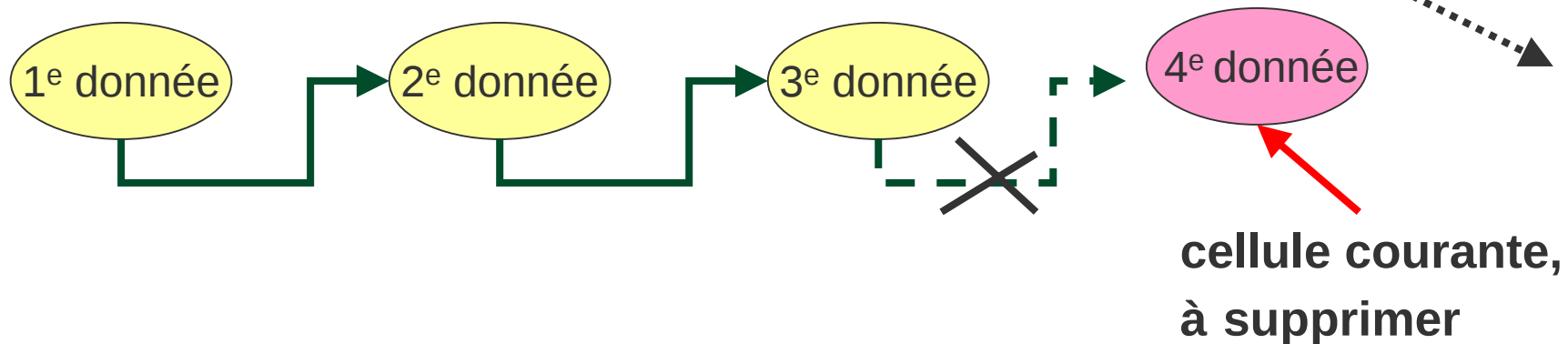
dernier() suivi de **supprimer()** $\Rightarrow$ supprimer en queue

Suppression d'une cellule

supprimer :
en milieu de liste



supprimer :
en fin de liste



Méthodes (suite et fin)

MProcédure saisirListe()

{Saisit des valeurs (de type Info), jusqu'à une valeur d'arrêt, et crée au fur et à mesure autant de cellules que nécessaire, en y affectant les valeurs saisies. Le curseur est ensuite replacé en tête de liste.}

paramètre (R) cible : Liste

MProcédure afficherListe()

{Affiche toutes les valeurs contenues dans la liste cible.}

paramètre (D) cible : Liste

MProcédure supprimerTout()

{supprime toutes les cellules de la liste (qui peut être vide); la liste devient vide et le curseur devient hors liste.}

paramètre (D/R) cible : Liste

Remarques importantes

- Méthodes de deux types :
 - méthodes “de base”: leur définition nécessite de modifier les attributs privés de la classe (sera fait en C++).
 - **saisirListe()**, **afficherListe()**, **supprimerTout()** : méthodes rajoutées à la classe pour rendre son utilisation plus commode; leur définition peut se faire à l’aide des méthodes de base.
- Rappel : les 3 “acteurs” de la programmation objet :
 - programmeur concepteur de la classe
 - programmeur utilisateur de la classe
 - utilisateur de l’application

Saisie d'une liste

MProcédure **saisirListe()**

{Saisit des valeurs (de type Info), jusqu'à une valeur d'arrêt (constante définie dans l'algorithme appelant), et crée au fur et à mesure autant de cellules que nécessaire, en y affectant les valeurs saisies. Le curseur est ensuite replacé en tête de liste.}

paramètre (R) cible : Liste

variables uneVal : Info

cpt : entier

début

saisir(uneVal) ; cpt ← 0

tant que uneVal ≠ valStop **faire**

cpt ← cpt + 1

cible.insérerAprès(uneVal)

saisir(uneVal)

ftq

cible.premier() *{replace le curseur en tête}*

afficher("La nouvelle liste contient ", cpt, " cellules.")

fin

Saisie d'une liste : simulation

Affichage d'une liste

MProcédure **afficherListe()**

{Affiche toutes les valeurs contenues dans la liste cible.}

paramètre (D) cible : Liste

variable copieCible : Liste

début

copieCible ← cible

*{copie de la cible,
pour permettre modification du curseur}*

copieCible.premier()

{place le curseur en tête}

tant que non **copieCible.horsListe()** **faire**

{arrêt quand curseur hors liste}

afficher(**copieCible.info()**)

*{récupère la valeur de la cellule
courante et l'affiche}*

copieCible.suivant()

*{place le curseur sur la cellule
suivante}*

ftq

fin

Exemple d'algorithme (1)

Algorithme ManipListes1

{Saisie et affichage d'une liste.}

constante (VALSTOP : entier) $\leftarrow$ 0

variable listeA : Liste

début

 listeA.saisirListe()

 afficher("Liste saisie : ")

 listeA.afficher()

fin

Exemple de fonction utilisant la classe liste (1)

Fonction valMin(uneListe) retourne réel

{retourne la plus petite valeur contenue dans une liste de réels supposée non vide}

paramètre (D) uneListe : ListeRéal

Exemple de fonction utilisant la classe liste (2)

Fonction inverse(uneListe) retourne Liste

{crée une nouvelle liste, en y affectant les valeurs de la liste uneListe mais dans l'ordre inverse. La liste uneListe est supposée non vide}

paramètre (D) uneListe : Liste

Exemple de fonction utilisant la classe liste (3)

Procédure supprimerGlobal(uneVal, uneListe)

{supprime toutes les cellules de la liste uneListe qui contiennent la valeur uneVal. Replace le curseur en tête.}

paramètre (D/R) uneListe : Liste
 (D) uneVal : Info

Exemple d'algorithme (2)

Algorithme ManipListes2

{Exemple d'algorithme manipulant les listes.}

constante (VALSTOP : entier) $\leftarrow$ 0

variables listeA, listeB, listeC : Liste

début

 listeA.saisir()

afficher("listeA: ")

 listeA.afficher()

si non listeA.vide()

alors listeB $\leftarrow$ inverse(listeA)

afficher("listeB contient les éléments de la listeA en ordre
 inverse : ")

 listeB.afficher()

 supprimerGlobal(0, listeA)

afficher("ListeA sans zéros: ")

 listeA.afficher()

afficher("Plus petite valeur de listeA : " , valMin(listeA))

fsi

fin

Files et Piles

Retour sur la motivation: pourquoi des listes chaînées?

- Possibilité de croître ou de diminuer selon les besoins
- Facilité de réordonnancement des éléments

Exemples :

- placer le dernier élément en tête : changer trois références
(tableau : tout décaler)
 - insertion d'un nouvel élément : changer deux références,
indépendamment de la longueur de la liste
 - effacement d'un élément
- **Mais : mal adaptées à d'autres opérations**
 - trouver le k-ième élément : parcours séquentiel de k références
(tableau : accès direct à l'indice k)
 - trouver l'élément qui précède un élément donné

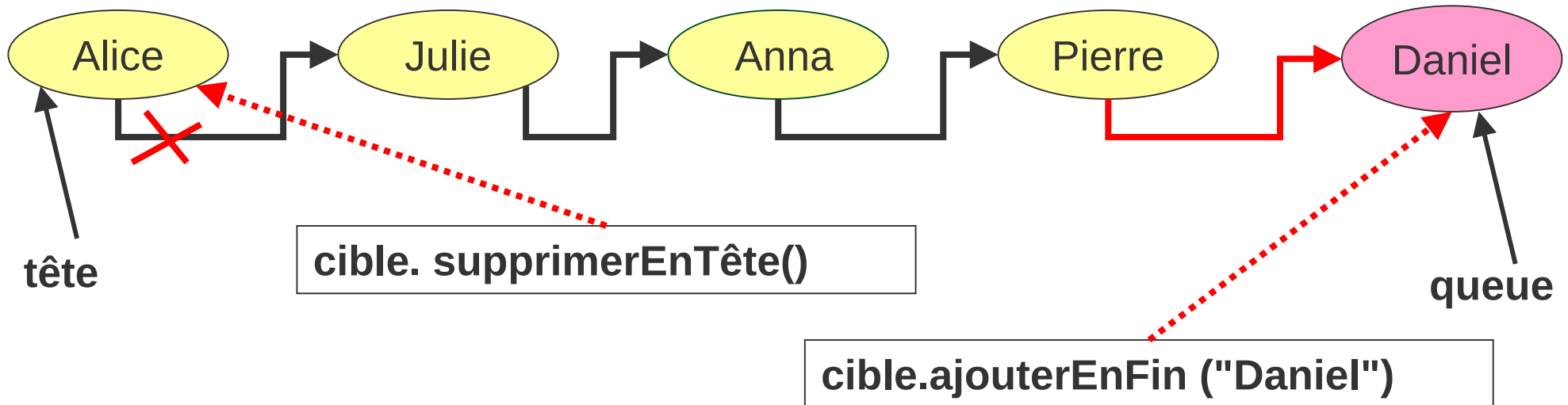
Files et Piles

Dans beaucoup d'applications, on peut se contenter de modes d'accès très restreints à la structure de données

Avantages :

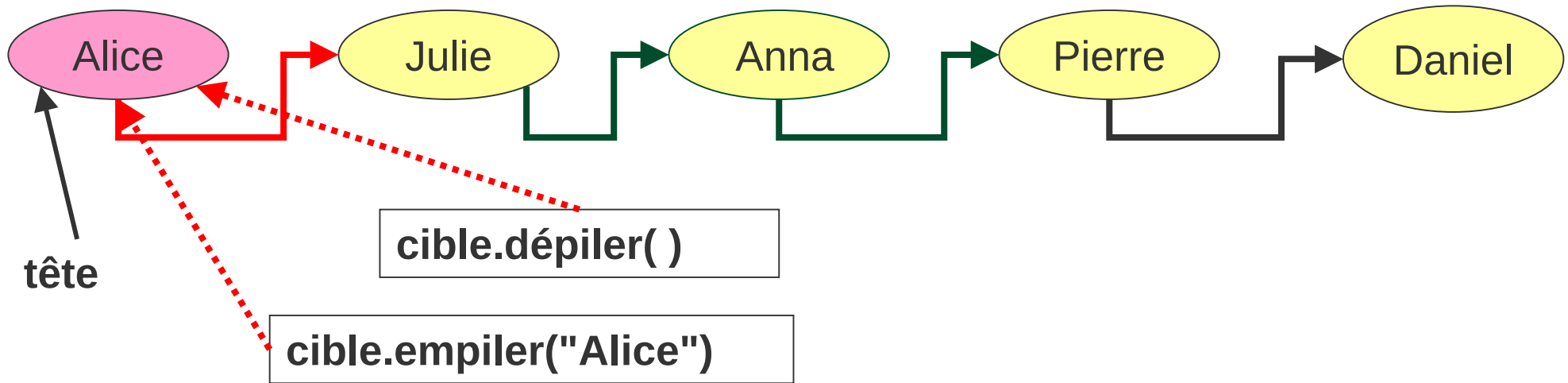
- le programme n'a pas à se préoccuper de détails de gestion (des références, par exemple)
- traitements plus simples et moins rigides (moins d'opérations)

Représentation d'une file par une liste chaînée



- les ajouts se font en fin de file, les suppressions en tête de file
 - seule l'information de la tête est accessible et traitable
- file d'attente à un guichet "premier rentré, premier sorti"
(FIFO : first in first out, queue)

Représentation d'une pile par une liste chaînée



- les ajouts comme les suppressions se font en tête de pile
 - seule l'information de la tête est accessible et traitable
- pile d'assiettes "dernier rentré, premier sorti"
(LIFO : last in first out, stack)

la classe File :

les besoins

Attributs :

- la tête et la queue, mais pas de curseur

Méthodes :

- **infoTête()** : retourne la valeur de l'information en tête de file
- **vide()** : indique si la file est vide
- **ajouterEnFin(val)** : ajoute une information en fin de file
- **supprimerEnTête()** : supprime (et retourne) l'information en tête de file
- **saisirFile()**
- **afficherFile()**

Définition de la classe File

Attributs :

tête : référence {référence à la tête de file}
queue : référence {référence à la queue de file}

Méthodes

Mfonction infoTête() retourne Info

{retourne la valeur enregistrée dans la cellule de tête. Erreur si la file est vide}

paramètre (D) cible : File

Mfonction vide() retourne booléen

{retourne vrai si la file est vide, faux sinon}

paramètre (D) cible : File

Mprocédure ajouterEnFin(val)

*{Crée une nouvelle cellule, y affecte la valeur val, et l'insère après la dernière cellule.
Si file était vide, elle contient maintenant l'unique cellule qui vient d'être créée.}*

paramètre (D/R) cible : File ; (D) val : Info

Mfonction supprimerEnTête() retourne Info

{Supprime la première cellule de la file et retourne la valeur qu'elle contient. Si la file ne contenant qu'une seule cellule, la file devient vide. Erreur si la file est vide.}

paramètre (D/R) cible : File

Méthodes (suite)

MProcédure saisirFile ()

{Saisit des valeurs (de type Info), jusqu'à une valeur d'arrêt (constante définie dans l'algorithme appelant), et crée au fur et à mesure autant de cellules que nécessaire, en y affectant les valeurs saisies.}

paramètre (R) cible : File

variables uneVal : Info, cpt : entier

début

saisir(uneVal) ; cpt ← 0

tant que uneVal ≠ VALSTOP **faire**

cpt ← cpt + 1

cible.ajouterEnFin(uneVal)

saisir(uneVal)

ftq

afficher("La nouvelle file contient ", cpt, " cellules.")

fin

Méthodes (suite)

MProcédure afficherFile()

{Affiche toutes les valeurs contenues dans la file cible.}

paramètre (D) cible : File

variables uneVal : Info
copieCible : File

début

copieCible ← cible

tant que non copieCible.vide() **faire**

uneVal ← copieCible.**supprimerEnTête()**

afficher(uneVal)

ftq

fin

la classe Pile :

les besoins

Attributs :

- la tête mais pas de curseur

Méthodes :

- **infoTête()** : retourne la valeur de l'information en tête de pile
- **vide()** : indique si la pile est vide
- **empiler(val)** : ajoute une information en tête de pile
- **dépiler()** : supprime (et retourne) l'information en tête de pile
- **saisirPile()**
- **afficherPile()**

Définition de la classe Pile

Attributs :

tête : référence {référence à la tête de pile}

Méthodes

MFonction infoTête() retourne Info

{retourne la valeur enregistrée dans la cellule de tête. Erreur si la pile est vide}

paramètre (D) cible : Pile

Mfonction vide() retourne booléen

{retourne vrai si la pile est vide, faux sinon}

paramètre (D) cible : Pile

Mprocédure empiler(val)

{Crée une nouvelle cellule, y affecte la valeur val, et l'insère en tête de pile. Si pile était vide, elle contient maintenant l'unique cellule qui vient d'être créée.}

paramètre (D/R) cible : Pile ; (D) val : Info

Mfonction dépiler() retourne Info

{Supprime la première cellule de la pile et retourne la valeur qu'elle contient. Si la pile ne contenant qu'une seule cellule, la pile devient vide. Erreur si la pile est vide.}

paramètre (D/R) cible : Pile

Méthodes (suite)

MProcédure **saisirPile()**

{Saisit des valeurs (de type Info), jusqu'à une valeur d'arrêt (constante définie dans l'algorithme appelant), et crée au fur et à mesure autant de cellules que nécessaire, en y affectant les valeurs saisies}

paramètre (R) **cible : Pile**

variables uneVal : Info

cpt : entier

début

saisir(uneVal) ; cpt $\leftarrow$ 0

tant que uneVal $\neq$ VALSTOP **faire**

 cpt $\leftarrow$ cpt + 1

cible.empiler(uneVal)

saisir(uneVal)

ftq

afficher("La nouvelle pile contient", cpt, "cellules.")

fin

Méthodes (suite)

MProcédure afficherPile()

{Affiche toutes les valeurs contenues dans la pile cible.}

paramètre (D) cible : Pile

variables uneVal : Info
copieCible : Pile

début

copieCible ← cible

tant que non copieCible.vide() **faire**

uneVal ← copieCible.dépiler()

afficher(uneVal)

ftq

fin

Exemple 1 : Parenthésage

Fonction bienFormé(tab, nbr) retourne **booléen**

{tab est un tableau de nbr parenthèses. Retourne vrai si le parenthésage est cohérent}

paramètres (D) tab: **tableau**[1, MAX] de **caractères**; nbr : **entier**

variables cpt, marque, val : **entier**; bienformé : **booléen**
unePile : **Pile** *{unePile est vide au départ}*

début

cpt ← 1; marque ← 0 *{jeton}* ; bienformé ← vrai

tant que bienformé **et** cpt ≤ nbr **faire**

si tab[cpt] = '('

alors unePile.empiler(marque)

sinon si unePile.vide() **alors** bienformé ← faux

sinon val ← unePile.dépiler() *{on a alors ')'}*

fsi

fsi

 cpt ← cpt +1

ftq

retourne (bienformé **et** unePile.vide())

fin

Parenthésage : simulation

Exemple 2: Évaluation d'une expression arithmétique

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

OBJECTIF :

1. empiler (5)
2. empiler (9)
3. empiler (8)
4. empiler(dépiler () + dépiler())
5. empiler (4)
6. empiler (6)
7. empiler (dépiler () * dépiler())
8. empiler (dépiler () * dépiler())
9. empiler (7)
10. empiler (dépiler () + dépiler())
11. empiler (dépiler () * dépiler())
12. afficher(dépiler())

Évaluation : simulation

D'abord : conversion infix $\rightarrow$ postfix

Procédure **infixVersPostfix**(tab, nbr, tab_res, nbr_res)

paramètres (D) tab : **tableau**[1, MAX] de caractères

(D) nbr : **entier**

(R) tabRes: **tableau**[1, MAX] de caractères

(R) nbrRes : **entier**

variables i, j : **entier** ; unePile : **Pile**

début

i $\leftarrow$ 0 ; j $\leftarrow$ 0

tant que (i < nbr)

i $\leftarrow$ i + 1

si opérateur(tab[i])

alors unePile.empiler (tab[i])

sinon si tab[i] = ')'

alors j $\leftarrow$ j + 1; tabRes[j] $\leftarrow$ unePile .dépiler() ;

sinon si nombre(tab[i]) **alors** j $\leftarrow$ j+1 ; tabRes[j] $\leftarrow$ tab[i]

fsi

fsi

ftq

nbrRes $\leftarrow$ j

fin

Conversion : simulation

Ensuite : calculer postfix

Fonction **calculerPostfix**(tab, nbr) retourne(entier)

paramètres (D) tab: **tableau**[1, MAX] de caractères

(D) nbr : **entier**

variables i, res : **entiers**

unePile : **Pile**

début

i ← 1; res ← 0

tant que (i ≤ nbr) **faire**

si tab[i] = '+'

alors res ← unePile.dépiler() + unePile.dépiler()

sinon si tab[i] = '-'

alors res ← unePile.dépiler() - unePile.dépiler()

sinon si tab[i] = '*'

alors res ← unePile.dépiler() * unePile.dépiler()

sinon si nombre(tab[i]) **alors** res ← tab[i]

fsi

 unePile.empiler (res)

 i ← i+1

ftq

retourne(unePile .dépiler())

fin

Calculer postfix : simulation

Retour à l'évaluation d'une expression arithmétique

Algorithme évaluation

{affiche l'évaluation d'une expression arithmétique}

variables	explInfix : tableau [1, MAX] de caractères	{expression en écriture infixe}
	expPostfix : tableau [1, MAX] de caractères	{expression en écriture postfixe}
	nbrEltsInfix : entier	{nombre d'éléments de l'expression infixe}
	nbrEltsPostfix : entier	{nombre d'éléments de l'expression postfixe}
	valeur : entier	{résultat de l'évaluation}

début

```
saisirExpInfix(expInfix, nbrEltInfix )
```

infixVersPostfix(expInfix, nbrEltsInfix , expPostfix , nbrEltsPostfix)

```
valeur ← calculerPostfix(expPostfix , nbrEltsPostfix )
```

afficher(« L'évaluation de votre expression est », valeur)

fin

Fin Volume 4