

Les listes chaînées

Année spéciale

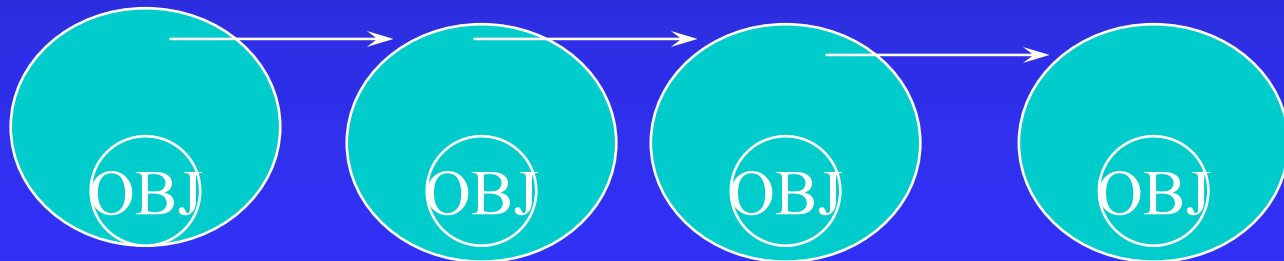
Jean-Pierre Fournier

Les listes chaînées

- constituent une alternative aux tableaux
 - ◆ pour construire une structure de données plus dynamique
 - l'ajout et le retrait d'information y est très rapide
 - la longueur de la liste est inconnue au départ
 - ◆ mais : une liste est constituée d'objets dispersés
 - ne constitue pas vraiment un tout
 - est difficile à sauvegarder et à recharger dans un fichier

Une liste est :

- un ensemble de cellules, reliées entre elles par des pointeurs
 - ◆ chaque cellule possède l'adresse de la cellule suivante de la liste

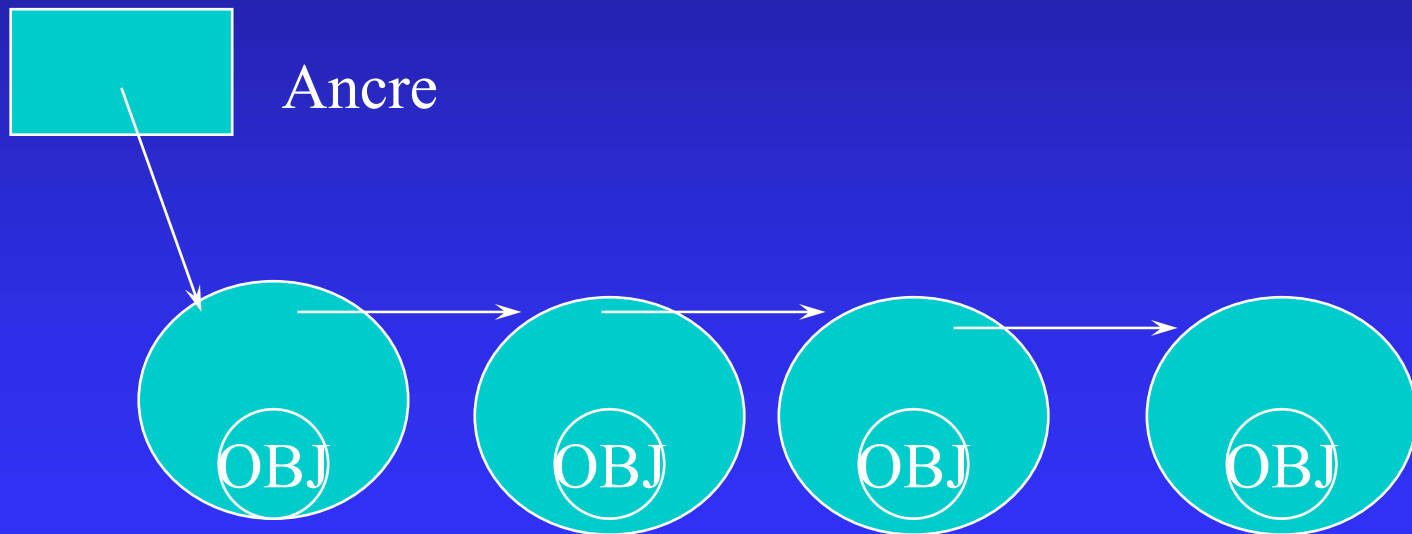


Une liste se construit :

- petit à petit
 - ◆ **p := allouer(cellule)** permet de créer une nouvelle cellule et de garder son adresse dans p.
 - ◆ il suffit d'insérer cette cellule dans le chaînage de la liste pour augmenter la taille de la liste
- elle se détruit aussi petit à petit :
libérer(p->cellule)

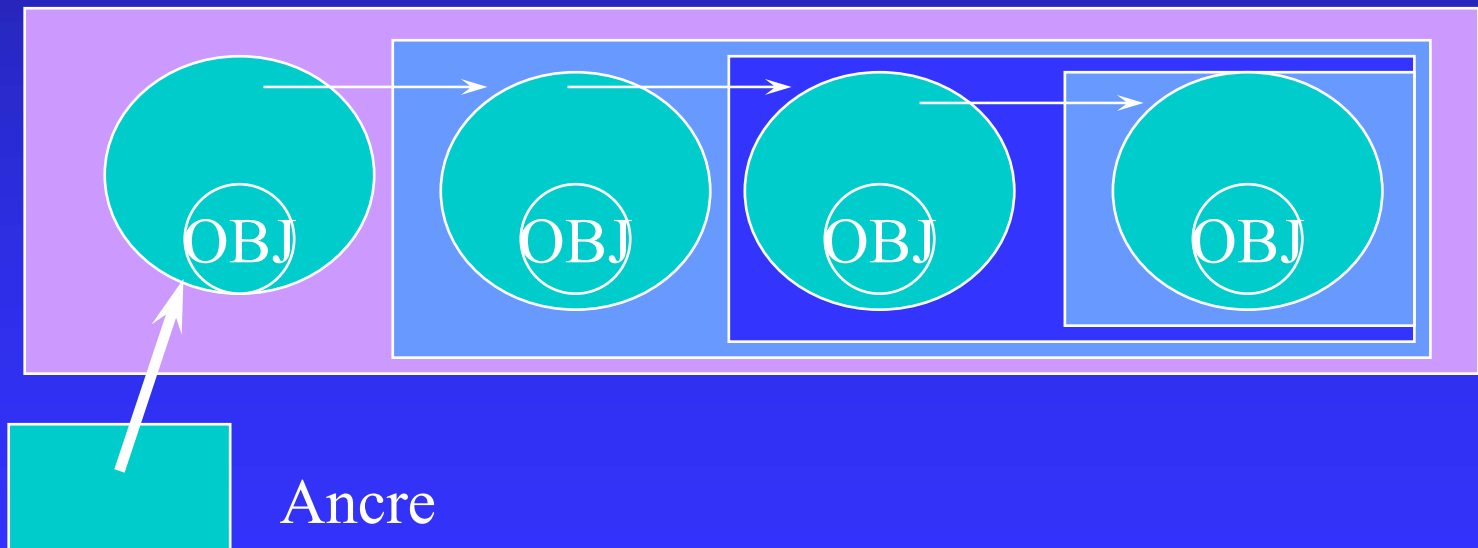
Une liste peut être gérée :

- itérativement : elle est alors vue comme une suite de cellules



Une liste peut être gérée :

- récursivement : elle est alors vue comme une cellule suivie d'une sous-liste



Pour l'utilisateur final

- Une liste est essentiellement un conteneur souple, où il peut déposer des informations dynamiques
 - ◆ ajout rapide
 - ◆ retrait rapide
 - ◆ recherche lente
 - ◆ chargement/sauvegarde lents

Les fonctionnalités

Procédure ajout(uneListe, uneInfo)

Paramètres

| uneListe Liste(E/S)
| uneInfo Info (E)

Procédure retrait(uneListe, uneClef)

Paramètres

| uneListe Liste(E/S)
| uneClef Clef (E)

Les fonctionnalités

Fonction recherche(*uneListe*, *uneClef*)
retourne Info

Paramètres

| *uneListe* Liste (E)
| *uneClef* Clef (E)

Les implémentations

- doivent permettre les fonctionnalités
- peuvent être diverses
- ne sont pas connues de l'utilisateur final
- pourront évoluer, puisqu'aucun algorithme de l'utilisateur de la classe ne s'appuiera sur leurs détails

Exemple 1



Classe Liste

Attributs privés

p pointeur sur Info

suivante pointeur sur Liste

Début

```
q := allouer(Liste)
```

```
i := allouer(Info)
```

```
i->Info := uneInfo
```

```
q -> Liste.suivante :=  
    adresse(uneListe)
```

```
q -> Liste.p := i
```

```
uneListe := q->Liste
```

Fin

Corps de
ajout(uneListe,
uneInfo)



Exemple 2 (simple)

Classe Liste

Attributs privés

val Info

suivante pointeur sur Liste

Début

q := allouer(Liste)


q -> Liste.val := uneInfo

q -> Liste.suivante :=
adresse(uneListe)

uneListe := q->Liste

Fin

Corps de
ajout(uneListe,
uneInfo)



Exemple 3

Classe Liste

Attributs privés

val Info

suivante Liste

Attention à la mise en œuvre !

Début

```
q := allouer(Liste)
```


```
q -> Liste.val := uneInfo
```

```
q -> Liste.suivante :=  
    uneListe
```

```
uneListe := q->Liste
```

Fin

Corps de
ajout(uneListe,
uneInfo)



Autres implémentations

- l'objet Liste peut contenir un pointeur vers une cellule, qui contient elle-même une information et un pointeur vers la cellule suivante (exploitation itérative)
- l'objet Liste peut contenir un objet cellule, contenant une information et un pointeur vers la cellule suivante (exploitation itérative)

Et encore...

- L'objet Liste peut contenir un objet Cellule (ou un pointeur vers un tel objet), qui contient lui-même une information (ou un pointeur vers une information) et un objet Liste (ou un pointeur vers un objet Liste)

La solution la plus simple :

- Celle de l'objet Liste qui contient une information et un objet Liste
 - ◆ mais la plupart des langages de programmation ne permettent pas cette implémentation (construction récursive infinie)
- Un compromis acceptable :
 - ◆ l'objet Liste contient une information et un pointeur vers un objet Liste