

# Cours n°6 : liste chaînée et récursivité

Alain Giorgetti

[giorgetti@univ-fcomte.fr](mailto:giorgetti@univ-fcomte.fr)

<http://lifc.univ-fcomte.fr/~giorgett/>

Laboratoire d'Informatique  
de l'Université de Franche-Comté



# Motivations

- Faiblesses de la structure de table
  - On construit un tableau de **B** objets, pour n'en stocker que **n**, avec **n** inférieur à **B**
  - Lorsque **B** objets sont stockés, on ne peut plus rien ajouter
  - Lors de suppressions et d'ajouts dans un tableau trié, il faut faire de nombreux décalages
- Structure collective plus **dynamique** : la **liste chaînée**
  - Construction en mémoire dynamique
  - Chaque élément (sauf le dernier) donne accès au suivant
    - ◆ par un **pointeur sur l'élément suivant** de la liste
    - ◆ le suivant **est unique**, s'il existe



# Structure de liste

- Définition d'une *liste*
  - Collection ordonnée finie d'éléments de même type
  - Nombre d'éléments **non borné** (dans le modèle)
  - Un élément est accessible à partir de l'élément précédent
- **Méthodes** disponibles
  - accéder au **premier** élément de la liste
  - accéder à l'élément qui **suit immédiatement** un élément connu de la liste
  - savoir si la liste est **vide**
  - **ajouter un élément** avant ou après un autre élément
  - **supprimer** un élément de la liste

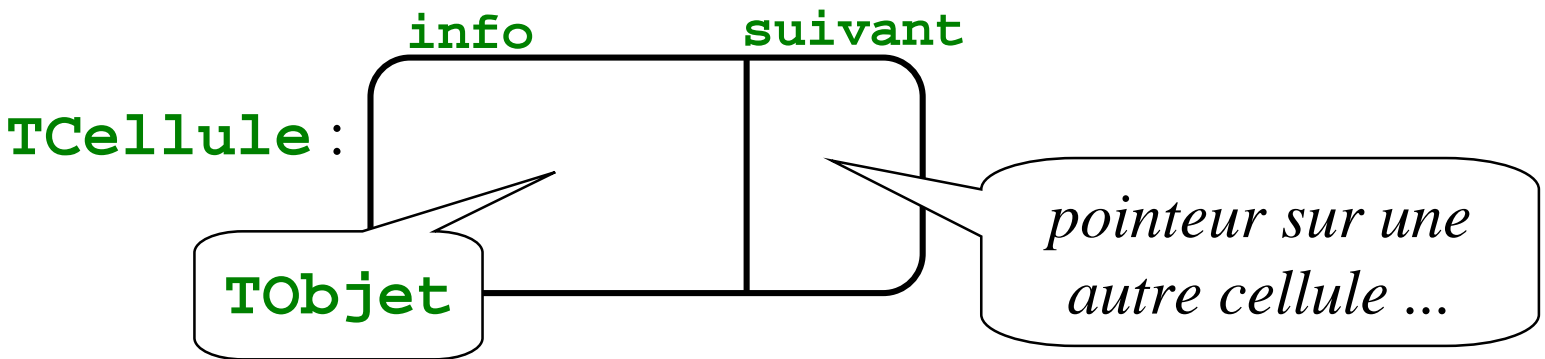


# Comment implanter une liste ?

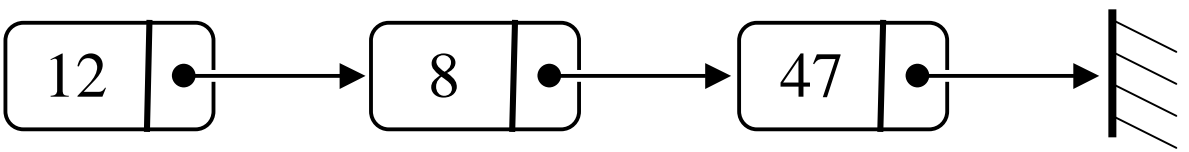
- Deux types d'objets
  - *cellule* : maillon de la liste, stockant un élément et un pointeur sur la cellule suivante
  - *liste* : une suite de cellules (finie, éventuellement vide)
- Type des éléments de la liste
  - des objets de **même type** connu
  - des objets quelconques : liste *générique*
    - ◆ créer un type ancêtre de tous les types des objets à stocker
    - ◆ stocker les adresses des objets
- Structure **récursive**
  - lorsqu'on ôte le premier élément d'une **liste** non vide, **le reste est encore une liste**

# Représentation d'une cellule

- Type d'une cellule : **TCellule**
  - un champ **info** pour stocker un élément
  - un champ **suisvant** pour pointer la cellule suivante



- Exemple : 3 cellules reliées (éléments de type entier)



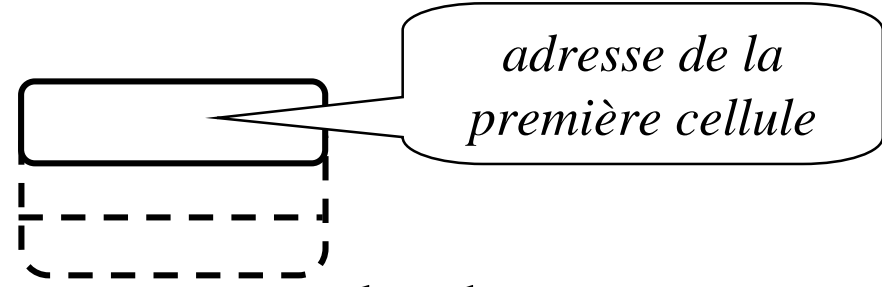
- Représente la liste d'entiers (12, 8, 7)

# Représentation d'une liste

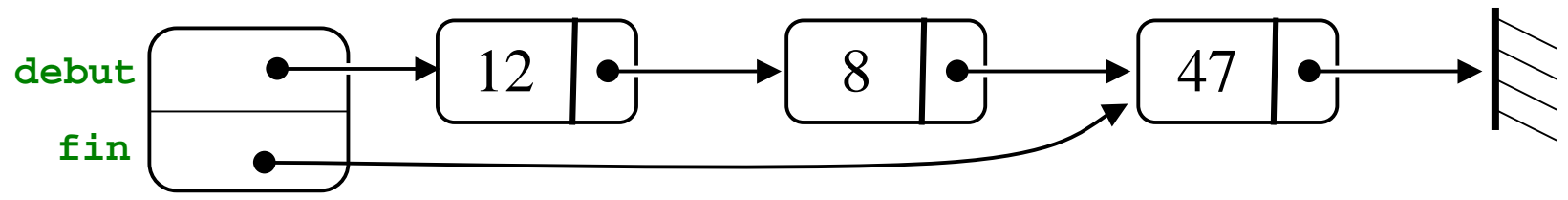
- Nom du type objet : **TListe**
  - un champ **debut** pour pointer la première cellule
  - plus tard, d'autres champs : **nbElt**, **fin** ...

- Représentation

**debut**



- Liste vide : valeur **nil** pour le champ **debut**
- Exemple de liste avec 3 éléments de type entier





# Types TListe et TCellule

INTERFACE

USES objet;

TYPE

TPtCellule = ^TCellule;

TCellule = OBJECT

info : TObjet;

suisvant : TPtCellule;

{}

CONSTRUCTOR init;

{Self est construit, Self.suisvant = nil}

END;

TListe = OBJECT

debut : TPtCellule;

{}

CONSTRUCTOR init;

{Self est construit, Self.debut = nil}

END;

IMPLEMENTATION ...

# Méthodes du type **TListe**

- Primitives d'accès (à spécifier)
  - Première cellule de la liste appelante  
**FUNCTION premier : TPtCellule;**
  - Reste de la liste appelante : c'est la liste privée de sa première cellule. Sans effet sur une liste vide.  
**PROCEDURE reste;**
    - ◆ on peut aussi implanter **reste** comme une fonction ...
  - Fonction booléenne qui indique si la liste est vide  
**FUNCTION estVide : boolean;**
- Méthodes utiles : primitives du type **TCellule**
  - Quelle est l'information contenue dans une cellule ?
  - Une cellule a-t-elle une cellule suivante ?

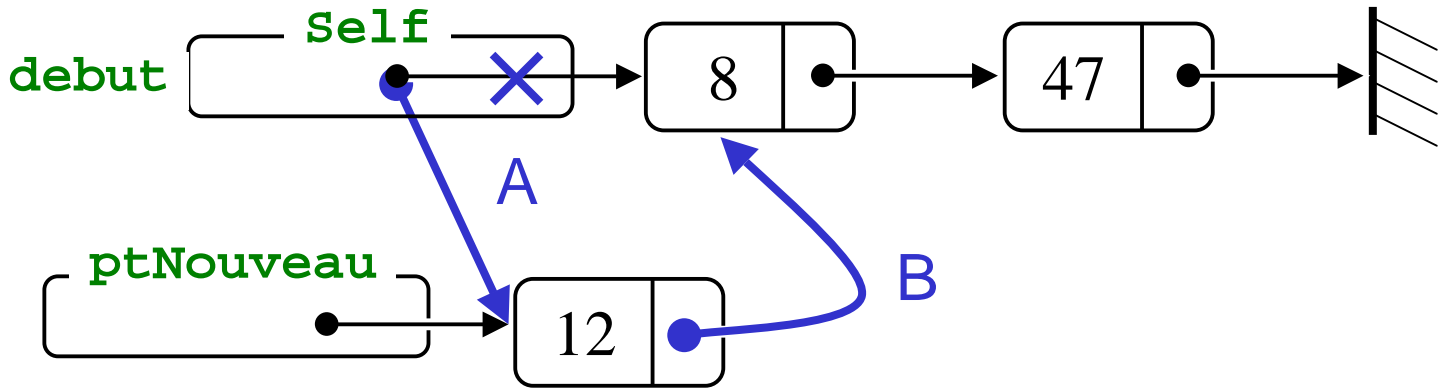
# Insérer un objet dans une liste

- Où insérer ?
  - Au début ou à la fin de la liste
  - Avant ou après un élément donné
- Spécification de l'insertion au début
  - dans l'interface de **TListe**

```
{ self = listeAvant, ptNouveau <> nil }  
PROCEDURE insererTete(ptNouveau : TPtCellule);  
{ self.debut = ptNouveau  
  self.debut^.suivant = listeAvant.debut }
```

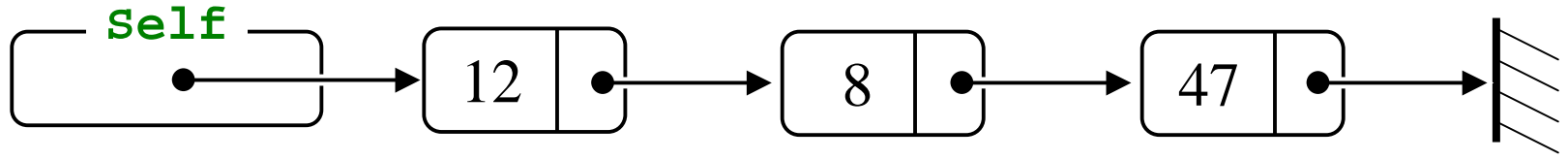
- Implantation
  - Attention à l'ordre des instructions
  - Commencer par concevoir l'algorithme

# Conception de l'insertion en tête



- Opérations à effectuer : flèches **grasses**
- Ordre des opérations et traduction en affectations
- .....
- .....

## • Résultat





# Bien programmer avec les listes

- Pour chaque méthode transformant des listes,
  - **dessinez** les listes et leurs pointeurs,
  - **dessinez** les déplacements de pointeurs,
  - traduisez les déplacements de pointeurs en affectations,
  - **ordonnez** les affectations dans le temps.
  - Enfin, **programmez** les affectations dans l'ordre
- Ce que vous évitez en suivant cette démarche
  - ◆ la perte d'une cellule ou d'une partie de la liste qui cesserait d'être pointée (chaîne brisée)
  - ◆ des pointeurs qui ne pointeraient plus sur rien

Le temps consacré à concevoir la méthode serait sinon perdu plusieurs fois à réparer ce genre d'erreurs

# Exemple d'utilisation

```
PROCEDURE utilise;  
VAR  
    uneListe : TListe;  
    unPtCellule : TPtCellule; { cellule ajoutee }  
BEGIN  
    uneListe.init;           { la liste est vide }  
    unPtCellule := NEW(TPtCellule,init);  
    unPtCellule^.info := 47;  
    uneListe.insererTete(unPtCellule);      { un element }  
    unPtCellule := NEW(TPtCellule,init);  
    unPtCellule^.info := 8;  
    uneListe.insererTete(unPtCellule);      { 2 elements }  
END;
```

- Ici, les objets stockés sont des entiers
- Représenter l'état de la mémoire à chaque étape



# Listes et récursivité

- Structure **récursive** : définie à partir d'elle même
  - Une **liste** est soit vide, soit composée d'une première cellule et d'un reste qui **est aussi une liste**
  - Les méthodes **premier** et **reste** permettent d'écrire tous les algorithmes sur les listes sous forme **récursive**
- Méthode **récursive** : qui s'appelle elle-même
  - Plus facile à écrire
  - Plus long à concevoir : prudence
  - Risque de non-terminaison
  - Une méthode récursive doit toujours contenir un cas d'arrêt : toujours utiliser la méthode **estVide**



# Affichage récursif d'une liste

Implantation de la méthode **TListe.afficher**

```
PROCEDURE TListe.afficher;  
BEGIN  
    IF NOT Self.estVide THEN BEGIN  
        Self.premier^.info.afficher;  
        Self.reste^.afficher;  
    END;  
END;
```

- Cette implantation suppose que
  - ◆ la méthode **TObjet.afficher** existe
  - ◆ la méthode **TListe.reste** est implantée comme une fonction
- Vérifier la concordance des types
- Penser à l'état de la liste appelante après appel

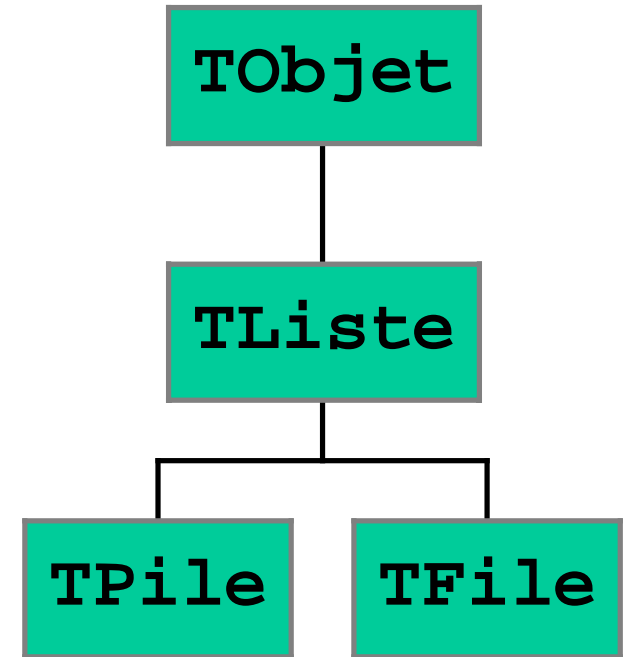


# Autres méthodes sur les listes

- Supprimer un objet de la liste
  - Au début ou à la fin
  - Avant ou après un élément donné
- Recherche d'un élément dans la liste
- Tri de la liste selon l'ordre de ses éléments
  - tri rapide (*Quicksort*) récursif
- Structures de pile, de file ...
  - Restriction de l'ensemble des méthodes disponibles
  - Modélisent certaines réalités (pile des appels, file d'attente)

# Hiérarchie de types de listes

- Restriction des accès
  - Pile : on ajoute et on supprime l'élément au sommet (dernier élément inséré)
  - File : on ajoute en fin de file et on supprime en début de file
- En Pascal
  - moins de méthodes dans **TPile** que dans **TListe**
  - utiliser des méthodes privées ?





# Conclusions

- Liste : structure de données
  - linéaire
    - ◆ Pas de branchements : un seul suivant
    - ◆ Circuits possible : on parle alors de *liste circulaire*
    - ◆ Chaînage arrière en plus : *liste doublement chaînée*
  - dynamique
    - ◆ allocation dynamique de cellules, une par une
  - récursive
    - ◆ soit la liste est vide, soit on traite le premier élément et le reste, qui est encore une liste
- Liste : type objet
  - méthodes de base : **estVide, Premier, Reste**
    - ◆ gérer aussi la destruction des cellules