ITI 1521. Introduction à l'informatique II*

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

Version du 20 mars 2011

Résumé

– Itérateur[1] (Partie 1)

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

1. Excellente introduction à l'analyse asymptotique (complexité du calcul, big-O), encapsulation, programmation orientée objet, interfaces et plus.

Motivation

Pour une implémentation (simplement) chaînée de l'interface **List** définie comme suit,

```
public interface List<E> {
    public abstract boolean add( E obj );
    public abstract E get( int index );
    public abstract boolean remove( E obj );
    public abstract int size();
}
```

Nous nommerons cette implémentation **LinkedList**.

Vous devez concevoir une méthode pour traverser la liste, par exemple afin d'afficher toutes les valeurs.

L'implémentation et son temps d'exécution diffèrent selon que la méthode se trouve à l'intérieur ou à l'extérieur de la classe.

$\Rightarrow$ Les difficultés seraient les mêmes si la liste était doublement chaînée.

Exemple

```
List<String> colors;
colors = new LinkedList<String>();

colors.add( "bleu" );
colors.add( "blanc" );
colors.add( "rouge" );
colors.add( "jaune" );
colors.add( "vert" );
colors.add( "orange" );
```

(A) — traverser la liste de l'intérieur

De l'intérieur, nous avons accès aux détails de l'implémentation, accès aux noeuds.

```
Node p = head;
while ( p != null ) {
    System.out.println( p.value );
    p = p.next;
}
```

(B) — traverser la liste de l'extérieur

De l'extérieur, nous devons utiliser **E get(int pos)**.

```
for ( int i=0; i < colors.size(); i++ ) {
    System.out.println( colors.get( i ) );
}
```

Comparez la vitesse des deux implémentations (intérieur et extérieur). L'implémentation de l'intérieur est plus rapide ou plus lente ? Les différences sont mineures ou majeures ?

Temps d'exécution

Voici les temps d'exécution pour des listes contenant $20,000$, $40,000$ et $80,000$ noeuds, moyenne de 5 exécutions (temps en millisecondes).

# noeuds	A	B
20,000	2	20,644
40,000	5	94,751
80,000	12	407,059

$\Rightarrow$ Pour $80,000$, noeuds il faut environ 6.5 minutes pour traverser la liste à l'aide de **get(pos)**, alors qu'il suffit 12 ms pour l'autre approche. Comment expliquer cette différence ?

```java
for ( int i=0; i< names.size(); i++ ) {
    System.out.println( names.get( i ) );
}
```

Appel	Nombre de noeuds visités
get(0)	1
get(1)	2
get(2)	3
get(3)	4
...	...
get(n-1)	n

Ainsi le nombre total de noeuds visités sera,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx n^2$$

Implémentation **A** visite n noeuds, implémentation **B** visite n^2 noeuds !

# noeuds	A	B
20,000	2	20,644
40,000	5	94,751
80,000	12	407,059

L'exécution des énoncés qui suivent (implémentation B).

```
for ( int i=0; i< names.size(); i++ ) {
    System.out.println( names.get( i ) );
}
```

Correspond à ceci de l'intérieur.

```
for ( int i=0; i< size(); i++ ) {
    Node p = head;
    for ( int j=0; j<i; j++ ) {
        p = p.next;
    }
    System.out.println( p.value );
}
```

Plutôt que (implémentation A).

```
Node p = head;
while ( p != null ) {
    System.out.println( p.value );
    p = p.next;
}
```

Peut-on faire mieux ?

Objectif : concevoir une approche afin de traverser la liste une et une seule fois.

L'utilisateur de la liste n'a pas accès à l'implémentation (`p.next` et autres) !

(!) La solution proposée sera applicable dans un contexte bien spécifique, LORSQUE TOUS LES NOEUDS DE LA LISTE SONT VISITÉS DE FAÇON SÉQUENTIELLE.

CE N'EST PAS UNE SOLUTION GÉNÉRALE POUR ACCÉLÉRER **get(i)**.

Itérateur : introduction

– L'itérateur est mécanisme uniforme afin de traverser une variété de structures de données, telles que les listes, mais aussi les arbres et autres (voir CSI 2514) ;
– Donne accès aux éléments un à la fois ;
– Fait partie des collections de Java.

Concept

```
≪créer un itérateur≫ (* positionné à gauche de la liste *)

tant que ( ≪il y encore des éléments≫ ) {
  ≪déplacer l'itérateur vers l'avant et retourner la valeur≫
  ≪utiliser la valeur≫
}
```

Concept

Interface Iterator

Séparons le concept d'itérateur de son implémentation.

```
public interface Iterator<E> {
    public abstract E next();
    public abstract boolean hasNext();
}
```

```java
/**
 * An iterator for lists that allows the programmer to traverse the
 * list.
 */
public interface Iterator<E> {
    /**
     * Returns the next element in the list.  This method may be called
     * repeatedly to iterate through the list.
     *
     * @return the next element in the list.
     * @exception NoSuchElementException if the iteration has no next element.
     */

    public abstract E next();

    /**
     * Returns <tt>true</tt> if this list iterator has more elements when
     * traversing the list in the forward direction. (In other words, returns
     * <tt>true</tt> if <tt>next</tt> would return an element rather than
     * throwing an exception.)
     *
     * @return <tt>true</tt> if the list iterator has more elements when
     *          traversing the list in the forward direction.
     */

    public abstract boolean hasNext();
}
```

Difficultés

- Quelle classe implémentera **Iterator** ?
- Comment créer et initialiser un itérateur ?
- Comment implémenter les déplacements ?
- Comment détecter la fin de l'itération ?

Implémentation -1-

Développons une première implémentation qui sera assez différente de l'implémentation finale, mais qui sera une bonne étape intermédiaire (les premières implémentations de Java avaient un mécanisme afin de traverser les listes qui ressemble beaucoup à celui que nous proposons).

Pour ce faire, la classe **LinkedList** implémente l'interface **Iterator**.

```
public class LinkedList<E> implements List<E>, Iterator<E> {
   private static class Node<E> { ... }
   private Node<E> head;
   public E next() { ... }
   public boolean hasNext() { ... }
   public Iterator<E> iterator() { ... }
   // ...
}
```

Usage

```
List<Integer> l;
l = new LinkedList<Integer>();

// Ajout d'éléments ...

// Positionner l'itérateur à gauche de la liste

Iterator<Integer> i;
i = l.iterator();

while ( i.hasNext() ) {
    Integer value = i.next();
    // traitements
}
```

Remarques

1. Pourquoi positionne-t-on l'itérateur à gauche de la liste et non sur le premier élément ?
   - La liste pourrait être vide.
2. Contrat de la méthode ≪**next()**≫ :
   (a) **Déplace l'itérateur d'une position vers l'avant** ;
   (b) **retourne la valeur de l'élément courant**.
3. Un danger vous guette lorsque vous utilisez la méthode **next()** ? Quel est-il ?
   - Regardez le contrat de la méthode **next()**, l'itérateur se déplace, puis retourne la valeur de l'élément courant. S'il n'y a pas d'élément suivant, une erreur surviendra.
   - Quels sont les deux cas particuliers (spéciaux) ?
     - La liste vide ;
     - L'itérateur était positionné sur le dernier élément au moment de l'appel.

Idiome :

```
LinkedList<Integer> l;
l = new LinkedList<Integer>();
Iterator<Integer> i;
i = i.iterator();
while ( i.hasNext() ) {
    Integer value = i.next();
    // ...
}
```

- Est-ce que la liste vide est traitée correctement ?
  - Oui. Le corps de la boucle n'est pas exécuté si la liste est vide.
- Est-ce que la fin de la liste est traitée correctement ?

Exemple : calcul de la somme

```
public class Sum {

    public static void main( String[] args ) {
        List<Integer> l = new LinkedList<Integer>();
        for ( int i=0; i<5; i++ ) {
            l.add( new Integer( i ) );
        }
        int sum = 0;

        Iterator<Integer> i = l.iterator();
        while ( i.hasNext() ) {
            Integer v = i.next();
            sum += v.intValue();
        }
        System.out.println( "sum = " + sum );
    }
}
```

Implémentation -1- (suite)

Il nous faut maintenant implémenter les méthodes de l'interface.

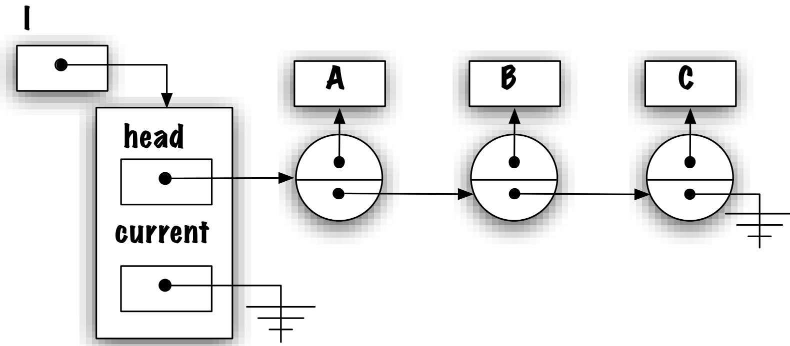
Quelles sont les variables d'instance nécessaires ?

Il n'en faut qu'une seule, **current**.

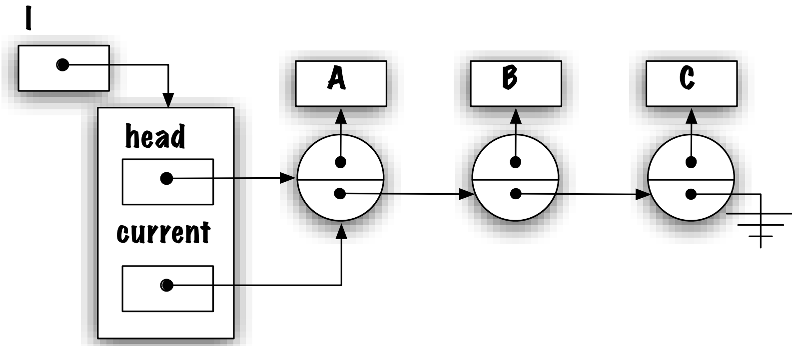
Nous utiliserons la valeur **null** afin de représenter l'état initial de l'itérateur, i.e. l'itérateur à la gauche de la liste.

Lors du premier appel, la méthode **next()** positionne la variable **current** sur le premier noeud.

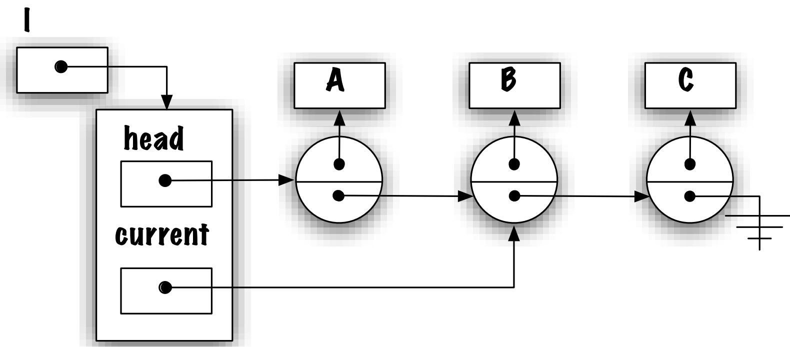
Pour chaque appel subséquent, **next()** déplace **current** d'une position vers l'avant.



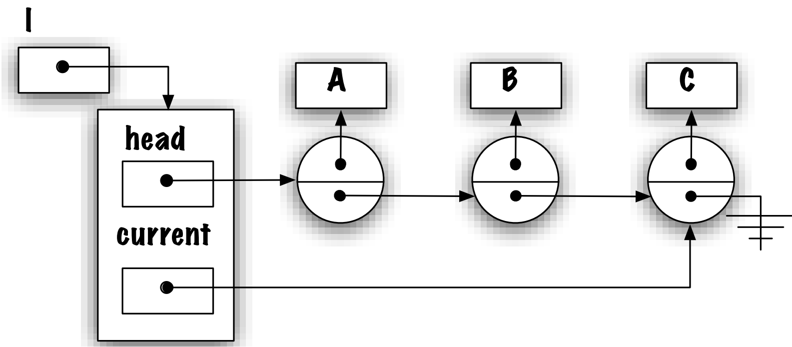
Iterator i = l.iterator() ;



suite à i.next() ;



suite à i.next() ;



suite à i.next() ;

Méthode **iterator()**.

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

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current; // <---

    public Iterator<E> iterator() {
        current = null;
        return this;
    }
}
```

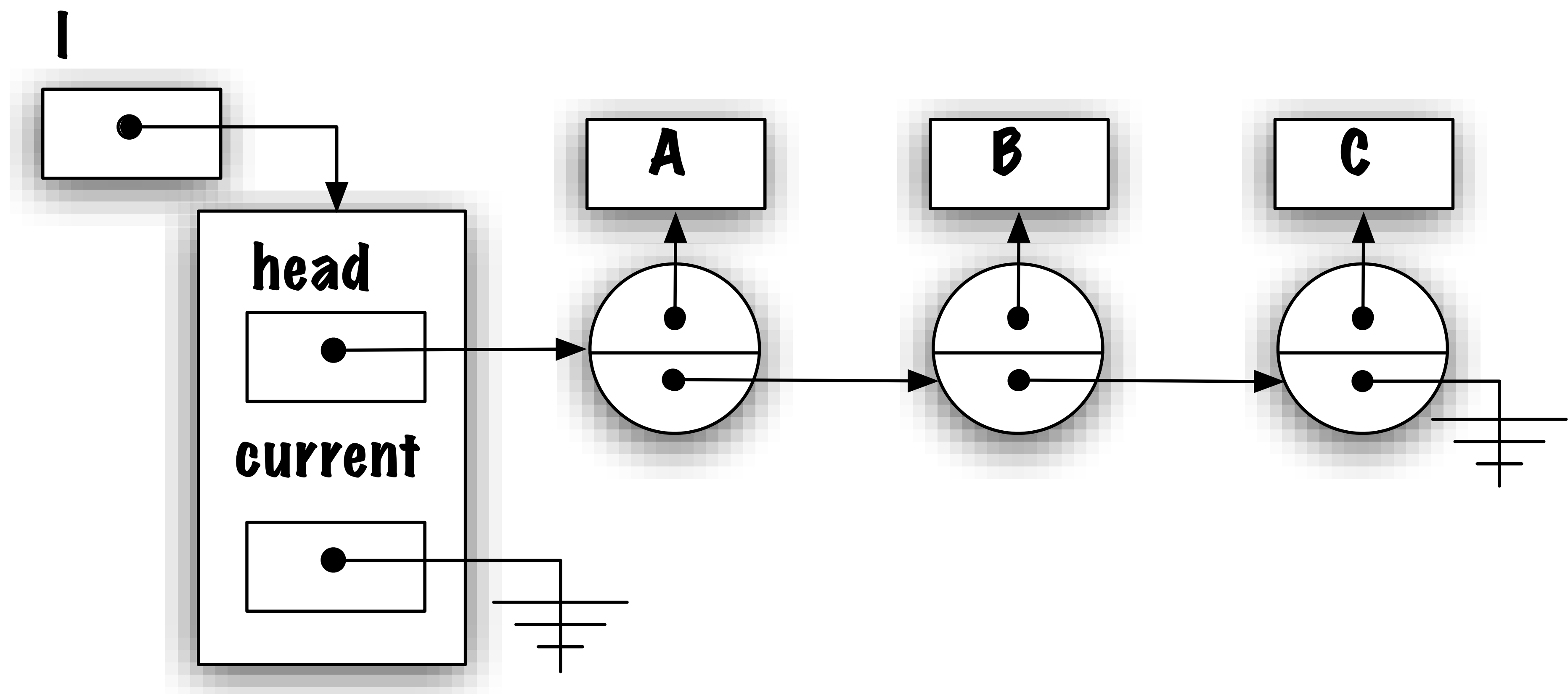
La méthode **iterator()** est-elle vraiment nécessaire ?

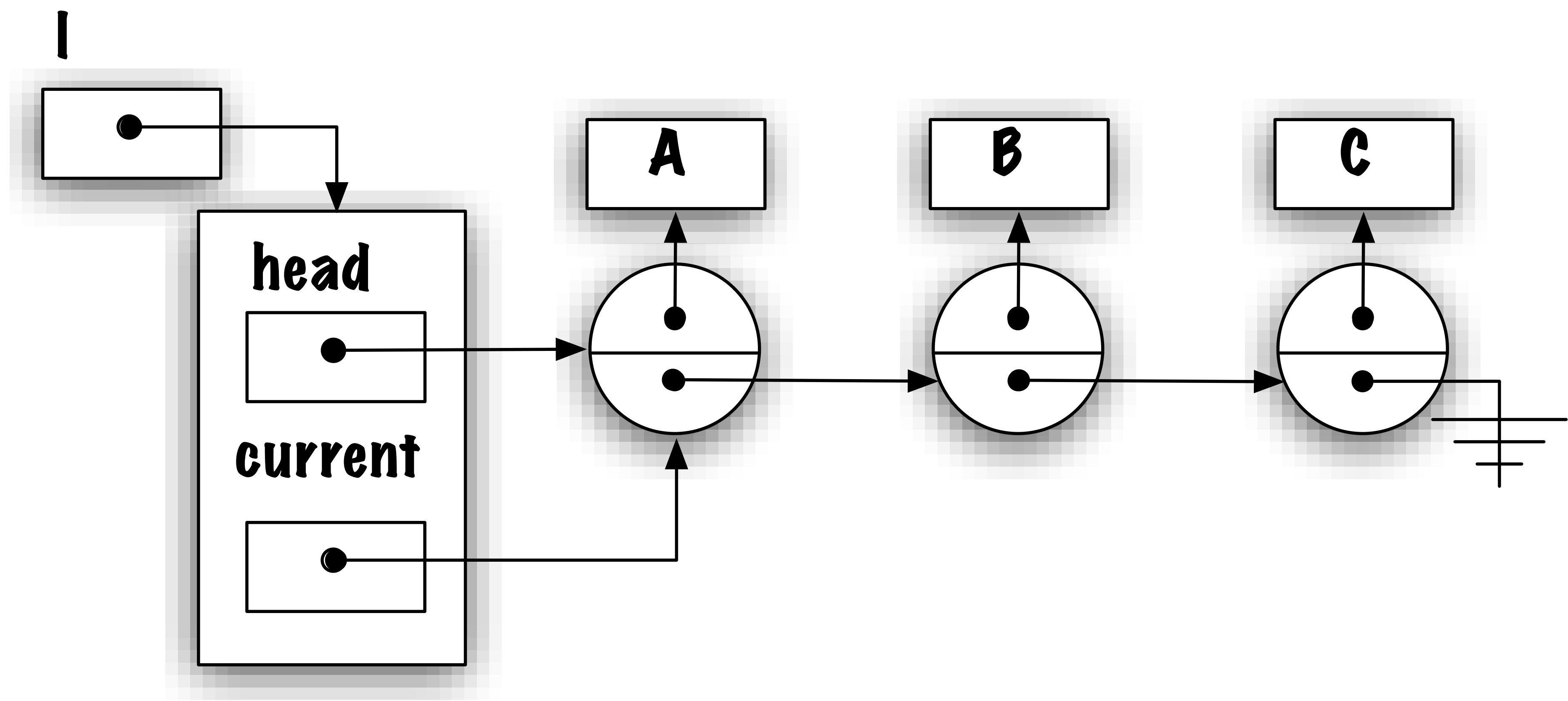
Entre autre, la méthode repositionne l'itérateur à gauche de la liste.

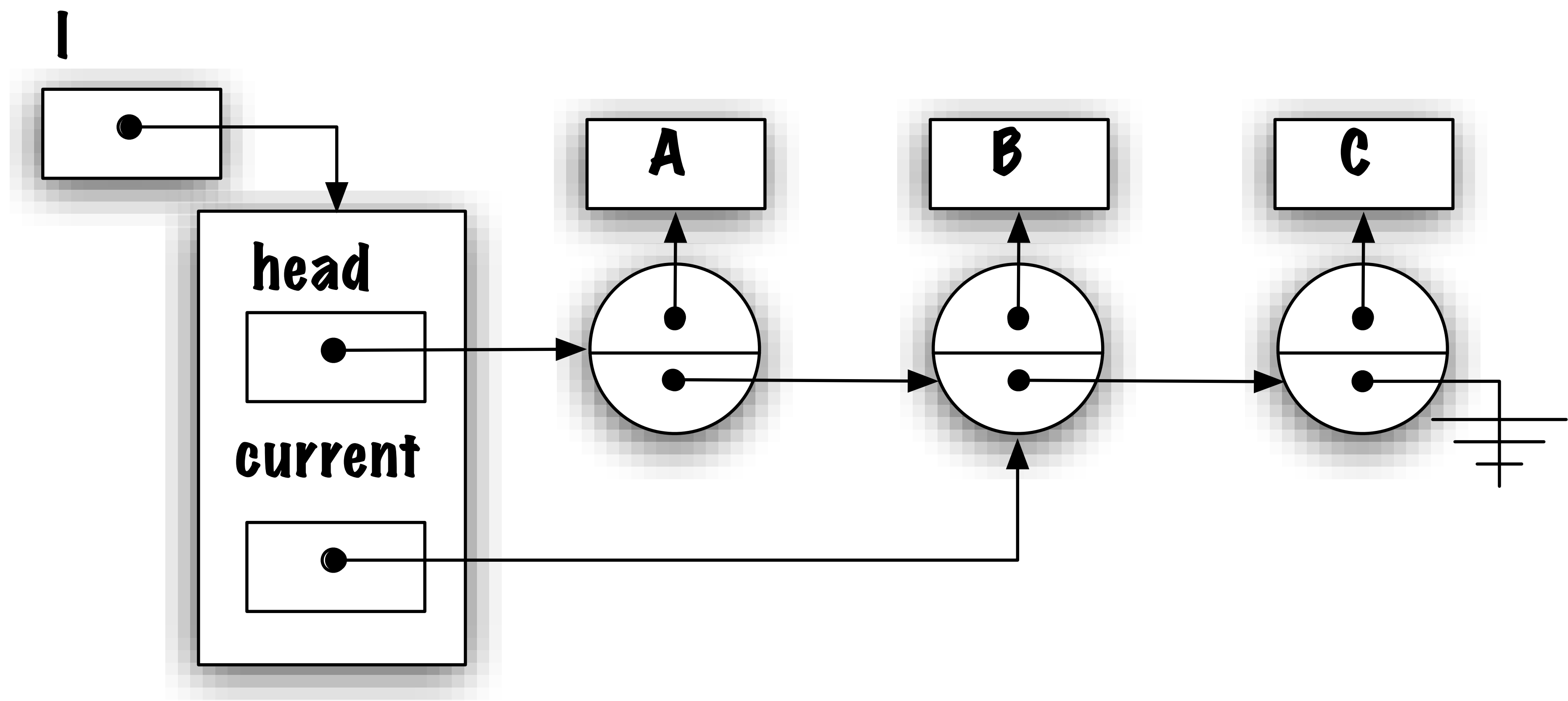
```
int sum = 0; int n = 0;
Iterator<Integer> i = l.iterator();
while ( i.hasNext() ) {
    Integer v = i.next();
    sum += v.intValue();
    n++;
}

i = l.iterator();
while ( i.hasNext() ) {
    Integer v = i.next();
    if ( v.intValue() > ( sum / n ) ) {
        System.out.println("est plus grand que la moyenne");
    }
}
```

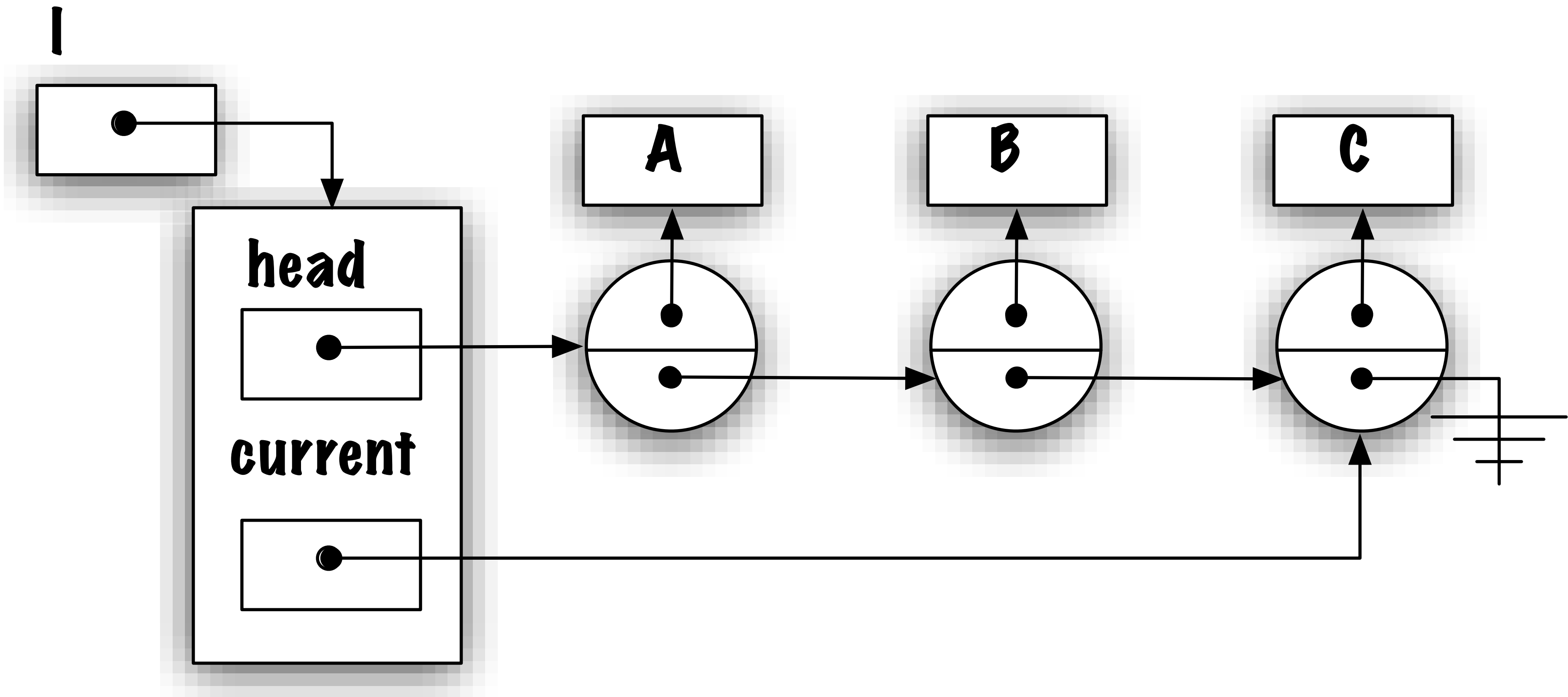
Méthode **next()**.

```
public class LinkedList<E> implements List<E>, Iterator<E> {
    private static class Node<E> { ... }
    private Node<E> head;
    private Node<E> current;
    public Iteraror<E> iterator() {
        current = null;
        return this;
    }
    public E next() {
        if ( current == null ) {
            current = head;
        } else {
            current = current.next;
        }
        if ( current == null ) {
            throw new NoSuchElementException();
        }
        return current.value;
    }
}
```


l
head
current
A
B
C


l
A
B
C
head
current


l
A
B
C
head
current


l
A
B
C
head
current

Limitations

Un seul itérateur. On ne peut pas concevoir des algorithmes tels que celui-ci à l'aide de cette implémentation.

```
while ( i.hasNext() ) {
    oi = i.next();
    while ( j.hasNext() ) {
        oj = j.next();
        // traite oi et oj
    }
}
```

L'implémentation de l'algorithme de tri par sélection serait difficile.

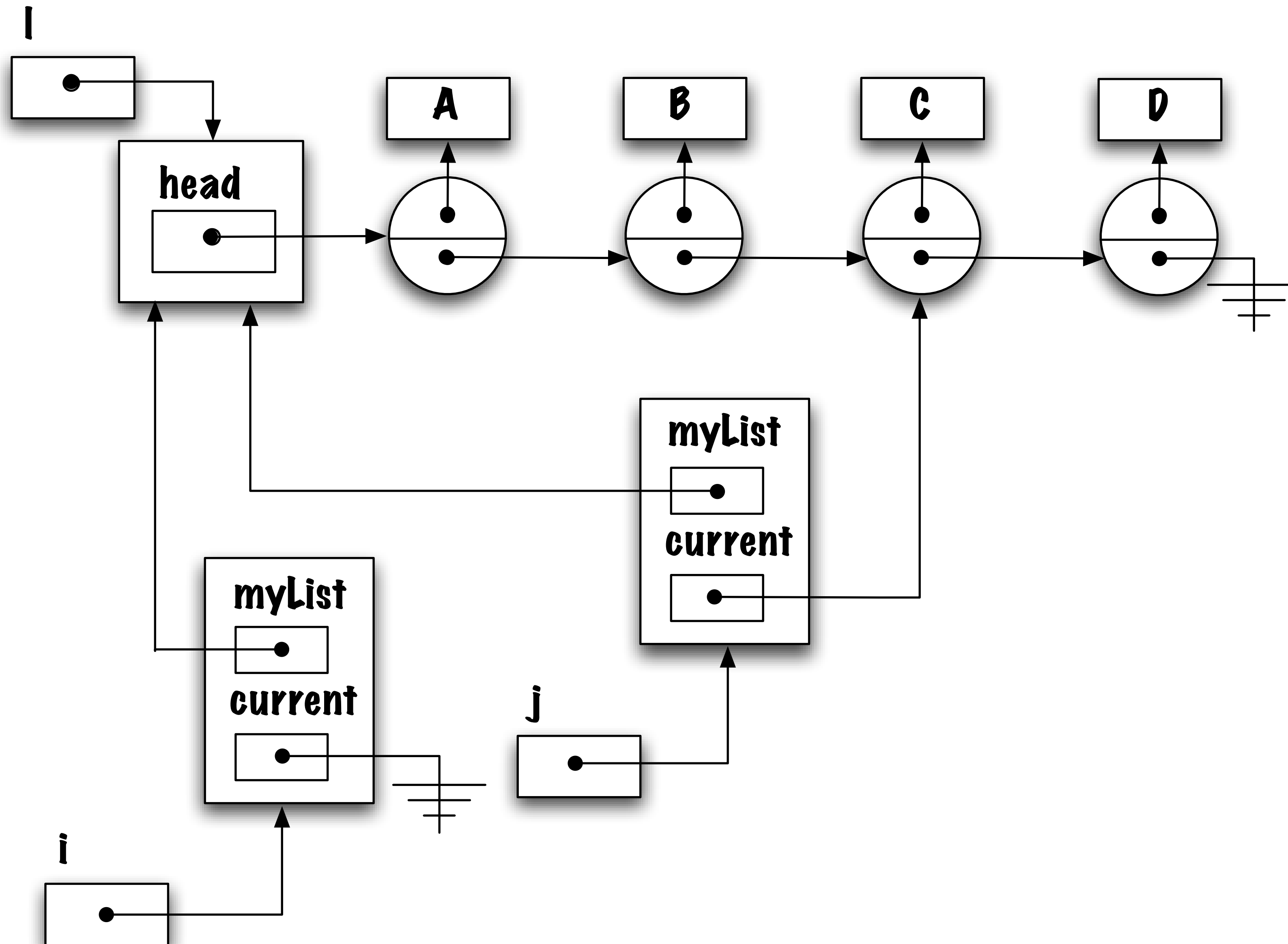
Que faut-il ?

- Chaque itérateur désigne un noeud ;
- **Les méthodes de l'itérateur doivent accéder l'implémentation de la classe Node** ;
- **Les méthodes de l'itérateur doivent accéder l'implémentation de la classe LinkedList**.

Implémentation -1.5-

Nous allons créer une classe de premier niveau ayant une variable d'instance désignant la liste de cette itérateur, ici **LinkedListIterator**, **Node** et **LinkedList** sont toutes des classes du même package, et la variable **head** est de visibilité **protected**.

```
public class LinkedListIterator<E> implements Iterator<E> {

    private Node<E> current;
    private LinkedList<E> myList;

    private LinkedListIterator( LinkedList<E> myList ) {
        this.myList = myList;
        current = null;
    }
    public boolean hasNext() {
        return ( ( current == null && myList.head != null ) ||
                 ( current != null && current.next != null ) );
    }
    public E next() {
        if ( current == null ) {
            current = myList.head;
        } else {
            current = current.next;
        }
        return current.value;
    }
}
```


l
head
A
B
C
D
myList
current
myList
current
j
i

La méthode **iterator()** de la classe **LinkedList** est définie comme suit :

```
public Iterator<E> iterator() {
    return new LinkedListIterator<E>( this );
}
```

Implémentation -1.75-

LinkedListIterator est maintenant une classe ≪static≫ imbriquée de la classe **LinkedList** ; ainsi la visibilité de la variable **head** est remise à **private**.

```
private static class LinkedListIterator<E> implements Iterator<E> {

    private Node<E> current;
    private LinkedList<E> myList;

    private LinkedListIterator( LinkedList<E> myList ) {
        this.myList = myList;
        current = null;
    }
    public boolean hasNext() {
        return ( ( current == null && myList.head != null ) ||
                 ( current != null && current.next != null ) );
    }
    public E next() {
        if ( current == null ) {
            current = myList.head;
        } else {
```

```
            current = current.next;
        }
        return current.value;
    }
} // end of LinkedListIterator

public Iterator<E> iterator() {
    return new LinkedListIterator<E>( this );
}
```