ITI 1521. Introduction à l'informatique II*

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

Version du 26 mars 2011

Résumé

– Traitement récursif des listes à l'extérieur de la classe **LinkedList**

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

Méthodes génériques

En plus des types génériques (interfaces et classes ayant des paramètres formels de type), les méthodes aussi peuvent avoir un paramètre de type.

```
public static < T extends Comparable<T> > T max( T a, T b ) {
    if ( a.compareTo( b ) > 0 ) {
        return a;
    } else {
        return b;
    }
}
```

Utiliser une méthode générique

Un appel de méthode pour une méthode générique ne requiert aucune construction syntaxique particulière. Les arguments de type sont inférés automatiquement.

```
Integer i1, i2, iMax;

i1 = new Integer( 1 );
i2 = new Integer( 10 );

iMax = max( i1, i2 );

System.out.println( "iMax = " + iMax );
```

Ici, le compilateur infère le type **Integer**.

Utiliser une méthode générique

```
String s1, s2, sMax;

s1 = new String( "alpha" );
s2 = new String( "bravo" );

sMax = max( s1, s2 );

System.out.println( "sMax = " + sMax );
```

Ici, le compilateur infère le type **String**.

FAQ

www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html

Introduction

L'implémentation actuelle ne permet pas l'écriture de méthodes récursives à l'extérieur de la classe **LinkedList**.

```
public int size() {
    return size( first );
}
private int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

Discussion

Quels changements permettraient le traitement récursif des listes à l'extérieur de la classe **LinkedList** ?

On ne peut rendre les noeuds de la liste visibles. Les autres classes pourraient alors changer la structure de la liste, ce qui causerait des interférences avec l'implémentation des itérateurs, entre autres.

Discussion

```
public static <E> int indexOf( LinkedList<E> xs, E obj ) {
    return indexOfRec( 0, xs, obj );
}

private static <E> int indexOfRec( int current, LinkedList<E> xs, E obj ) {
    int pos;
    if ( current == xs.size() ) {
        pos = -1;
    } else if ( obj.equals( xs.get( current ) ) ) {
        pos = current;
    } else {
        pos = indexOfRec( current + 1, xs, obj );
    }
    return pos;
}
```

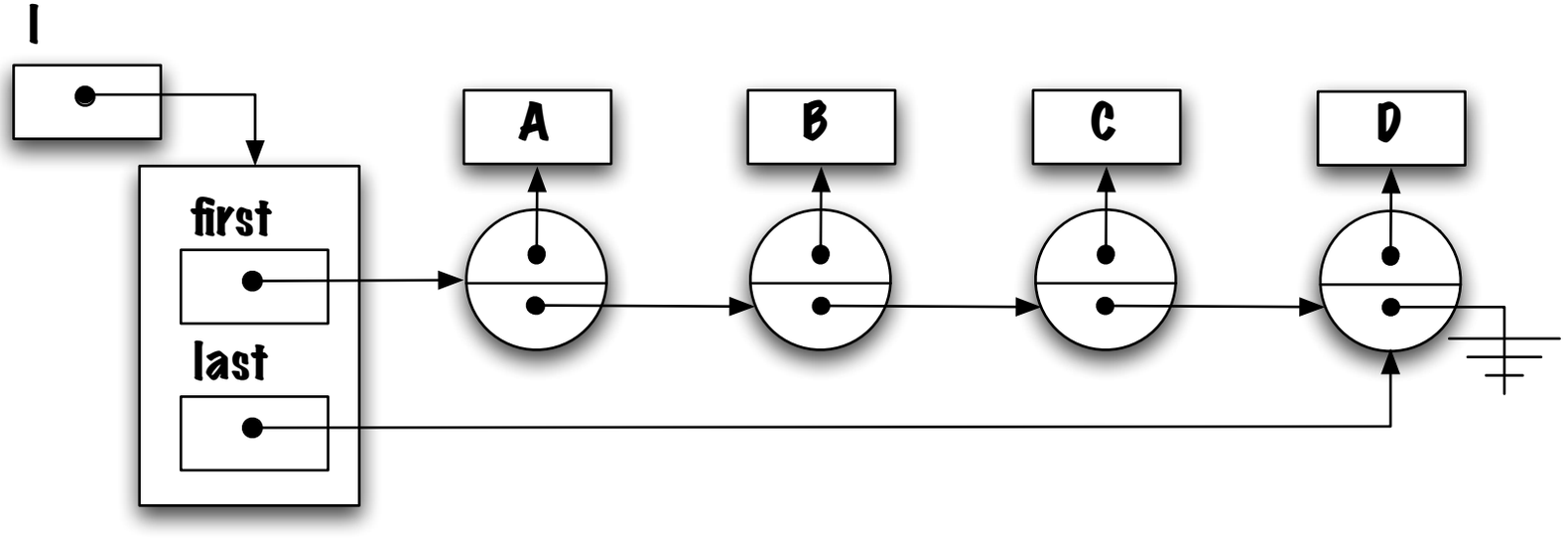
Les mécanismes proposés devront nous permettre d'éviter de retraverser la liste à partir du début pour tous les appels récursifs successifs.

Discussion

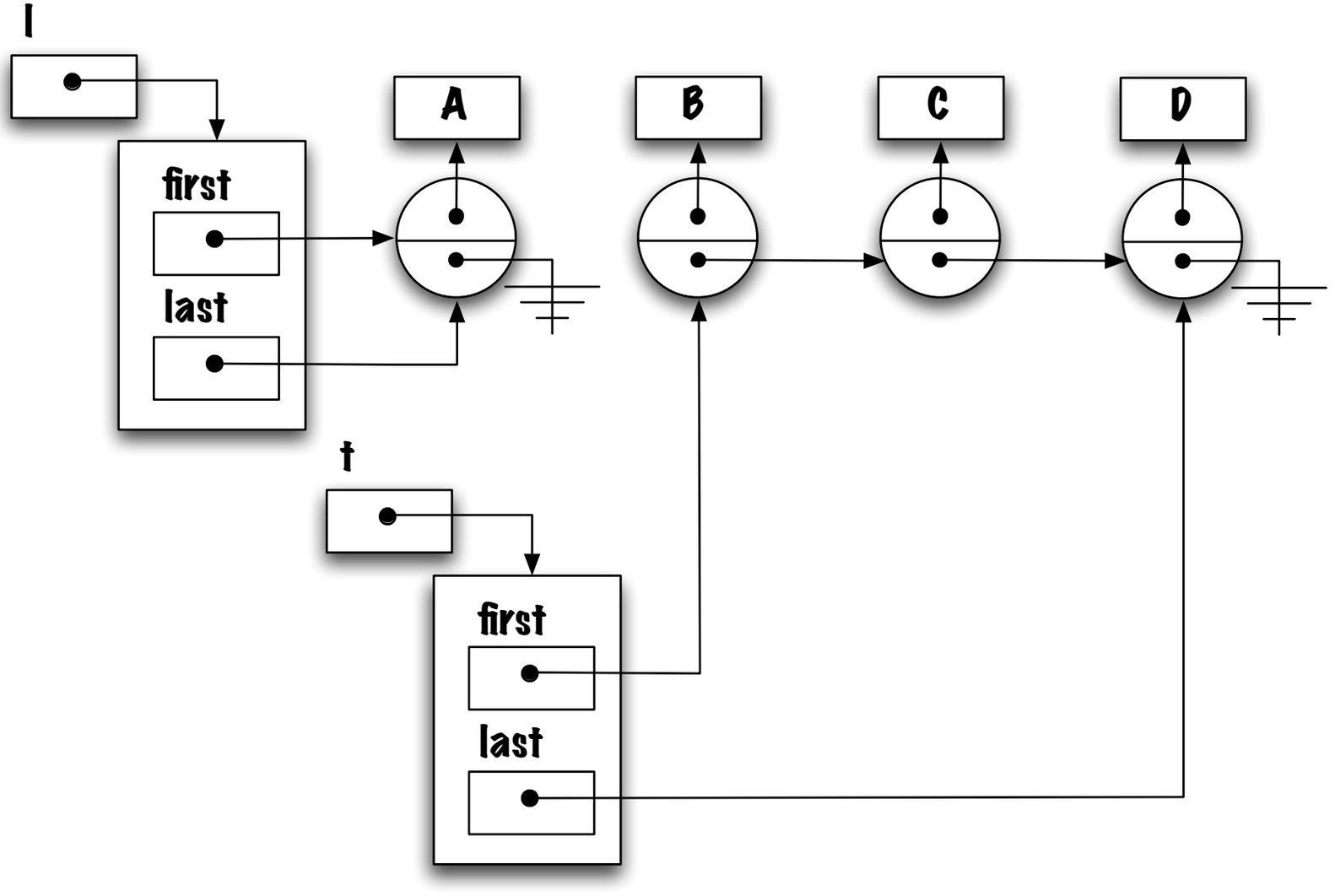
Afin d'éviter toute confusion, je nommerai cette nouvelle implémentation **Sequence**. C'est une structure de données linéaire utilisant des noeuds chaînés qui possède aussi des méthodes supplémentaires permettant les traitements récursifs de l'extérieur.

Sequence<E> split()

Étant donné l'instance **l**.

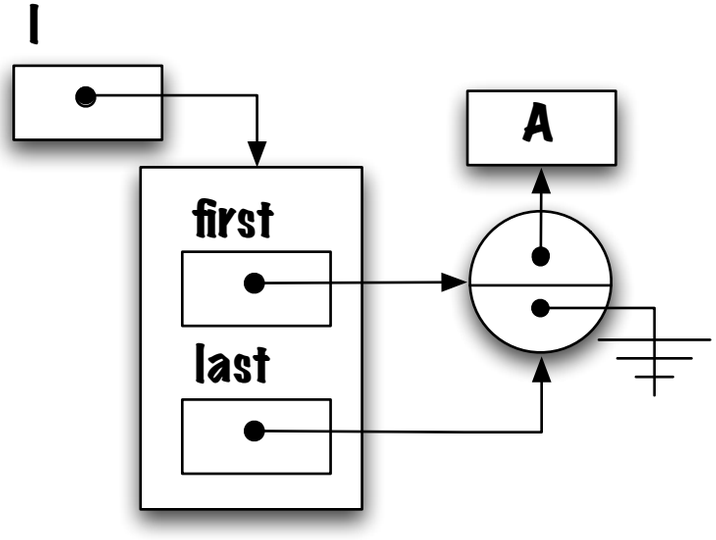


Sequence<E> t = l.split() retourne le reste de la liste et transforme **l** de sorte que **l** ne contient que son premier élément.

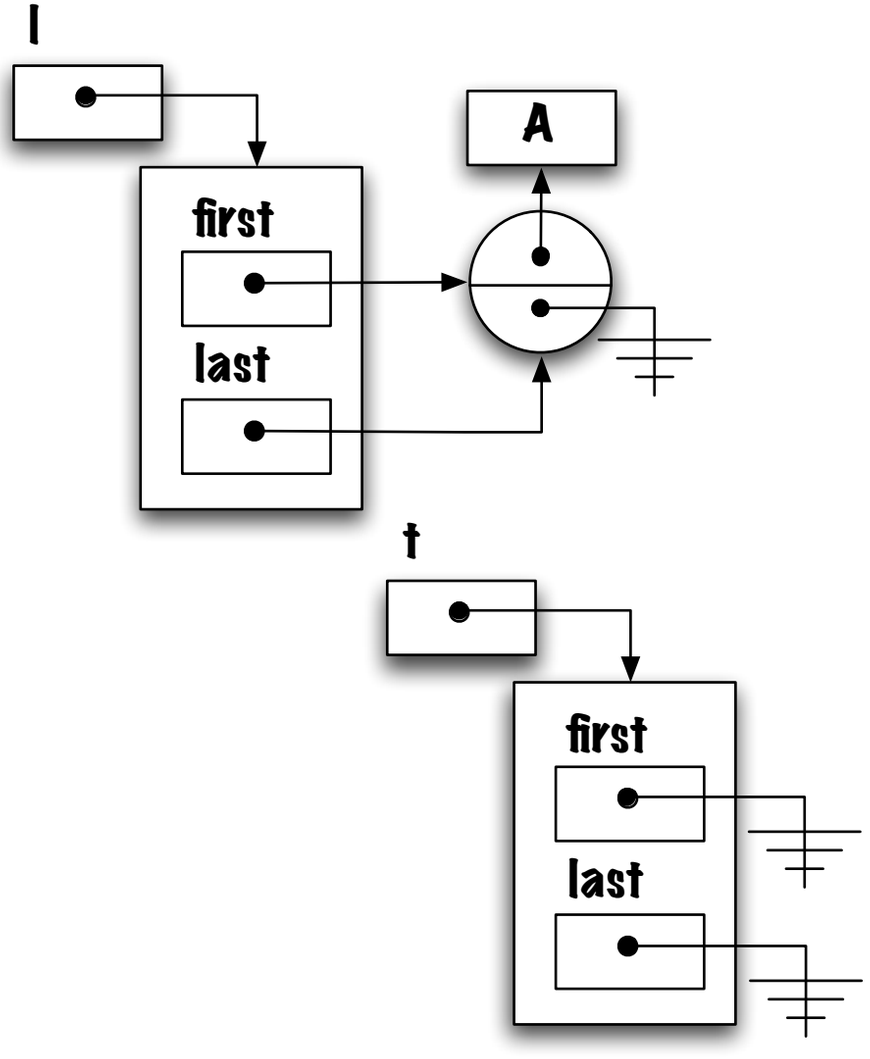


Sequence<E> split()

Étant donné l'instance **l**.

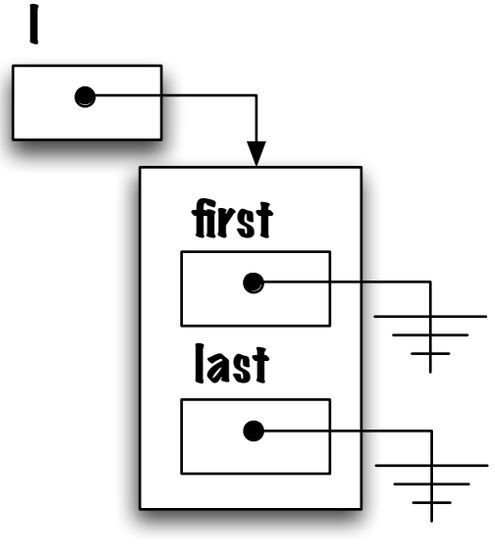


Sequence<E> t = l.split() retourne le reste de la liste et transforme **l** de sorte que **l** ne contient que son premier élément.



Sequence<E> split()

Étant donné l'instance **l**.



Sequence<E> t = l.split() . . . lance une exception.

Sequence<E> split()

Est-ce utile ?

Pouvons-nous l'utiliser afin d'implémenter des méthodes récursives à l'extérieur de la classe **Sequence** ?

public static <E> int size(Sequence<E> l)

Que pensez-vous de ceci ?

```
public static <E> int size( Sequence<E> l ) {

    if ( l.isEmpty() ) {
        return 0;
    } else {
        return 1 + size( l.split() );
    }

}
```

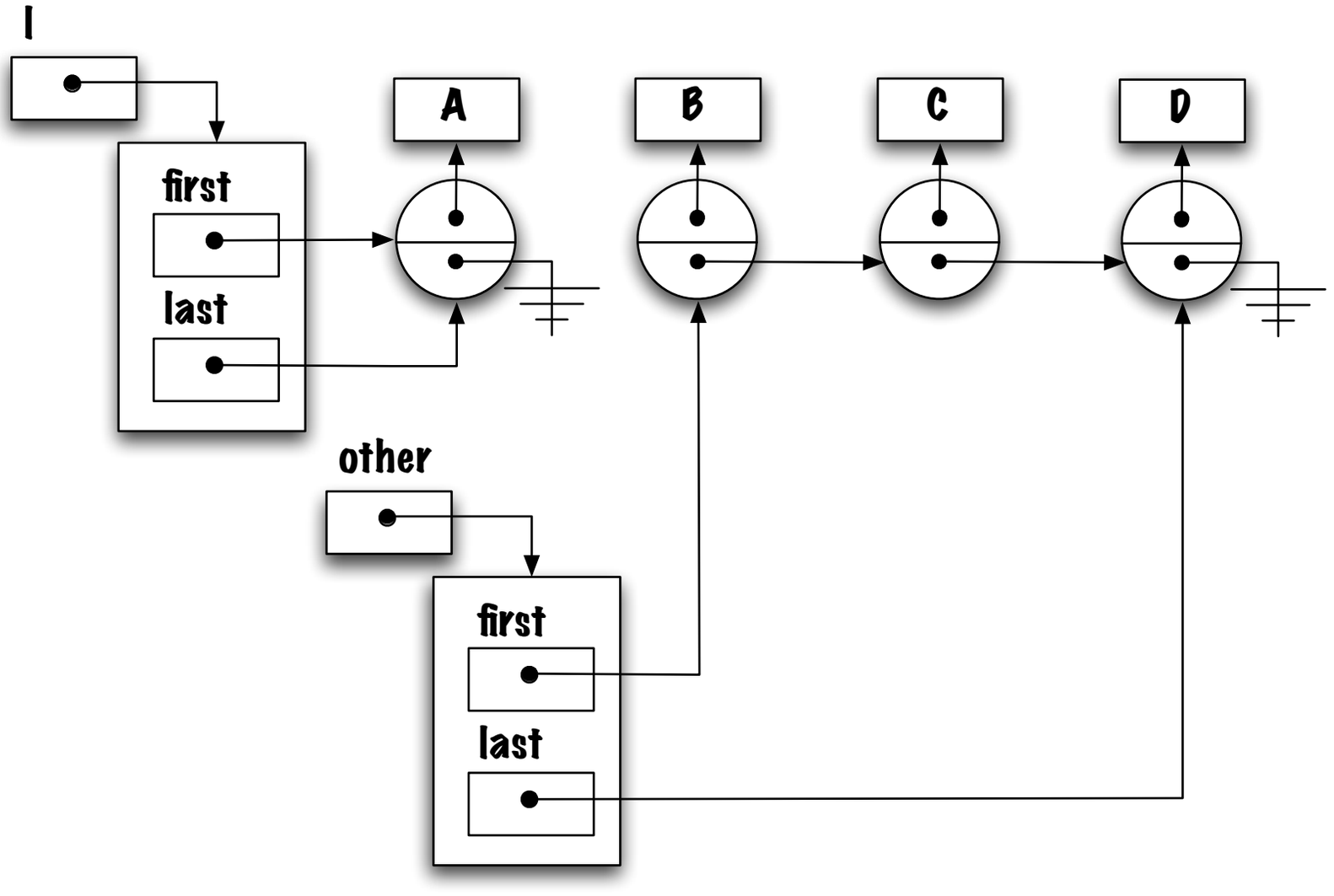
Est-ce que ça fonctionne ? Oui et non. Est-ce que le résultat est exact ? Oui. Y-a-t-il des problèmes ? Suite à un appel à la méthode **size**, le reste de la liste est perdu.

Discussion

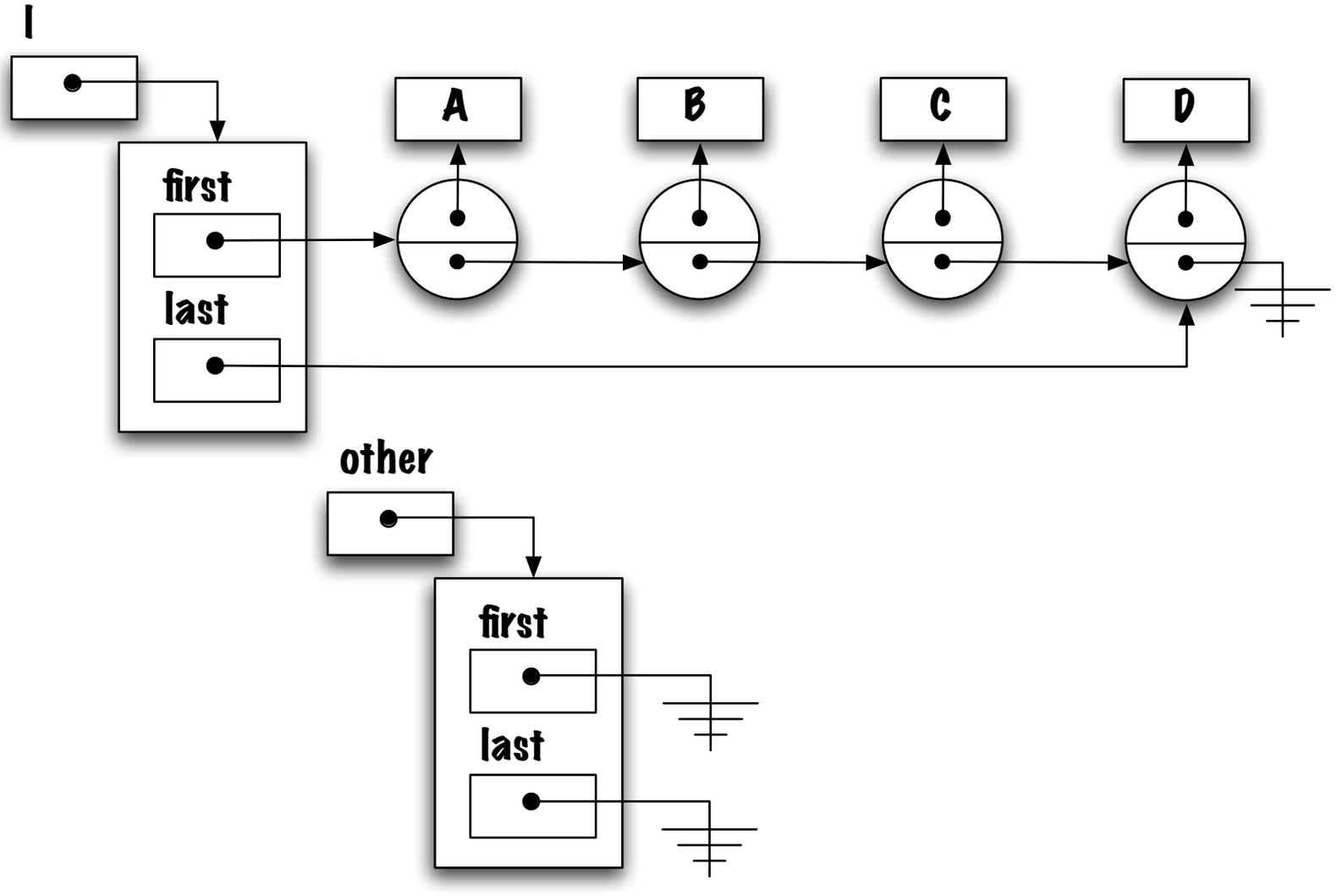
Suite à un appel à méthode **size**, tous les éléments restants ont été perdus. Que faire ?

Suite à l'appel récursif, il faut ≪rapiécer≫ le reste de la liste au premier élément.

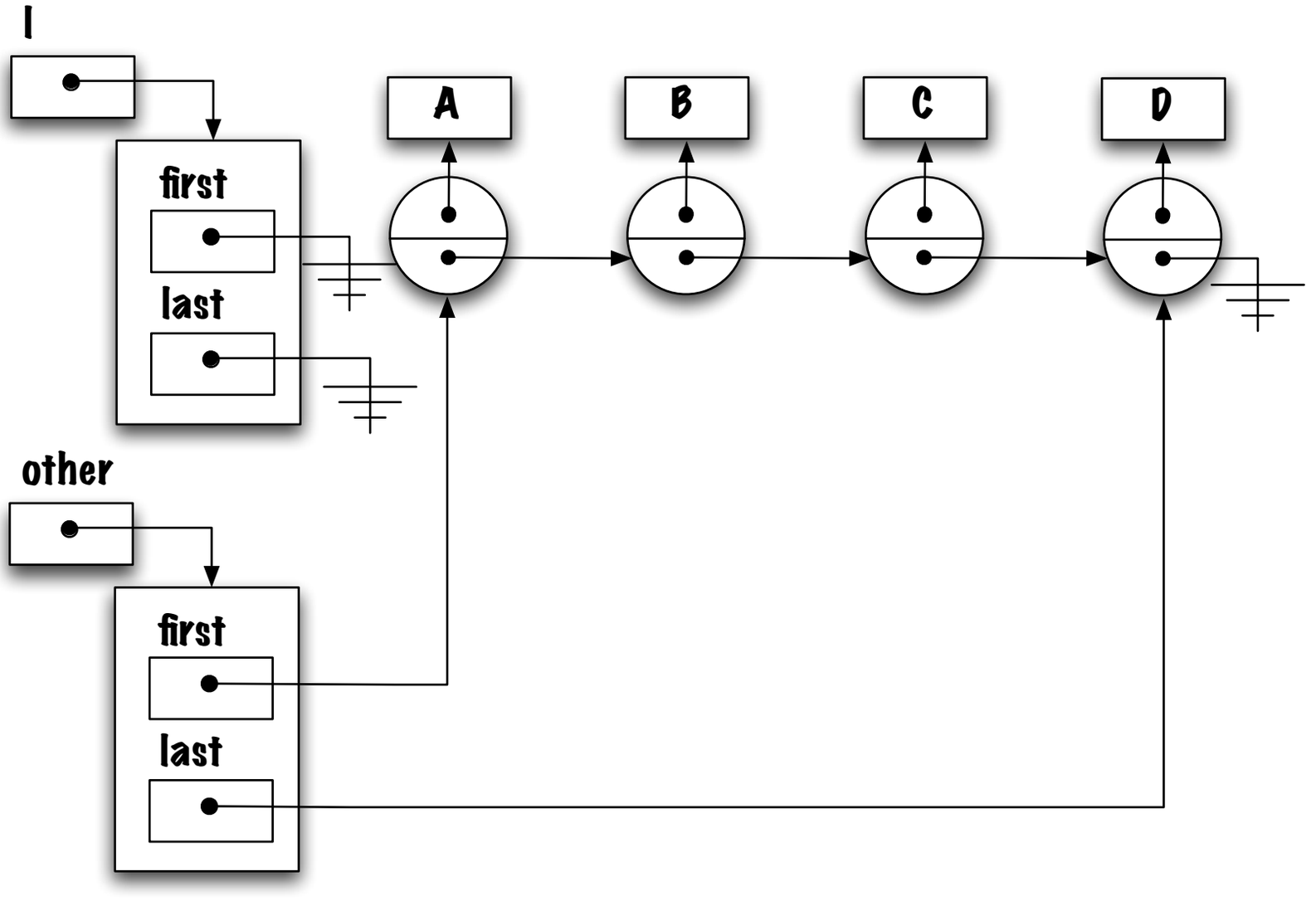
Étant donné deux instances **l** et **other**.



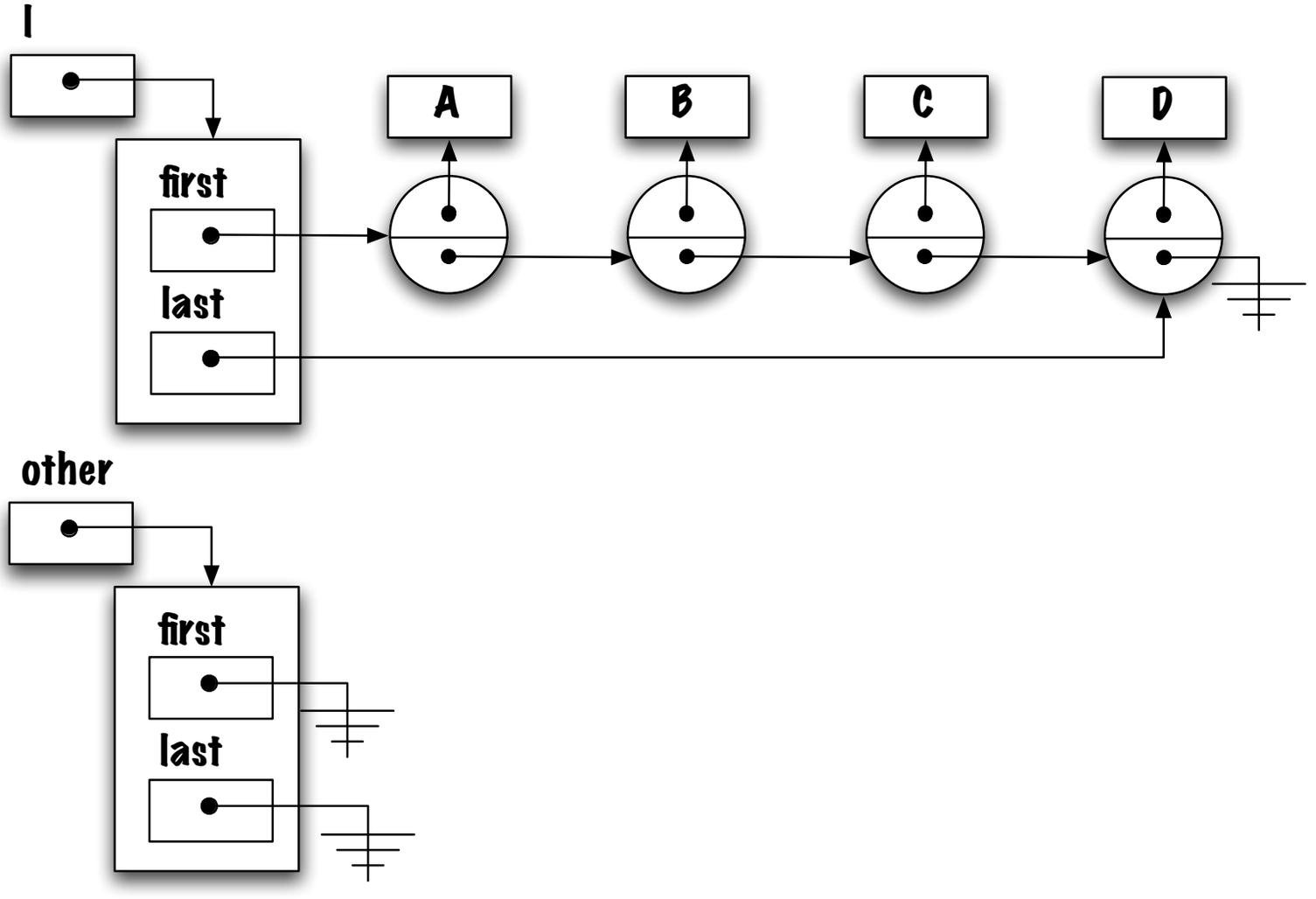
l.join(other) retire tous les éléments de **other** et les ajoute à cette liste.



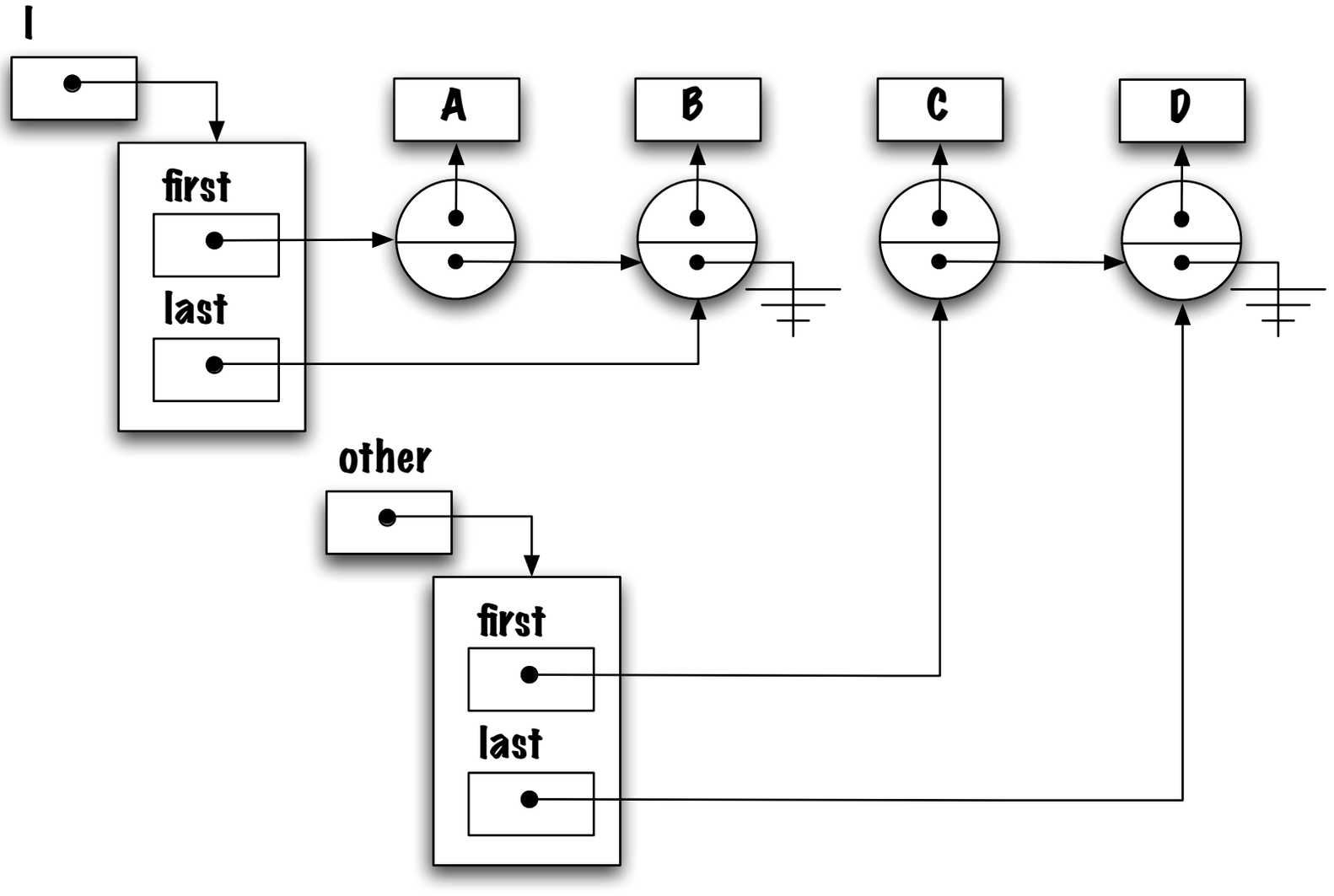
Étant donné deux instances **l** et **other**.



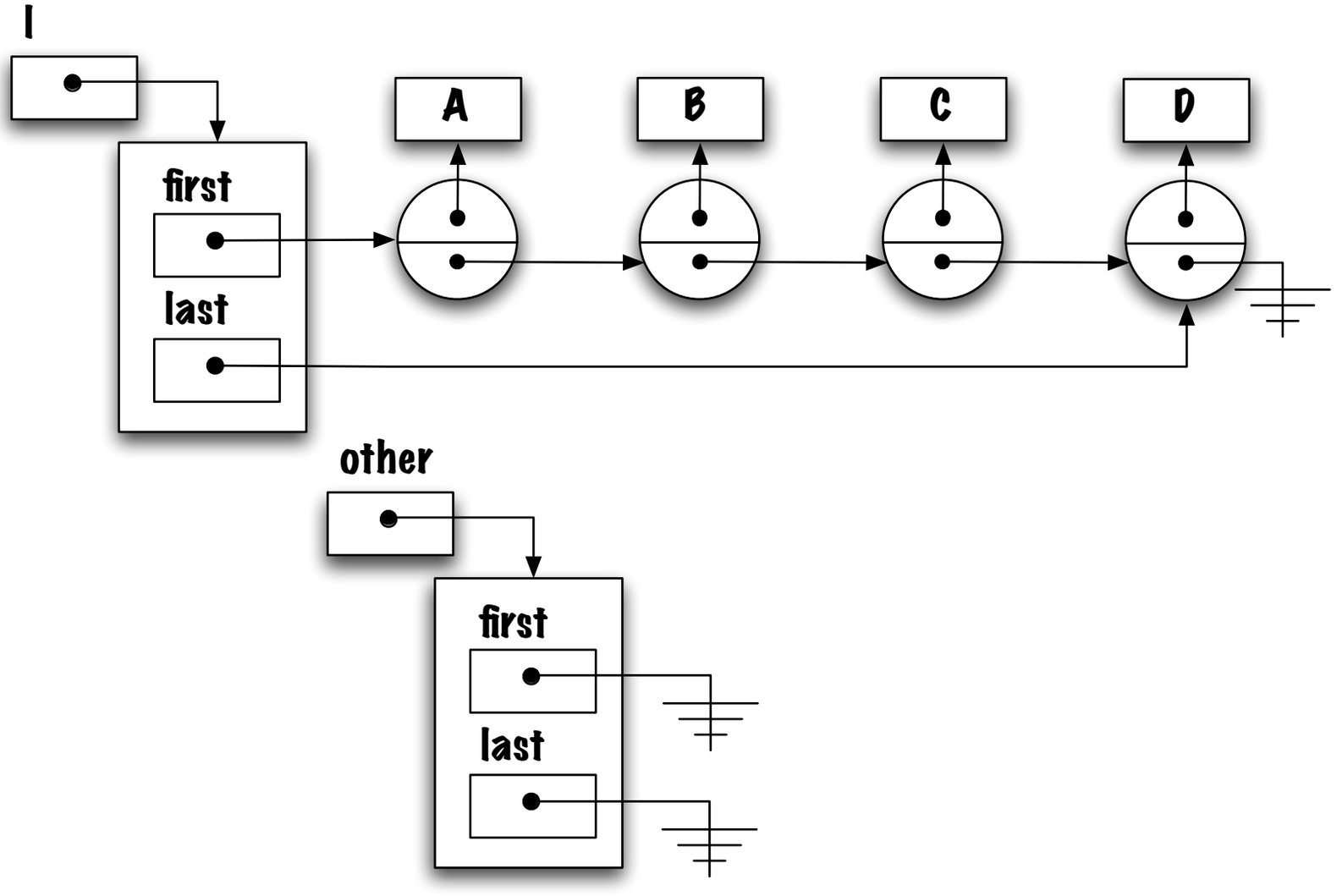
l.join(other) retire tous les éléments de **other** et les ajoute à cette liste.



Étant donné deux instances **l** et **other**.



l.join(other) retire tous les éléments de **other** et les ajoute à cette liste.



public static <E> int size(Sequence<E> l)

Apportez tous les changements nécessaires à la méthode **size** afin que la liste demeure inchangée suite à un appel à cette méthode.

```
public static <E> int size( Sequence<E> l ) {

    if ( l.isEmpty() ) {
        return 0;
    } else {
        return 1 + size( l.split() );
    }

}
```

public static <E> int size(Sequence<E> l)

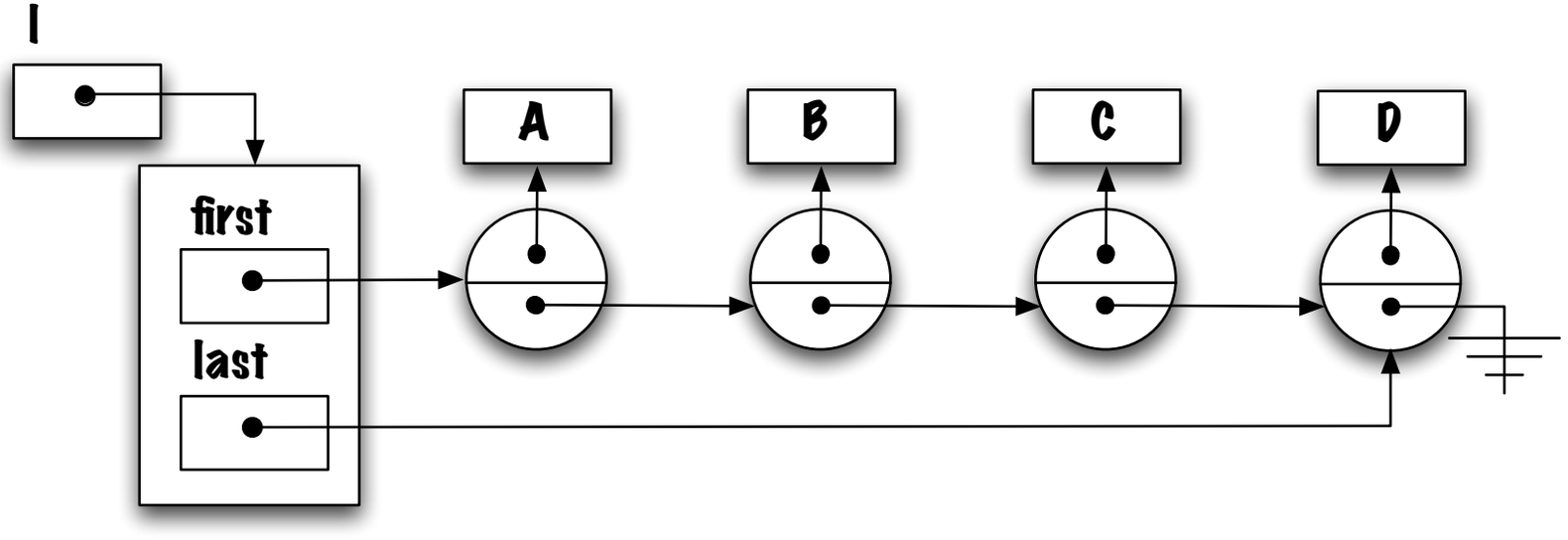
```
public static <E> int size( Sequence<E> l ) {

    int length;

    if ( l.isEmpty() ) {

        length = 0;

    } else {

        Sequence<E> t = l.split();
        length = 1 + size( t );
        l.join( t );
    }
    return length;
}
```

public static <E> int size(Sequence<E> l)

```
public static <E> int size( Sequence<E> l ) {

    if ( l.isEmpty() ) {

        return 0;

    } else {

        Sequence<E> t = l.split();
        int length = 1 + size( t );
        l.join( t );
        return length;

    }
}
```

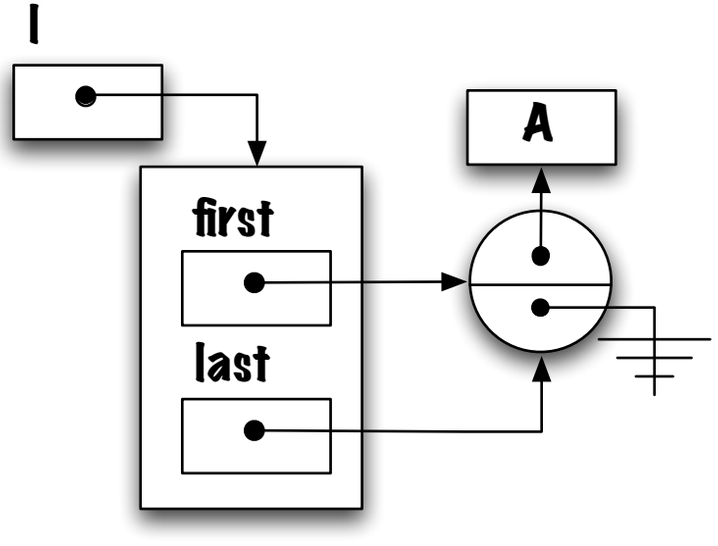
Sequence<E> split()

Implémentation du cas général.



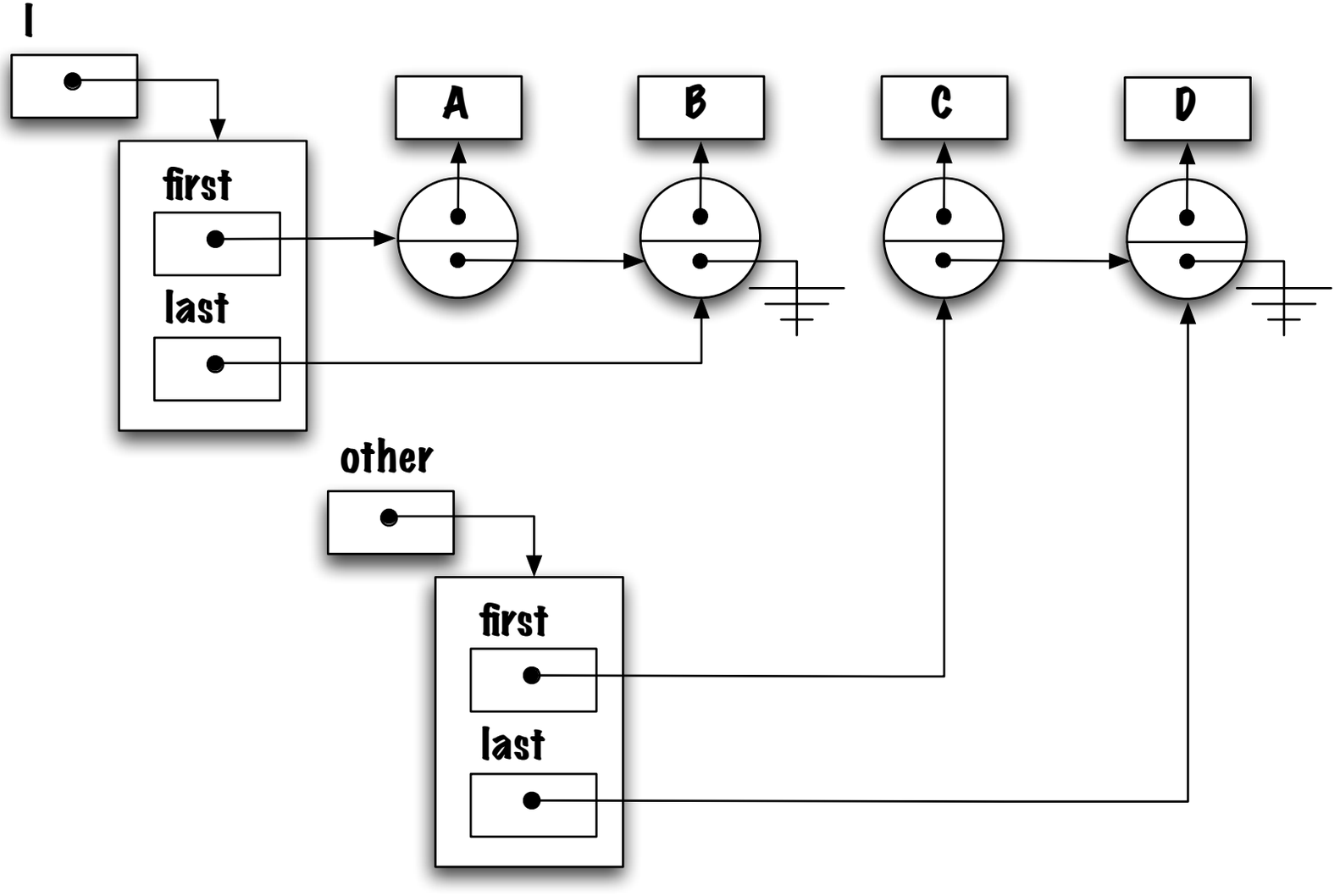
Sequence<E> split()

Implémentation du cas spécial.



```
public Sequence<E> split() {
    if ( isEmpty() ) {
        throw new NoSuchElementException();
    }

    Sequence<E> tail = new Sequence<E>();
    if ( first.next ! = null ) {
        tail.first = first.next;
        tail.last = last;
        first.next = null;
        last = first;
    }
    return tail;
}
```

Implémentez **join**.



```
public void join( Sequence<E> other ) {

    if ( ! other.isEmpty() ) {
        if ( isEmpty() ) {
            first = other.first;
            last = other.last;
        } else {
            last.next = other.first;
            last = other.last;
        }
        other.first = null; // other becomes empty
        other.last  = null;
    }
}
```

E head()

Pour faciliter l'implémentation de certaines solutions, ajoutons aussi la méthode **E head()**.

```
public E head() {
    if ( isEmpty() ) {
        throw new NoSuchElementException();
    }
    return first.value;
}
```

$\Rightarrow$ **E head()** est une méthode semblable à la méthode **Object peek()** des piles (**Stack**).

void reverse(Sequence<E> l)

Cas de base :

Liste vide. Que fait-on ? Rien.

Cas général :

Étant donné,

```
l -> (A) -> (B) -> (C) -> ... -> (Y) -> (Z)
```

Appliquons la stratégie “head+tail” à la lettre, i.e. on renverse le reste de la liste. Essayez à l’aide d’un crayon et d’un papier. Avons nous fait des progrès ?

void reverse(Sequence<E> l)

```
t = l.split();
```

Produira,

```
l -> (A)
t -> (B) -> (C) -> ... -> (Y) -> (Z)
```

reverse(t)

Produira,

```
t -> (Z) -> (Y) -> ... -> (C) -> (B)
```

Alors, progrès ? Oui, c'est presque la réponse finale.

void reverse(Sequence<E> l)

Qu'est-ce qui manque ? Ajouter l'élément **(A)** à la fin de la liste.

Comment ?

```
t.join( l );
```

Produira,

```
t -> (Z) -> (Y) -> ... -> (C) -> (B) -> (A)
```

void reverse(Sequence<E> l)

Que contient **l** ? **l** est vide.

Nous avons la solution sauf que la solution est dans la mauvaise liste ! **t** est une variable locale de la méthode **reverse**.

Comment transférer les éléments de **t** vers **l** ?

Que pensez-vous de ceci ?

```
l = t;
```

Ça ne fonctionne pas ? Pourquoi ?

Parce que **l** est un paramètre de la méthode **reverse**. Toute affectation **l = . . .** n'affecte que le paramètre (une référence locale) et sera sans effet pour l'appelant.

void reverse(Sequence<E> l)

Quelle est la solution ?

```
l.join( t );
```

Tous les éléments de **t** sont transférés dans **l** (**t** est maintenant vide).

void reverse(Sequence<E> l)

Solution finale.

```
public static <E> void reverse( Sequence<E> l ) {

   if ( l.isEmpty()) {
       return;
   }

   Sequence<E> t = l.split();
   reverse( t );
   t.join( l ); // ajoute l à la fin de t, l sera vide
   l.join( t ); // ajoute les éléments de t à l

}
```

void reverse(Sequence<E> l)

```
public static <E> void reverse( Sequence<E> l ) {

   if ( l.isEmpty()) {
       return;
   }
   Sequence<E> t = l.split();
   reverse( t );
   t.join( l );
   l.join( t );

}
```

Discussion

Quels sont les avantages et désavantages de cette approche ?

Exercices

À titre d'exercices, vous pouvez refaire tous les exemples des deux dernières semaines, soient **size, findMax, get, indexOf, isIncreasing, remove, subList**, etc.

subList : stratégie 1

La sous liste est créée dès le départ et passée en paramètre à la méthode **subListRec**.

```
static <E> Sequence<E> subList( Sequence<E> xs, int from, int to ) {
    Sequence<E> result;
    result = new Sequence<E>();
    subListRec( 0, result, xs, from, to );
    return result;
}
```

subList : stratégie 1

```
static <E> void subListRec( int current, Sequence<E> result, Sequence<E> xs, int from, int to ) {

    if ( current > to ) {
        return;
    }

    if ( current >= from ) {
        result.add( xs.head() );
    }

    Sequence<E> tail;
    tail = xs.split();

    subListRec( current+1, result, tail, from, to );

    xs.join( tail );

}
```

subList : stratégie 2

La sous-liste est créée lorsque la méthode **subListRec** atteint la limite supérieure de l'intervalle (tout le travail est fait dans la méthode **subListRect**).

```
static <E> Sequence<E> subList2( Sequence<E> xs, int from, int to ) {
    return subListRec2( 0, xs, from, to );
}
```

subList : stratégie 2

```
static <E> Sequence<E> subListRec2( int current, Sequence<E> xs, int from, int to ) {

    Sequence<E> result;

    if ( current == to ) {

        result = new Sequence<E>();
        result.add( xs.head() );

    } else {

        Sequence<E> tail;
        tail = xs.split();

        result = subListRec2( current+1, tail, from, to );

        if ( current >= from ) {
            result.addFirst( xs.head() );
        }

        xs.join( tail );
    }
    return result;
}
```