ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 14 mars 2011

Résumé

– Listes chaînées (partie 2)
  – Pointeur arrière
  – Listes doublement chaînées
  – Noeud factice (dummy node)

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Temps d'exécution

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.

Temps d'exécution

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

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

– 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 ? Pour les accès directs (aléatoires) en lecture ;
– Quand devrait-on utiliser une liste chaînée ? Si tous les accès se font au début de la liste ;
– Quelle implémentation consomme plus de mémoire ?

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.

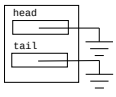
On pourrait bien sûr ajouter les éléments dans l'ordre inverse, mais ça ne ferait que déplacer le problème, la méthode **addFirst()** serait lente.

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.

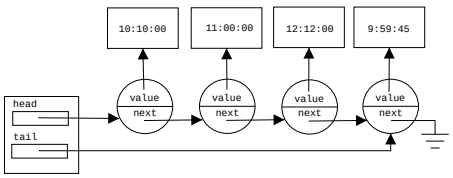
Que nous faudrait-il dans ce cas-ci pour éviter un parcours ?

Oui une nouvelle variable d'instance pointant sur le dernier élément de la liste.

Représentation de la liste vide :



Cas général :




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

    private static class Node<T> {

        private E 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;

    // ...
}
```

⇒ On a ajouté une variable d'instance, **tail**.

```
public void addLast( E o ) {

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

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

```
public E removeFirst() {

    Node<E> nodeToDelete = head;
    E result = nodeToDelete.value;

    head = head.next;

    noteToDelete.value = null; // ''scrubbing''
    noteToDelete.next = null;

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

    return result;
}
```

⇒ Toutes les méthodes sont modifiées en conséquence.

Temps d'exécution (révision 1)

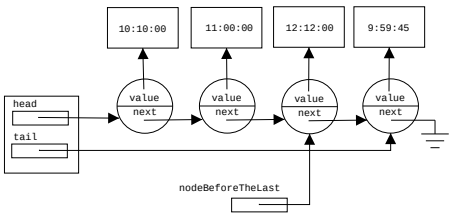
	ArrayList	LinkedList
void addFirst(E o)	variable	constant
void addLast(E o)	variable	**constant**
void add(E o, int pos)	variable	variable
E get(int pos)	constant	variable
void removeFirst()	variable	constant
void removeLast()	constant	variable

Discussion : cette modification a-t-elle un impact (favorable) sur la vitesse d'exécution de la méthode **removeLast()** ?

Non, aucun impact significatif.

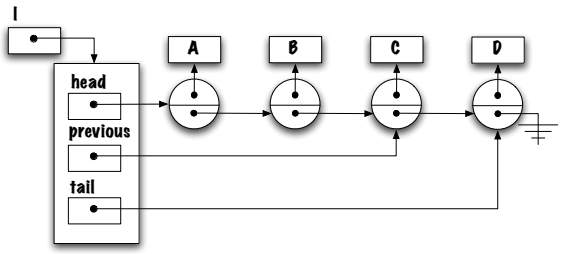
Accélérer removeLast()

Le pointeur **tail** ne nous aide en rien pour l'opération **removeLast**, il nous faut encore parcourir toute la liste.



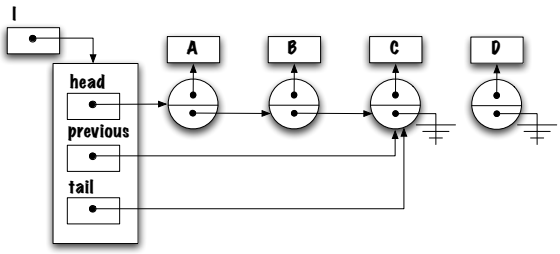
⇒ Qu'est-ce qu'il nous faut ? Que pensez-vous d'une nouvelle variable d'instance `previous` ?

Accélérer removeLast()



Qu'en pensez-vous ?

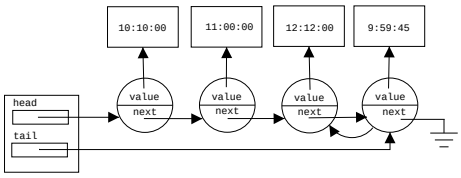
Accélérer removeLast()



Déplacer la référence arrière est maintenant facile et rapide !

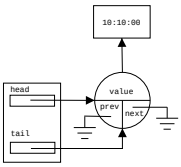
Sauf que le déplacement de la référence **previous** est difficile et coûteux.

Nous devons accéder à l'élément qui précède le dernier :

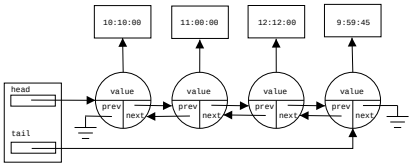


Mais aussi à tous ses prédécesseurs !

Liste vide :



Singleton :



Cas général :

```
public class DoublyLinkedList<E> implements List<E> {
    private static class Node<T> {
        private T value;
        private Node<T> previous; // <---
        private Node<T> next;
        private Node( T value, Node<T> previous, Node<T> next ) {
            this.value = value;
            this.previous = previous; // <---
            this.next = next;
        }
    }
    private Node<E> head;
    private Node<E> tail;
    public LinkedList() {
        head = tail = null;
    }
    // ...
}
```

⇒ C'est une liste doublement chaînée.

removeLast() (cas spécial : singleton)

removeLast() (cas général)

```
public E removeLast() {
    // pre-condition: ?

    Node<E> toDelete = tail;
    E savedValue = toDelete.value;

    if ( head.next == null ) {
        head = null;
        tail = null;
    } else {
        tail = tail.previous;
        tail.next = null;
    }
    toDelete.value = null;
    toDelete.next = null;

    return savedValue;
}
```

⇒ `removeLast()` sans parcours de la liste.

Temps d'exécution (révision 2)

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

Simple ? Pas si simple ?

Presque toutes les méthodes ont un ou des cas particuliers.

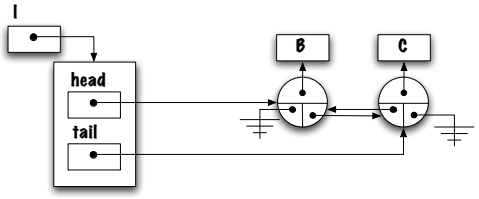
add(int pos, E o)

Pré-conditions ?

```
if ( o == null ) {
    throw new IllegalArgumentException( "null" );
}
if ( pos < 0 ) {
    throw new IndexOutOfBoundsException( Integer.toString( pos ) );
}
```

add(int pos, E o)

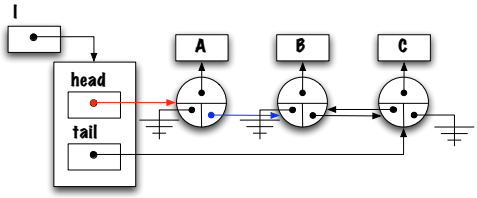
Cas spécial (spéciaux) ?



Ajout en position 0.

add(int pos, E o)

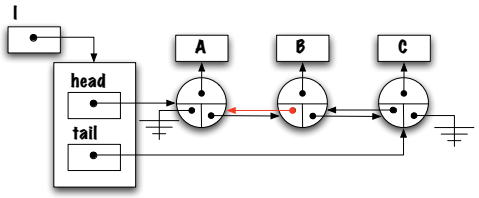
Cas spécial : `head = new Node<E>( o, null, head )`



Qu'est-ce qui manque ?

add(int pos, E o)

Cas spécial : `head.next.previous = head`



add(int pos, E o)

Cas spécial :

```
if ( pos == 0 ) {

    head = new Node<E>( o, null, head );
    head.next.previous = head;

}
```

Avons nous pensez à tous les cas possibles ?

Et si la liste était vide ?

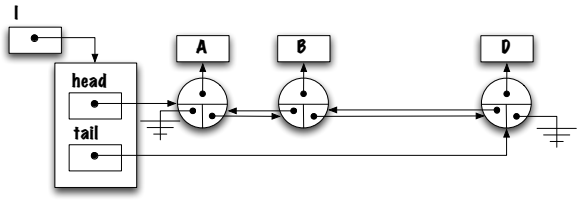
add(int pos, E o)

Cas spécial :

```
if ( pos == 0 ) {

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

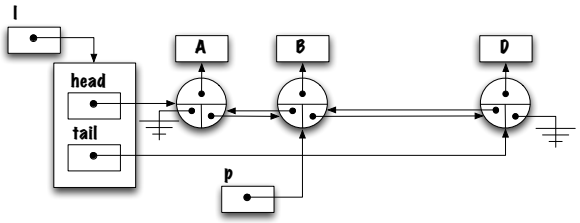
add(int pos, E o)

Cas général : ajout en position 2



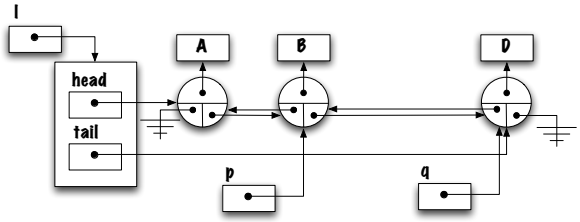
add(int pos, E o)

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



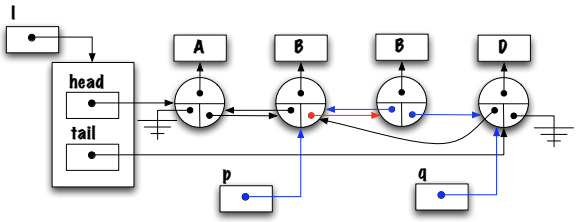
add(int pos, E o)

Cas général : **q = p.next**



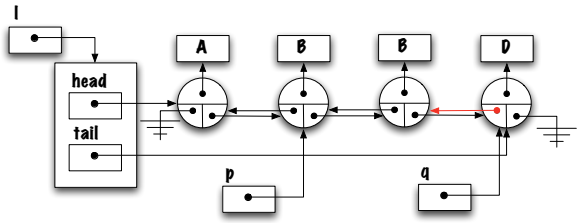
add(int pos, E o)

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



add(int pos, E o)

Cas général : **q.previous = p.next**



add(int pos, E o)

Cas général :

```
Node<E> p = head;

for ( int i = 0; i < (pos-1); i++ ) {
   p = p.next;
}
Node<E> q = p.next;

p.next = new Node<E>( o, p, q );
q.previous = p.next;
```

Avez-vous pensez à tous les cas ?

Et si **pos** était trop plus grand ?

add(int pos, E o)

Cas général :

```
Node<E> p = head;
for ( int i = 0; i < (pos-1); i++ ) {
  if ( p == null ) {
      throw new IndexOutOfBoundsException( Integer.toString( pos ) );
  } else {
      p = p.next;
  }
}
Node<E> q = p.next;
p.next = new Node<E>( o, p, q );
q.previous = p.next;
```

Avez-vous pensez à tous les cas ?

Et si l'ajout se faisait en dernière position ?

add(int pos, E o)

```
Node<E> p = head;
for (int i = 0; i < (pos-1); i++) {
  if ( p == null ) {
      throw new IndexOutOfBoundsException( Integer.toString( pos ) );
  } else {
      p = p.next;
  }
}
Node<E> q = p.next;
p.next = new Node<E>( o, p, q );
if ( p == tail ) {
  tail = p.next;
} else {
  q.previous = p.next;
}
```

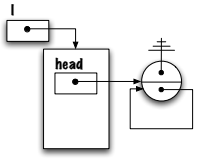
Noeud factice

La technique d'implémentation suivante permet d'éliminer plusieurs cas spéciaux.

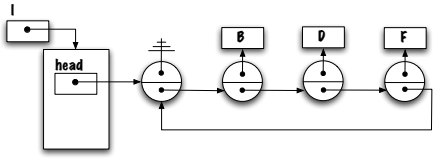
La technique utilise un noeud factice (*dummy node*) ne contenant pas d'élément.
De plus, la liste est circulaire.

Noeud factice

Liste vide :



Cas général :

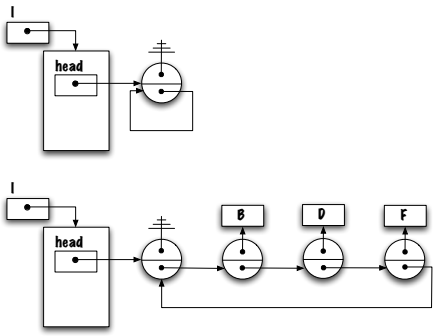


```
public class LinkedList<E> implements List<E> {

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

    public LinkedList() {
        head = new Node<E>( null, null ); // <---
        head.next = head;  // <---
    }
    // ...
}
```

```
// Ajout dans une liste simplement chainée (sans noeud factice)

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

Noeud factice (addLast)

Le nouvel élément sera ajouté après un noeud tel que . . .




```
// Noeud factice

public void addLast( E o ) {
    Node<E> p = head;
    while ( p.next != head ) {
        p = p.next;
    }
    p.next = new Node<E>( t, head );
}
```

Remarques (noeud factice)

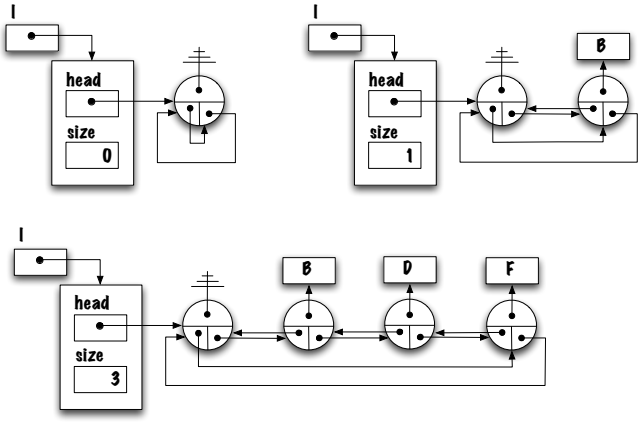
Qu'est-ce qui complique l'implémentation des méthodes d'une liste chaînée sans noeud factice ?

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**.

Pour l'implémentation ayant un noeud factice, les traitements sont uniformes, on change toujours la variable **next** du noeud qui précède.

Les noeuds de la liste peuvent aussi être doublement chaînés. On peut aussi ajouter une variable afin de compter le nombre d'éléments.



Collection

En Java, les classes utilisées pour sauvegarder des éléments sont regroupées dans une hiérarchie dont la racine est **Collection**.

Les collections sont linéaires, hiérarchiques, arborescentes ou non ordonnées.

Les collections linéaires sont les listes, piles et files. Tous les éléments ont un prédécesseur et un successeur (à l'exception du premier et du dernier élément).

Les arborescences permettent de représenter des arbres et des structures hiérarchiques.

Les graphes permettent de représenter les distances entre les villes, par exemple.

Les collections non ordonnées incluent les ensembles et les tables de hashage.