ITI 1521. Introduction à l'informatique II

Liste : implémentations

by

Marcel Turcotte

Version du 19 mars 2020

Préambule

Préambule

Aperçu

Aperçu

Liste : implémentations

Nous portons toute notre attention sur trois implémentations de l'interface **List** 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.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Expliquer** le rôle des variables références dans l'implémentation d'une liste chaînée.
- **Modifier** l'implémentation d'une liste simplement ou doublement chaînée afin d'y ajouter une nouvelle méthode.
- **Justifier** la fonction du noeud factice dans l'implémentation d'une liste circulaire doublement chaînée.
- **Discuter** les avantages et les désavantages, notamment, au niveau du temps d'exécution et de l'utilisation de la mémoire, pour les trois implémentations d'une liste vue dans ce cours, soit la liste simplement chaînée, la liste doublement chaînée, et la liste circulaire doublement chaînée débutant par un noeud factice.

Lectures :

- Pages 84-89, 103 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

Définitions

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 délétion n'importe où dans la liste.

Implémentations

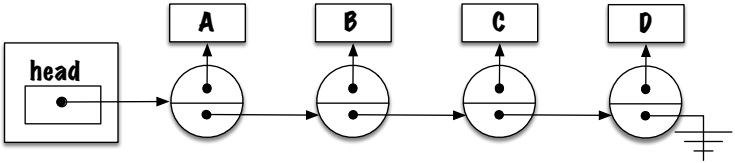
- **ArrayList**
- **LinkedList**

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;
    }
}
```

LinkedList



Définition

- Nous comparons l'efficacité des implémentations à base de tableaux (**ArrayList**) et à base de listes chaînées (**LinkedList**).
  - Toutes deux peuvent contenir un nombre illimité d'objets, donc **ArrayList** utilise un tableau dynamique.
- Nous dirons que le temps d'exécution est **variable** (lent), si le nombre d'opérations varie selon le nombre d'éléments présentement sauvegardés dans la structure de données, et **constant** (rapide) sinon.

Implémentations

Comparer ArrayList et LinkedList

- Pouvez-vous prédire laquelle des deux implémentations sera la plus rapide ?

	ArrayList	LinkedList
void addFirst(E elem)		
void addLast(E elem)		
void add(E elem, int pos)		
E get(int pos)		
void removeFirst()		
void removeLast()		

Discussion

- Pour certaines opérations, lorsque l'une des implémentations est **rapide**, l'autre est **lente** ;
- En regardant le tableau ci-haut, **quand** devrait-on utiliser une implémentation à base de **tableaux** ?

- Quand devrait-on utiliser une liste chaînée ?

- Quelle implémentation consomme plus de **mémoire** ?

Implémentations

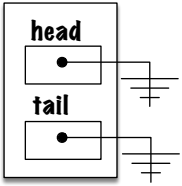
Référence vers le noeuds arrière

Accélérer addLast pour une liste simplement chaînée

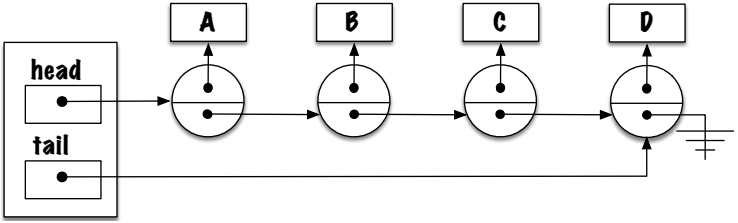
- Il y a une technique d'implémentation simple permettant d'**accélérer** l'ajout à l'**arrière** d'une liste chaînée.
- Qu'est-ce qui rend l'implémentation actuelle **coûteuse** ?
- Oui, il faut **parcourir la liste** d'un bout à l'autre afin d'ajouter l'élément à la toute fin.
- On pourrait bien sûr ajouter les éléments dans l'ordre inverse, mais ça ne ferait que déplacer le problème, la méthode **addFirst()** serait lente.
- Pour la méthode **size()**, nous avons vu que l'utilisation d'une variable d'instance supplémentaire, **count**, pouvait nous éviter de parcourir la liste inutilement.
- Que nous faudrait-il dans ce cas-ci pour **éviter un parcours** ?
- Oui, une nouvelle **variable d'instance** pointant sur le **dernier** élément de la liste.

Diagramme de mémoire

- Représentation de la **liste vide** :



- **Cas général** :



LinkedList

```
public class LinkedList<E> implements List<E> {

    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;
        }
    }

    private Node<E> head;
    private Node<E> tail;

    // ...
}
```

addLast

```
public void addLast(E elem) {

    Node<E> newNode;
    newNode = new Node<E>(elem, null);

    if (head == null) {
        head = newNode;
        tail = head;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
}
```

Modifier toutes les autres méthodes en conséquence

```
public E removeFirst() {

    E saved;
    saved = head.value;

    head = head.next;

    if (head == null) {
        tail = null;
    }

    return saved;
}
```

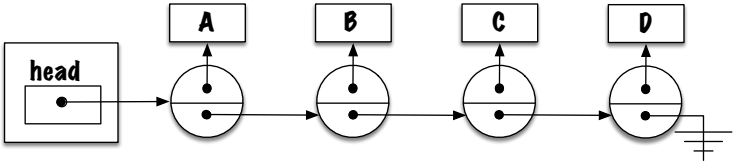
Comparer ArrayList et LinkedList

- Ajout d'une **référence arrière**.

	ArrayList	LinkedList
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	**constant**
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	variable

- Est-ce que **removeLast** est aussi plus **rapide** maintenant ?

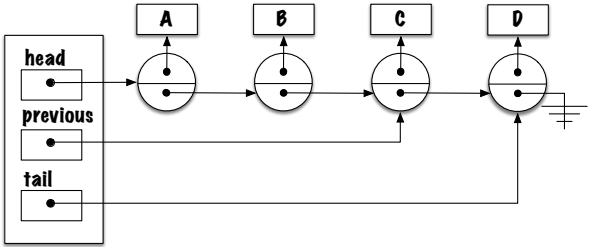
LinkedList



Implémentations

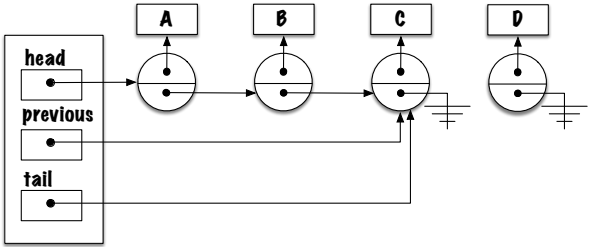
Noeuds doublement chaînés

Accélérer removeLast



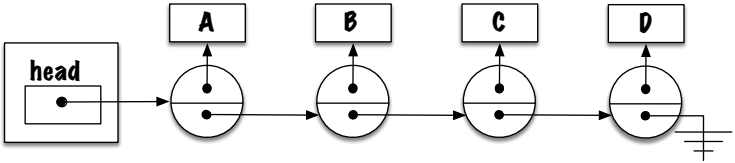
- Qu'en pensez-vous ?

Accélérer removeLast

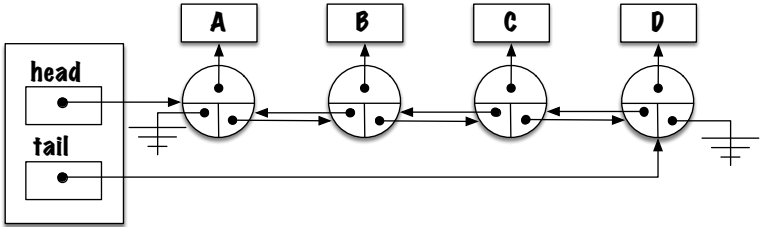


- Déplacer la référence arrière est maintenant facile et **rapide** !
- Sauf que le déplacement de la référence **previous** est difficile et coûteux.

LinkedList



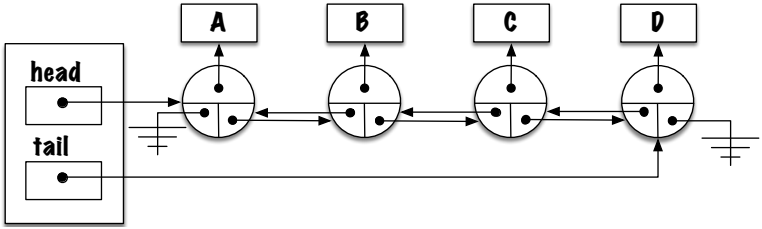
LinkedList



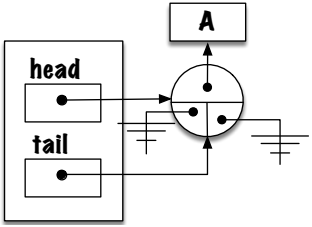
Liste doublement chaînée

```
public class LinkedList<E> implements List<E> {

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

  private Node<E> head;
  private Node<E> tail;

  // ...
}
```

removeLast : cas général



removeLast : cas spécial



```
public E removeLast() {

    E saved;
    saved = tail.value;

    if (head.next == null) {
        head = null;
        tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }

    return saved;
}
```

Comparer ArrayList et LinkedList

- Éléments **doublement** chaînés.

	ArrayList	**LinkedList**
void addFirst(E elem)	variable	constant
void addLast(E elem)	variable	constant
void add(E elem, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	**constant**

Discussion

- Quel sera l'**impact** de cette modification ?

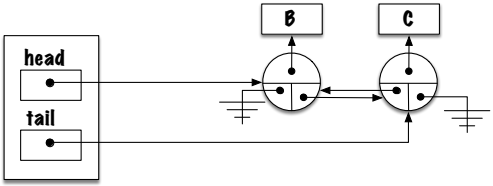
Préconditions : add(int pos, E elem)

- Quelles sont les **conditions préalables** de la méthode **add** ?

```
if (elem == null ) {
    throw new NullPointerException( "null" );
}
if (pos < 0 || pos > size) {
    throw new IndexOutOfBoundsException(pos);
}
```

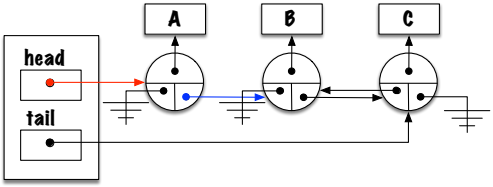
Cas spéciaux : add(int pos, E elem)

- Quels sont les **cas spéciaux** ?



Cas spécial : add(int pos, E elem)

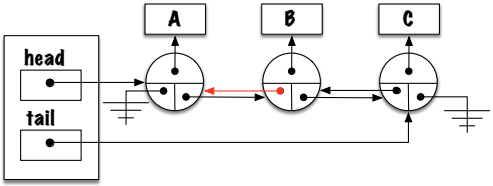
Cas spécial : head = new Node<E>(elem, null, head)



- **Qu'est-ce qui manque ?**

Cas spécial : add(int pos, E elem)

Cas spécial : head.next.previous = head



Cas spécial : add(int pos, E elem)

Cas spécial :

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    head.next.previous = head;
}
```

- Avons-nous pensé à **tous les cas possibles** ?
  - Et si la liste était **vide** ?

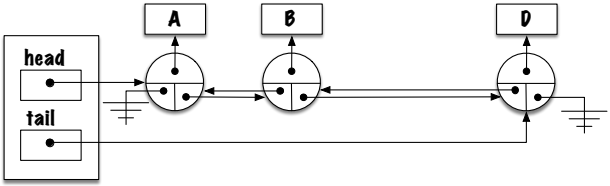
Cas spécial : add(int pos, E elem)

Cas spécial :

```
if (pos == 0) {
    head = new Node<E>(elem, null, head);
    if (tail == null) {
        tail = head;
    } else {
        head.next.previous = head;
    }
}
```

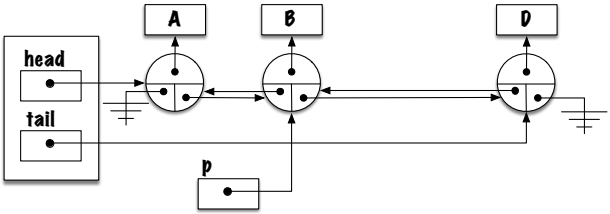
Cas général : add(int pos, E elem)

Cas général : ajout en position 2.



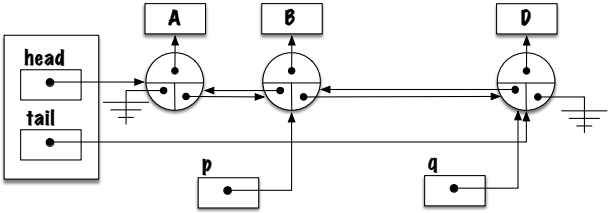
Cas général : add(int pos, E elem)

Cas général : traversons la liste jusqu'à **pos-1**



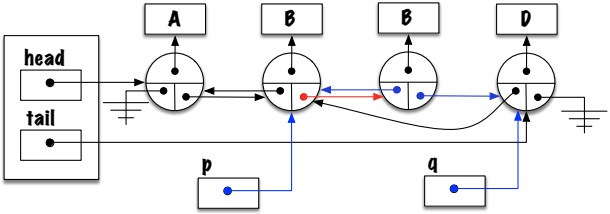
Cas général : add(int pos, E elem)

Cas général : q = p.next



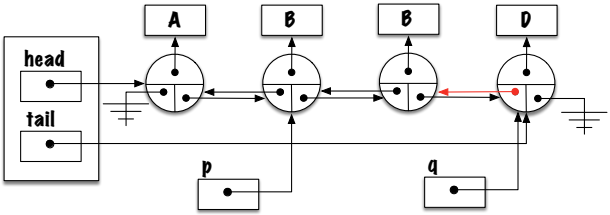
Cas général : add(int pos, E elem)

Cas général : p.next = new Node<E>(elem, p, q)



Cas général add(int pos, E elem)

Cas général : q.previous = p.next



Cas général add(int pos, E elem)

Cas général :

```
Node<E> before, after;
before = head;

for (int i = 0; i < (pos-1); i++) {
   before = before.next;
}

after = before.next;

before.next = new Node<E>(elem, before, after);
after.previous = before.next;
```

- Avons-nous pensé à tous les cas ?
  - Et si **before** désignait le dernier élément ?

add(int pos, E elem)

```
Node<E> before, after;

before = head;

for (int i = 0; i < (pos-1); i++) {
      before = before.next;
}

after = before.next;
before.next = new Node<E>(elem, before, after);

if (before == tail) {
  tail = before.next;
} else {
  after.previous = before.next;
}
```

Implémentations

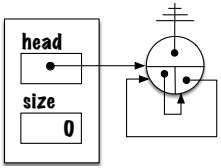
Noeud factice

Noeud factice

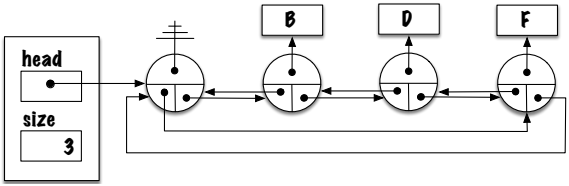
- La technique d'implémentation suivante permet d'**éliminer plusieurs cas spéciaux**.
  - La technique utilise un **noeud factice** (noeud fictif, noeud muet ou en anglais *dummy node*) ne contenant pas d'élément.
  - De plus, la liste est **circulaire** !

Noeud factice

Liste **vide** :



Cas **général** :

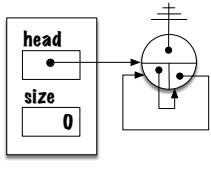


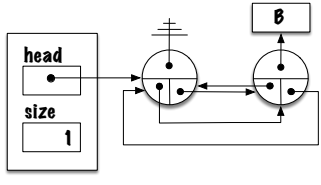
```
public class LinkedList<E> implements List<E> {
   private static class Node<T> {
      private T value;
      private Node<T> prev;
      private Node<T> next;
      private Node(T value, Node<T> prev, Node<T> next) {
         this.value = value;
         this.prev = prev;
         this.next = next;
      }
   }
   private Node<E> head;
```

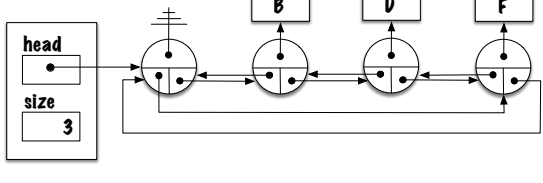
- Donnez l'**implémentation du constructeur**

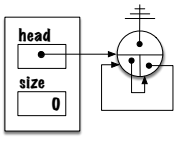
Discussion

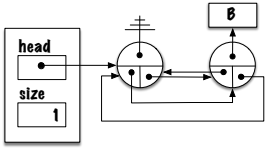
- **Qu'est-ce qui complique** l'implémentation des méthodes d'une liste chaînée sans noeud factice ?
  - Les méthodes ont généralement un **cas spécial** pour les modifications en **première position**.
  - En général, il faut changer la variable **next** du noeud qui précède, sauf si l'on traite le premier noeud, il faut alors changer la variable **head**.
  - Les modifications à la fin de la liste posent aussi un problème puisqu'il faut modifier la valeur de **tail**.
- Pour l'implémentation ayant un noeud factice, les traitements sont uniformes, on change toujours la variable **next** du noeud qui précède.

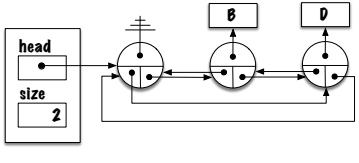

head
size
0

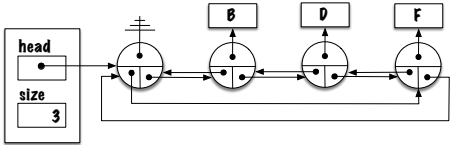

B
head
size
1


B
D
F
head
size
3


head
size
0


B
head
size
1


B
D
head
size
2


B
D
F
head
size
3

Prologue

Résumé

- Une référence vers l'**élément arrière** facile l'ajout à la **fin** de liste.
- Les **noeuds doublement chaînés** facilitent le retrait du **dernier** élément, mais aussi on peut parcourir la liste en sens inverse.
- La liste **circulaire** possédant un **noeud factice** n'a pas de cas particuliers !

Prochain module

- **Listes** : les itérateurs

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa