

# ITI 1521. Introduction à l'informatique II

Liste : concept

Marcel Turcotte



uOttawa

École de science informatique et de génie électrique (SIGE)  
Université d'Ottawa

Version du 14 mars 2016

# Aperçu

## Liste : concept

Suite à notre exploration des piles et des files, nous examinons un autre type abstrait de données (TAD) linéaire, la liste. Nous découvrons la généralité de ce TAD. Nous mentionnons l'implémentation à l'aide d'un tableau, mais nous portons notre attention sur trois implémentations à l'aide d'éléments chaînés : la liste simplement chaînée, la liste doublement chaînée, et la liste circulaire, doublement chaînée, débutant par un noeud factice.

## Objectif général :

- Cette semaine, vous serez en mesure de concevoir une implémentation de grade industrielle du type abstrait de données liste.

# Objectifs d'apprentissage

- ❖ **Expliquer** le rôle des variables références dans l'implémentation d'une liste chaînée.
- ❖ **Concevoir** une méthode pour traverser une liste simplement chaînée.

## Lectures :

- ❖ Pages 88-97 de E. Koffman et P. Wolfgang.

# Plan

1. Préambule
2. Définitions
3. Implémentations
4. Prologue

# Définition

Une liste (**List**) est un type abstrait de données (TAD) permettant de sauvegarder des objets, tel que chaque élément a un prédécesseur et un successeur (donc linéaire), et **n'ayant aucune restriction au niveau de l'accès aux données**; on peut inspecter, faire une insertion ou une déletion n'importe où dans la liste.

# Opérations

Les **opérations** de base sont :

**int size()** : retourne le nombre d'éléments sauvegardés ; la liste vide a une taille 0 ;

**int get(int index)** : l'accès aux éléments se fait par position (ou par contenu).

- ❖ Quel sera l'index du premier élément de la liste, 0 ou 1 ?

- ❖ Comme pour les tableaux, le premier élément se trouve à la position 0 ;

**void add(int index, E elem)** : ajout d'un élément à une position quelconque ;

**void remove(int index)** : retrait d'un élément par position (ou contenu).

# Remarques

- ❖ Les **listes** sont donc plus **générales** que les piles et les files.
- ❖ Ces dernières peuvent être implémentées à l'aide d'une liste.

# List

```
public interface List<E> {  
    void add(int index, E elem);  
    boolean add(E elem);  
    E remove(int index);  
    boolean remove(E o);  
    E get(int index);  
    E set(int index, E element);  
    int indexOf(E o);  
    int lastIndexOf(E o);  
    boolean contains(E o);  
    int size();  
    boolean isEmpty();  
}
```

- ❖ L'interface ci-haut déclare un sous-ensemble des méthodes de l'interface **java.util.List**.



# Implémentations

## ❖ ArrayList

## ❖ LinkedList

- ❖ Listes **simplement** chaînées
- ❖ Listes **doublement** chaînées
- ❖ Noeud **factice** («*dummy node*»)
- ❖ Traitement **itératif** (**I**terator)
- ❖ Traitement **récurusif**

De nouveaux concepts seront introduits au besoin afin d'améliorer l'efficacité des implémentations.

**Efficacité** par rapport au **temps d'exécution** et/ou **usage de la mémoire** ; nous nous intéresserons surtout à la vitesse d'exécution.

## Discussion : implémenter **List** à l'aide d'un tableau

- ❖ **Résumez** les grandes lignes de l'implémentation d'une liste à l'aide d'un **tableau** :
  - ❖ Une **variable d'instance** désigne un tableau ;
  - ❖ Une **variable d'instance** pour compter le nombre d'éléments ;
  - ❖ On **crée le tableau** dans le constructeur de la classe ;
  - ❖ On utilise la technique du **tableau dynamique**.
- ❖ Étant donné une liste contenant les éléments **a**, **b**, **c** et **d**, donnez le contenu du tableau après l'appel **add(2, z)**.
- ❖ Pour la liste résultante, donnez le résultat de l'appel **remove(0)**.

# Liste simplement chaînée

- ❖ L'implémentation la plus simple est la liste simplement chaînée (**SinglyLinkedList**).
- ❖ Nous utiliserons une classe imbriquée «static» afin de représenter les noeuds de la liste. Chaque noeud contient une valeur et est connecté à son suivant.

```
private static class Node<T> {  
    private T value;  
    private Node<T> next;  
    private Node(T value, Node<T> next) {  
        this.value = value;  
        this.next = next;  
    }  
}
```

# Liste **simplement** chaînée

- ❖ La classe **SinglyLinkedList** a une variable d'instance qui désigne le premier élément de la liste, que nous nommerons **head**.
- ❖ La classe imbriquée est parfois nommée **Elem** ou **Entry**.

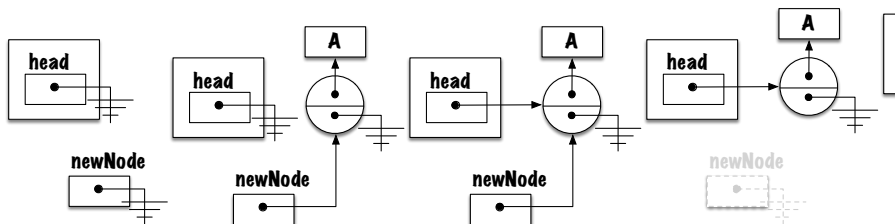
# addFirst(E elem)

L'**insertion** d'un élément en tête de liste nécessite :

1. la création d'un **nouveau noeud**, ainsi que
2. l'**ajout** de l'élément à la liste.

```
public void addFirst(E elem) {  
  
    Node<E> newNode;  
    newNode = new Node<E>(elem, null);  
  
    if (head == null) {  
        head = newNode;  
    } else {  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

# Cas général : addFirst(E elem)



# Discussion

- ❖ Est-ce que cette distinction entre le cas de la **liste vide** et le cas de la **liste ayant des éléments** est vraiment **nécessaire** ?
- ❖ **Que pensez-vous** de cette implémentation ?

```
public void addFirst(E elem) {  
    head = new Node<E>(elem, head);  
}
```

# Discussion (suite)

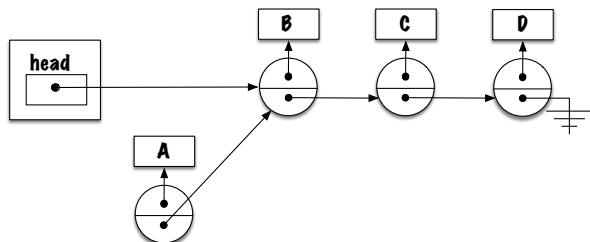
- ❖ **Est-ce que** ça fonctionne pour le **cas spécial** et le **cas général** ?
  - ❖ **Oui**, ça fonctionne.
  - ❖ **Pourquoi ?**
    - ❖ Parce que Java évalue d'abord le côté droit de l'expression.



# addFirst(E elem)

Évaluation du côté droit.

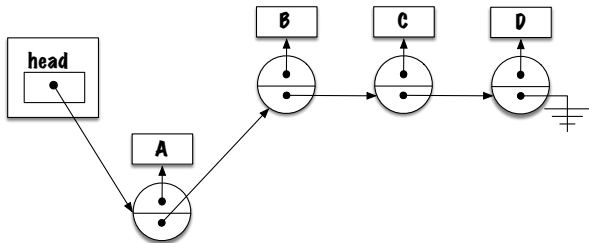
```
head = new Node<E>(elem, head);
```



# addFirst(E elem)

Le résultat (une référence vers l'élément nouvellement créé) est affecté à la variable **head**.

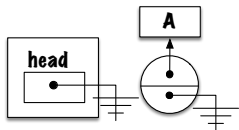
```
head = new Node(elem, head);
```



# addFirst(E elem)

De même, le résultat de l'évaluation du côté droit, ici **head** est **null**, est affecté à la variable d'instance **next** du noeud.

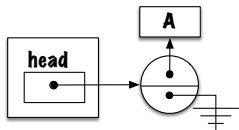
```
head = new Node<E>(elem , head );
```



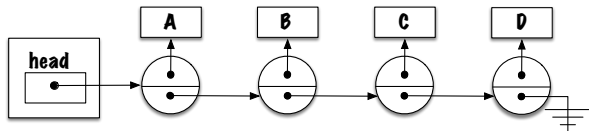
# addFirst(E elem)

Le résultat (une référence vers l'élément nouvellement créé) est affecté à la variable **head**.

```
head = new Node<E>( elem , head );
```

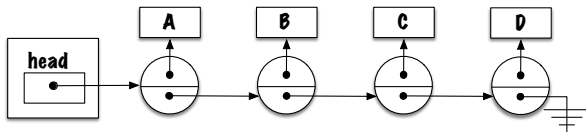


# Discussion $\text{add}(E \text{ elem})$



- ❖ La méthode  **$\text{add}(E \text{ elem})$**  ajoute l'élément à la fin de la liste.
  - ❖ Sans ajouter une référence vers le dernier noeud, **comment** ajoute-t-on l'élément à la fin de la liste ?
    - ❖ La solution doit être **générale** !

# add(E elem)



# add(E elem)

- Quel sera la **condition d'arrêt** de la boucle ?

```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
  
    p = head;  
    while ( ) {  
        p = p.next;  
    }  
  
    p.next = newNode;  
}
```

# add(E elem)

## ❖ Qu'en pensez-vous ?

```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
  
    p = head;  
    while (p != null) {  
        p = p.next;  
    }  
  
    p.next = newNode;  
}
```





## add(E elem)

- Après l'exécution de la boucle, la valeur de **p** est **null**, une exception de type **NullPointerException** sera lancée lors de l'exécution de **p.next = newNode**.

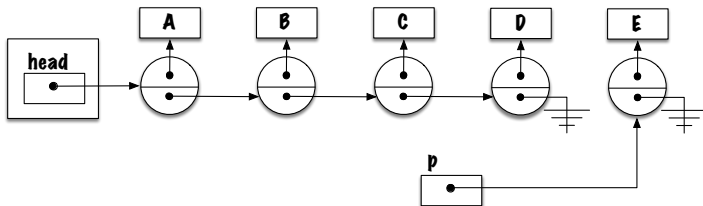
```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
    p = head;  
  
    while (p.next != null) {  
        p = p.next;  
    }  
  
    p.next = newNode;  
}
```

# add(E elem)

- ❖ Nouvelle proposition !
- ❖ **Qu'en pensez-vous ?**

```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
    p = head;  
  
    while (p != null) {  
        p = p.next;  
    }  
  
    p = newNode;  
}
```

# add(E elem)



# add(E elem)

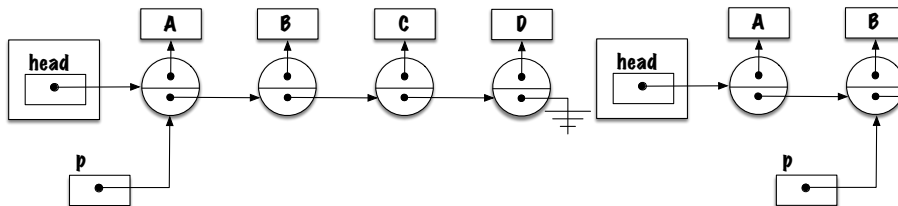
- Quel sera la **condition d'arrêt** de la boucle?

```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
  
    p = head;  
    while ( ) {  
        p = p.next;  
    }  
  
    p.next = newNode;  
}
```

# add(E elem)

```
public void add(E elem) {  
  
    Node<E> newNode, p;  
  
    newNode = new Node<E>(elem, null);  
  
    p = head;  
    while (p.next != null) {  
        p = p.next;  
    }  
  
    p.next = newNode;  
}
```

# add(E elem)



# add(E elem)

- ❖ Que se passe-t-il si la liste est vide?
  - ❖ `p.next != null` cause l'exception **NullPointerException** !
- ❖ Quelle **variable** reçoit le nouvel élément ?

```
public void add(E elem) {  
    Node<E> newNode, p;  
    newNode = new Node<E>(elem, null);  
    p = head;  
    while (p.next != null) {  
        p = p.next;  
    }  
    p.next = newNode;  
}
```



# add(E elem)

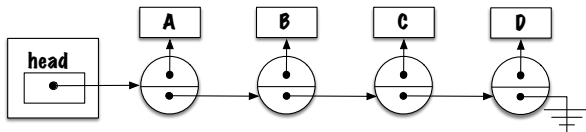
```
public void add(E elem) {
    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
    } else {
        Node<E> p;
        p = head;
        while (p.next != null)
            p = p.next;
        p.next = newNode;
    }
}
```

# Piège !

```
public void add(E elem) {  
    Node<E> newNode;  
    newNode = new Node<E>(elem, null);  
  
    if (head == null) {  
        head = newNode;  
    } else {  
        while (head.next != null)  
            head = head.next;  
        head.next = newNode;  
    }  
}
```

# add(E elem)



# Exercises

Implémentez ces méthodes :

- **E removeFirst()**

- **E removeLast()**

- Quel sera le **critère d'arrêt** de la boucle ?

# remove(E elem)

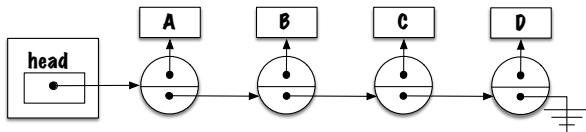
- Accès aux éléments **par contenu** !
- Retourne **true** si **elem** a été retiré et **false** sinon.

# remove(E elem)

Élaborez votre **stratégie** !

1. **Traverser** la liste
2. **Critère d'arrêt** ?
3. **Retrait**

# remove(E elem)



# remove(E elem)

## ❖ Qu'en pensez-vous ?

```
public boolean remove(E elem) {  
    Node<E> p, r;  
    p = head;  
    while (p != null && ! p.value.equals(elem)) {  
        p = p.next;  
    }  
    r = p;  
    // ...  
    return true;  
}
```



```
public boolean remove(E elem) {  
  
    Node<E> p, r;  
    p = head;  
  
    while (p.next != null && ! p.next.value.equals(elem)) {  
        p = p.next;  
    }  
  
    r = p.next;  
    p.next = r.next;  
  
    return true;  
}
```

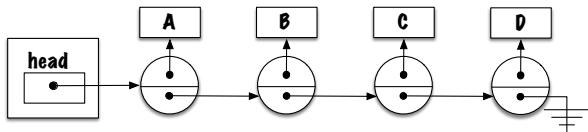
- ❏ Que se passe-t-il si la liste est **vide** ?
- ❏ Que se passe-t-il si l'élément est **absent** de la liste ?

```
public boolean remove(E elem) {
    boolean result = true;
    if (head == null) {
        result = false;
    } else if (head.value.equals(elem) ) {
        head = head.next;
    } else {
        Node<E> p;
        p = head;
        while (p.next != null && ! p.next.value.equals(elem))
            p = p.next;
    }
    if (p.next == null) {
        result = false;
    } else {
        p.next = p.next.next;
    }
}
return result;
}
```

# E get(int pos)

- ❖ Retourne la valeur sauvegardée à la position **pos** de la liste.
  - ❖ Accès **par position** !
- ❖ Le **premier élément** de la liste est à la **position 0**.
- ❖ **Élaborez** votre stratégie.

# E get(int pos)



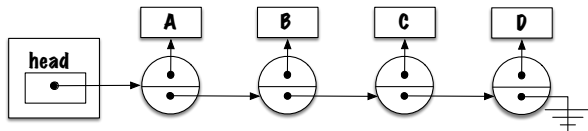
# E get(int pos)

```
public E get(int pos) {  
  
    Node<E> p;  
    p = head;  
  
    for (int i=0; i<pos; i++) {  
        p = p.next;  
    }  
  
    return p.value;  
}
```

# Exercise

- Implémentez la méthode **E remove(int pos)**.

# E remove(int pos)



# E remove(int pos)

```
public E remove(int pos) {
    // pre-conditions : ?
    E saved;
    Node<E> r;
    Node<E> p;
    p = head;
    for (int i=0; i<(pos-1); i++) {
        p = p.next;
    }
    r = p.next;
    p.next = r.next;
    saved = r.value;
    return saved;
}
```

- ❏ Qu'arrive-t-il si **pos == 0** ?



# E remove(int pos)

```
public E remove(int pos) {
    E saved;
    Node<E> r;
    if (pos == 0) {
        r = head;
        head = head.next;
    } else {
        Node<E> p;
        p = head;
        for (int i=0; i<(pos-1); i++) {
            p = p.next;
        }
        r = p.next;
        p.next = r.next;
    }
    saved = r.value;
    return saved;
}
```

# Résumé

- ❖ Une liste (**List**) est un type abstrait de données (TAD) permettant de sauvegarder des objets, tel que chaque élément a un prédécesseur et un successeur (donc linéaire), et **n'ayant aucune restriction au niveau de l'accès aux données** ; on peut inspecter, faire une insertion ou une déletion n'importe où dans la liste.
- ❖ Pour **traverser** une liste, on utilise une variable locale de type **Elem** que l'on initialise avec la valeur de **head**.

# Prochain module

- **Listes** : techniques d'implémentation

# References I



E. B. Koffman and Wolfgang P. A. T.  
*Data Structures : Abstraction and Design Using Java.*  
John Wiley & Sons, 2e edition, 2010.



# Pensez-y !

L'impression de ces notes n'est probablement pas nécessaire !