

# Structure de Données

## Pointeur et Liste

Marie Pelleau

[marie.pelleau@univ-cotedazur.fr](mailto:marie.pelleau@univ-cotedazur.fr)

Semestre 3

## Rappels

### Variable

- Une variable sert à mémoriser de l'information
- Ce qui est mis dans une variable est en fait mis dans une partie de la mémoire

### Structures de données

- Permettent de gérer et d'organiser des données
- Sont définies à partir d'un ensemble d'opérations qu'elles peuvent effectuer sur les données
- Ne regroupent pas nécessairement des objets du même type

## Plan

- 1 Pointeur
- 2 Liste
  - Liste simplement chaînée
  - Liste doublement chaînée
- 3 Implémentation
- 4 Exemple d'utilisation de listes

## Besoin d'indirections

### Présentation habituelle de certains algorithmes

- On a un tableau d'entiers
- On veut trier ce tableau
- Un élément du tableau est directement un type de base (un entier, un flottant, un booléen...)
- Parfois on ne voudrait pas avoir accès à la valeur en soit, mais plutôt à un objet lié à l'indice et associé à cette valeur
- On veut simplement parcourir les éléments d'un ensemble, pas uniquement les valeurs de ces éléments : on associe l'élément à une valeur

## Besoin d'indirections

Recherche dichotomique : ce qui nous intéresse

- n'est pas la valeur
- n'est pas uniquement l'appartenance de la valeur
- c'est la position de la valeur dans le tableau, donc son indice

On pourrait travailler uniquement avec des indices et des tableaux

⇒ Un indice représentant un objet particulier

### Inconvénient

C'est compliqué

- quand on veut supprimer un objet (que devient son indice ?)
- quand on veut insérer un objet (que devient son indice ?)
- quand on veut ajouter un objet (les tableaux doivent être agrandis)

## Pointeur

- Un pointeur est un type de données dont la valeur fait référence (référence) directement (pointe vers) à une autre valeur
- Un pointeur référence une valeur située quelque part d'autre en mémoire habituellement en utilisant son adresse
- Un pointeur est une variable qui contient une adresse mémoire
- Un pointeur permet de réaliser des indirections : désigner des objets, sans être ces objets

## Besoin d'indirections

Il est plus pratique de travailler directement avec des objets et d'associer des valeurs à ces objets

```
Class MonObjet { ... }
```

```
MonObjet moj1 = new MonObjet (...);
MonObjet moj2 = new MonObjet (...);
// on définit 2 objets
```

```
MonObjet obj; // on définit un autre objet
obj = moj1;
obj.setValue(8); // change une donnée de moj1
```

```
obj = moj2;
obj.setValue(12); // change une donnée de moj2
```

obj change indirectement moj1 et moj2, c'est une indirection

## Pointeur

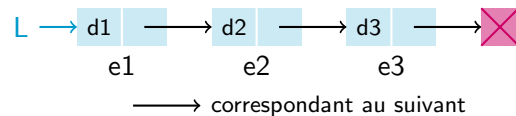
- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur
- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur
- Un pointeur qui ne pointe vers aucune valeur aura la valeur **nil**

## Liste

- Une **liste chaînée** désigne une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments
- L'accès aux éléments d'une liste se fait de manière séquentielle
  - chaque élément permet l'accès au suivant (contrairement au cas du tableau dans lequel l'accès se fait de manière absolue, par adressage direct de chaque cellule dudit tableau)
- **Un élément contient un accès vers une donnée**

## Liste simplement chaînée

## Représentation



- $\text{premier}(L) = e1$
- $\text{donnée}(e1) = d1$ ,  $\text{suivant}(e1) = e2$
- $\text{donnée}(e2) = d2$ ,  $\text{suivant}(e2) = e3$
- $\text{donnée}(e3) = d3$ ,  $\text{suivant}(e3) = \text{nil}$

## Liste

Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, des pointeurs vers les éléments qui lui sont logiquement adjacents dans la liste

## Opérations/syntaxe

- **premier**(L) : désigne le premier élément de la liste
- **nil** : désigne l'absence d'élément

## Liste simplement chaînée

- **donnée**(elt) : désigne la donnée associée à l'élément elt
- **suivant**(elt) : désigne l'élément suivant elt

## Liste

## Trois opérations principales

- Parcours de la liste
- Ajout d'un élément
- Suppression d'un élément

À partir de là d'autres opérations vont être obtenues : recherche d'une donnée, remplacement, concaténation de liste, fusion de listes, ...

## Liste vs Tableau

### Principal avantage des listes sur les tableaux

- L'ordre des éléments de la liste peut être différent de leur ordre en mémoire
- Les listes chaînées vont permettre l'ajout ou la suppression d'un élément en n'importe quel endroit de la liste en temps constant

### Inconvénient

- Certaines opérations peuvent devenir coûteuses comme la recherche d'un élément contenant une certaine donnée
- Pas de recherche dichotomique dans une liste : on ne peut pas atteindre le  $i^{\text{ème}}$  élément sans parcourir

## Lvalue et Rvalue

- Pour se simplifier la vie, on accepte de faire `suivant(elt) <- valeur`
- On remarque qu'il n'y a pas d'ambiguïté
- Cela s'appelle une Lvalue ou Left-value (on accepte de mettre à gauche de l'affectation)
- Le cas normal est la Rvalue (right-value)

## Invention des listes chaînées

- La représentation de listes chaînées à l'aide du diagramme avec une flèche vers le suivant a été proposé par Newell and Shaw dans l'article "Programming the Logic Theory Machine" Proc. WJCC, February 1957
- Newell et Simon ont obtenu l'ACM Turing Award en 1975 pour avoir "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing"

## Liste

### Initialisation d'une liste

La liste est vide, son premier élément est `nil`

```
initListe(L) {
  premier(L) <- nil
}
```

### Initialisation de L



## Liste

## Compter le nombre d'éléments

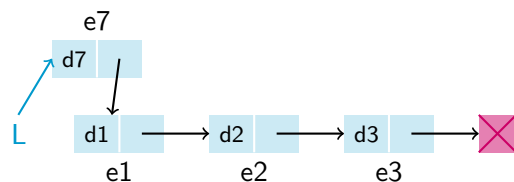
```
entier nombreElements(L) {
  cpt <- 0
  elt <- premier(L)
  tant que (elt ≠ nil) {
    cpt <- cpt + 1
    elt <- suivant(elt)
  }
  retourner cpt
}
```

## Liste

## Ajout d'un élément

```
ajouteAuDébut(elt, L) {
  // elt n'est pas dans L
  suivant(elt) <- premier(L)
  premier(L) <- elt
}
```

## Ajout de e7



## Liste

## Ajout d'un élément

- On ajoute un élément `elt` au début de la liste
- On suppose qu'il n'est pas déjà dans la liste (sinon que se passe-t-il ?)

## Principes

- Le premier de la liste deviendra `elt`
- Mais où est l'ancien premier ? Il devient le suivant de `elt`

## Attention

L'ordre de mise à jour est important ! On ne doit pas perdre le premier

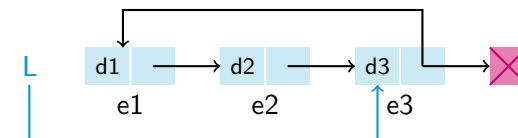
- Le suivant de `elt` est mis à jour
- Puis le premier de la liste

## Liste

## Ajout d'un élément

```
ajouteAuDébut(elt, L) {
  // elt n'est pas dans L
  suivant(elt) <- premier(L)
  premier(L) <- elt
}
```

## Ajout de e3



## Liste

## Insertion d'un élément

- On insère un élément `elt` après un autre `p`
- On suppose que `elt` n'est pas déjà dans la liste et que `p` y est (sinon que se passe-t-il ?)

## Principes

- Le suivant de `elt` devient le suivant de `p`
- Le suivant de `p` devient `elt`

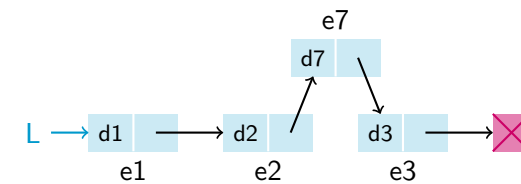
## Attention

L'ordre de mise à jour est important !

## Liste

## Insertion d'un élément

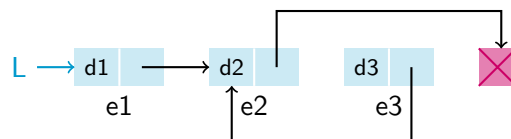
```
insèreArrière(elt, p, L) {
  // elt n'est pas dans L, p est dans L
  suivant(elt) ← suivant(p)
  suivant(p) ← elt
}
```

Insertion de `e7` après `e2`

## Liste

## Insertion d'un élément

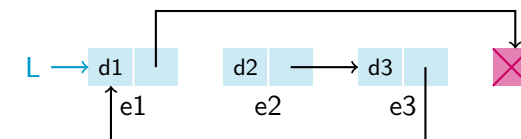
```
insèreArrière(elt, p, L) {
  // elt n'est pas dans L, p est dans L
  suivant(elt) ← suivant(p)
  suivant(p) ← elt
}
```

Insertion de `e2` après `e3`

## Liste

## Insertion d'un élément

```
insèreArrière(elt, p, L) {
  // elt n'est pas dans L, p est dans L
  suivant(elt) ← suivant(p)
  suivant(p) ← elt
}
```

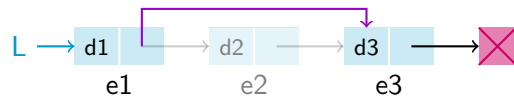
Insertion de `e1` après `e3`

## Liste

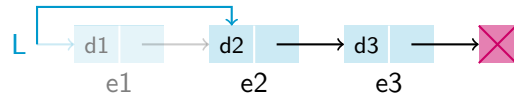
## Suppression d'un élément

- On supprime un élément de la liste
- On a besoin du précédent !
- Le premier peut changer !

## Suppression de e2



## Suppression de e1



## Liste

## Suppression d'un élément

- On supprime un élément de la liste
- On a besoin du précédent !
- Le premier peut changer !

## Principe

Le suivant du précédent devient le suivant de elt

## Gestion de tous les cas

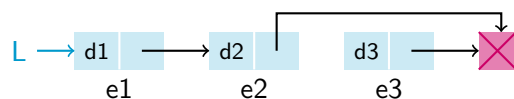
- elt est le premier
- elt n'est pas le premier
- p est bien le précédent de elt

## Liste

## Suppression d'un élément

```
supprime(elt, p, L) {
  //elt est dans L, p son précédent
  si (premier(L) = elt) {
    premier(L) <- suivant(elt)
  } sinon {
    si (suivant(p) = elt) {
      suivant(p) <- suivant(elt)
    }
  }
}
```

## Suppression de e3 – supprime(e3, e2, L)



## Plan

- 1 Pointeur
- 2 Liste
  - Liste simplement chaînée
  - Liste doublement chaînée
- 3 Implémentation
- 4 Exemple d'utilisation de listes

## Liste

## Liste simplement chaînée

- **donnée**(elt) désigne la donnée associée à l'élément elt
- **suivant**(elt) désigne l'élément suivant elt

## Liste doublement chaînée

- **donnée**(elt) désigne la donnée associée à l'élément elt
- **suivant**(elt) désigne l'élément suivant elt
- **précédent**(elt) désigne l'élément précédant elt

## Liste

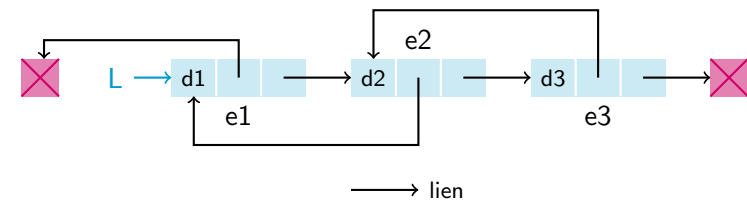
## Trois opérations principales

- Parcours de la liste
- Ajout d'un élément
- Suppression d'un élément

À partir de là d'autres opérations vont être obtenues : recherche d'une donnée, remplacement, concaténation de liste, fusion de listes, ...

## Liste doublement chaînée

## Représentation



- **premier**(L) = e1
- **donnée**(e1) = d1, **suivant**(e1) = e2, **précédent**(e1) = nil
- **donnée**(e2) = d2, **suivant**(e2) = e3, **précédent**(e2) = e1
- **donnée**(e3) = d3, **suivant**(e3) = nil, **précédent**(e3) = e2

## Liste

## Ajout d'un élément

- On ajoute un élément elt au début de la liste
- On suppose qu'il n'est pas déjà dans la liste (sinon que se passe-t-il ?)

## Principes

- Le suivant de elt est le premier
- Le premier de la liste deviendra elt

## Attention

L'ordre de mise à jour est important !

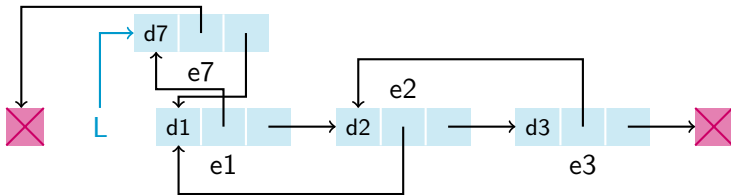


## Liste

## Ajout d'un élément

```
ajouteAuDébut(elt, LD) {
  // elt n'est pas dans LD
  suivant(elt) <- premier(LD)
  précédent(elt) <- nil
  précédent(premier(LD)) <- elt
  premier(LD) <- elt
}
```

## Ajout de e7



## Liste

## Insertion d'un élément

- On insère un élément `elt` après un autre `p`
- On suppose que `elt` n'est pas déjà dans la liste et que `p` y est (sinon que se passe-t-il ?)

## Principes

- Le suivant de `elt` devient le suivant de `p`
- Le précédent de `elt` est `p`
- Le suivant de `p` devient `elt`
- Le précédent de `suivant(p)` devient `elt`

Sont modifiés : `suivant(elt)`, `précédent(elt)`, `suivant(p)`, `précédent(suivant(p))`

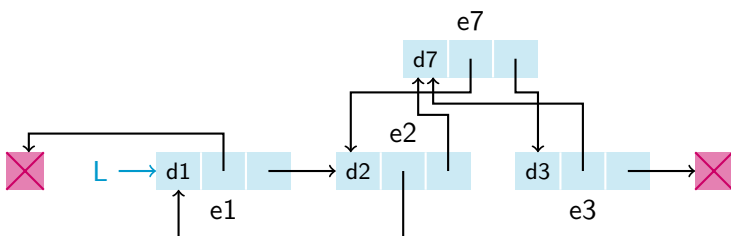
## Liste

## Insertion d'un élément

```
insèreAprès(elt, p, LD) {
  // elt n'est pas dans LD, p est dans LD
  suivant(elt) <- suivant(p)
  précédent(elt) <- p
  précédent(suivant(p)) <- elt
  suivant(p) <- elt
}
```

Mauvais code !  
Pourquoi ?

## Insertion de e7 après e2

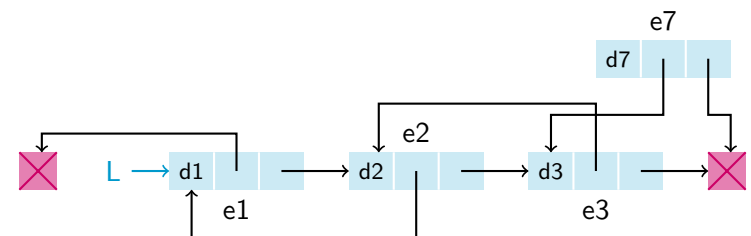


## Liste

## Insertion d'un élément

```
insèreAprès(elt, p, LD) {
  // elt n'est pas dans LD, p est dans LD
  suivant(elt) <- suivant(p)
  précédent(elt) <- p
  précédent(suivant(p)) <- elt
  suivant(p) <- elt
}
```

## Insertion de e7 après e3 ?



## Liste

## Insertion d'un élément

```

insèreAprès(elt, p, LD) {
  // elt n'est pas dans LD, p est dans LD
  suivant(elt) <- suivant(p)
  précédent(elt) <- p
  si (suivant(p) ≠ nil) {
    précédent(suivant(p)) <- elt
  }
  suivant(p) <- elt
}

```

## Liste

## Suppression d'un élément

```

supprime(elt, LD) {
  // elt dans LD
  suiv <- suivant(elt)
  prec <- précédent(elt)
  si (prec = nil) {
    premier(LD) <- suiv
  } sinon {
    suivant(prec) <- suiv
  }
  si (suiv ≠ nil) {
    précédent(suiv) <- prec
  }
}

```

## Liste

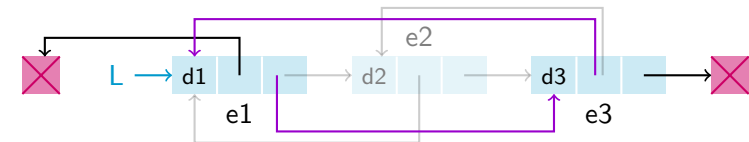
## Suppression d'un élément

- On supprime un élément de la liste
- On n'a pas besoin du précédent !
- Le premier peut changer !

## Principes

- Le suivant du précédent devient le suivant de elt
- Le précédent du suivant devient le précédent de elt

## Suppression de e2



## Plan

- 1 Pointeur
- 2 Liste
  - Liste simplement chaînée
  - Liste doublement chaînée
- 3 Implémentation
- 4 Exemple d'utilisation de listes

## Implémentation

- Par un tableau
- À l'aide de pointeur

## Pointeur

- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur
- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur
- Un pointeur qui ne pointe vers aucune valeur aura la valeur **nil**
- Un pointeur c'est un indice dans le grand tableau de la mémoire

## Tableaux simulant une liste

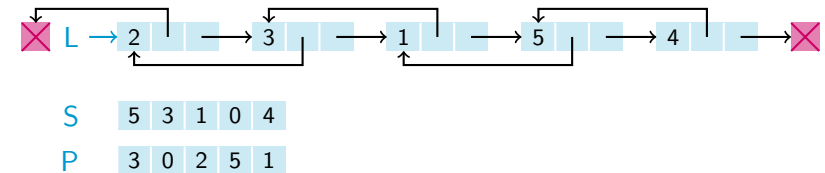
### Tableau S des suivants

- $S[i]$  indice de l'élément suivant l'élément d'indice  $i$

### Tableau P des précédents

- $P[i]$  indice de l'élément précédant l'élément d'indice  $i$

### Exemple



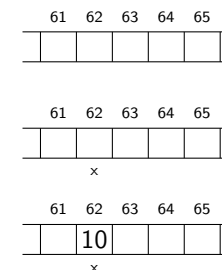
## Pointeur Implémentation

```
int x // Réserve un emplacement pour un entier en mémoire
x = 10 // Écrit la valeur 10 dans l'emplacement réservé
```

- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée
- Physiquement cette valeur se situe en mémoire

- `int x`

- `x = 10`



## Pointeur Implémentation

```
int x // Réserve un emplacement pour un entier en mémoire
x = 10 // Écrit la valeur 10 dans l'emplacement réservé
```

- `int x`

61	62	63	64	65

x
- `x = 10`

61	62	63	64	65
	10			

x
- `&x` : adresse de x en C (ici 62)
- En C : `int* px; // pointeur sur un entier`

61	62	63	64	65
	10			

x

95	96	97	98
	?		

px
- `px = &x; // adresse de x`

61	62	63	64	65
	10			

x

95	96	97	98
	62		

px

## Pointeur Implémentation

- Si px contient l'adresse de x
- Alors \*px contient la valeur qui se trouve à l'adresse de x, donc la valeur de x
- Si je change la valeur qui se trouve à l'adresse de x, alors je change la valeur de x
- `px = &x` Si je modifie \*px alors je modifie x
- `px = &y` Si je modifie \*px alors je modifie y
- `px = &bidule` Si je modifie \*px alors je modifie bidule
- px désigne l'objet pointé
- \*px modifie l'objet pointé

## Pointeur et référence

- Une **référence** est une valeur qui permet l'accès en lecture et/ou écriture à une donnée située soit en mémoire principale soit ailleurs
- Une référence n'est pas la donnée elle-même mais seulement une information de localisation
- Ressemble à quelque chose de connu, non ?
- Le typage des références permet de manipuler les données référencées de manière abstraite tout en respectant leurs propres contraintes de type
- Le type de référence le plus simple est le pointeur. Il s'agit simplement d'une adresse mémoire
- Mais pointeur typé = référence typé
- Mais on peut changer le type d'un pointeur, pas d'une référence (cast/transtypage autorisé)

## Pointeur et référence

### En Java

Uniquement des références typées

- `MyObject a, b, obj;`
- `obj = a;` si on modifie obj alors on modifie a
- `obj = b;` si on modifie obj alors on modifie b
- `myFonction(obj)` obj est passé en entrée/sortie

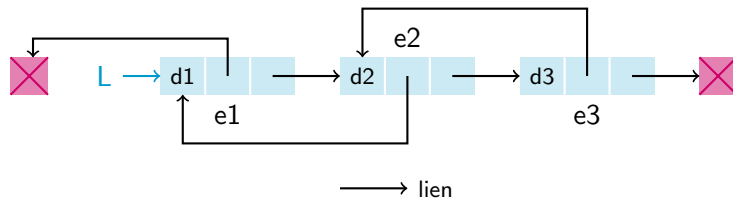
### En C/C++

Pointeurs typés mais type changeable

- `MyObject a, b;`
- `MyObject* obj;`
- `obj = &a;` si on modifie obj alors on modifie a
- `obj = &b;` si on modifie obj alors on modifie b
- `myFonction(MyObject* obj)` obj est passé en entrée/sortie
- `myFonction(MyObject obj)` obj est passé en entrée

## Liste doublement chaînée

## Représentation



- ListeElement
  - suivant : pointeur vers ListeElement
  - précédent : pointeur vers ListeElement
  - donnée : pointeur vers l'objet
- Liste
  - premier : pointeur vers ListeElement

## Liste doublement chaînée

```
class List {
    ListElt _premier
    List() {
        _premier = nil
    }
}
```

## Liste doublement chaînée

```
class ListElt {
    ListElt _suiv
    ListElt _prec
    MaClasseDonnee _data
    ListElt() {
        _suiv = nil
        _prec = nil
        _data = null
    }
}
```

## Plan

- 1 Pointeur
- 2 Liste
  - Liste simplement chaînée
  - Liste doublement chaînée
- 3 Implémentation
- 4 Exemple d'utilisation de listes

## Pile

### Liste simplement chaînée : opérations

- `initListe` (L)
- `nombreElements`(L)
- `premier`(L)
- `ajouteAuDébut`(elt, L)
- `insèreAprès`(elt, p, L)
- `supprime`(elt, p, L)

### Implémentation par une liste

Une liste L simplement chaînée

- `Sommet`(P) : renvoyer `premier`(L)
- `Empiler`(P, elt) : `ajouteAuDébut`(elt, L)
- `Déempiler`(P) : `supprime`(`premier`(L), `nil`, L)
- `estVide`(P) : renvoyer `nombreElements`(L)= 0

## Pile

### Implémentation par une liste

Nécessite par rapport à un tableau

- Plus de place
- Plus d'opérations
- Plus d'allocations

## File

### Liste simplement chaînée : opérations

- `initListe` (L)
- `nombreElements`(L)
- `premier`(L)
- `ajouteEnFin`(elt, L)
- `insèreAprès`(elt, p, L)
- `supprime`(elt, p, L)

### Implémentation par une liste

Une liste L simplement chaînée

- `Sommet`(F) : renvoyer `premier`(L)
- `Enfiler`(F, elt) : `ajouteEnFin`(elt, L)
- `Défiler`(F) : `supprime`(`premier`(L), `nil`, L)
- `estVide`(F) : renvoyer `nombreElements`(L)= 0

## File

### Implémentation par une liste

Beaucoup plus simple qu'un tableau cette fois

- Gestion mémoire plus complexe