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 récursifs des listes (partie 2)

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

“Head + tail”

```
if ( ... ) { // Cas de base

   Calcul du résultat sans appel récursif

} else { // Cas général

  // prétraitement

  s = méthode( p.next ); // appel récursif

  // posttraitement

}
```

E get(int index)

Nous allons créer une implémentation récursive de la méthode **E get(int index)**. Quelle était la stratégie adoptée pour la méthode non récursive ? Il fallait compter le nombre de noeuds visités et terminer l'exécution de la boucle **while** après avoir visité **index** noeuds.

Pour une méthode récursive, comment détermine-t-on le nombre de noeuds visités ?

Tout d'abord, nous allons augmenter la signature de la méthode afin d'y ajouter un paramètre désignant l'élément courant (**p**) de l'appel récursif.

```
private E get( Node<E> p, int index );
```

Si **index** représente la position de l'élément par rapport à la liste débutant à la position **p**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **p.next** ? C'est bien ça, **index-1**.

Que fera la méthode si la valeur de l'index est 0 ? Elle doit retourner **p.value**. Sans appel récursif ? Non ! C'est donc le cas de base. En effet.

private E get(Node<E> p, int index)

```
private E get( Node<E> p, int index ) {

    if ( index == 0 ) {
       return p.value;
    }

    return get( p.next, index - 1 );

}
```

Que se passerait-il si la valeur initiale d'**index** était plus grande que le nombre total d'éléments de la liste ? **index** $>$ **0** et **p** $==$ **null**.

private E get(Node<E> p, int index)

```
private E get( Node<E> p, int index ) {

    if ( p == null ) {
       throw new IndexOutOfBoundsException();
    }

    if ( index == 0 ) {
       return p.value;
    }

    return get( p.next, index - 1 );
}
```

public E get(int index)

```
public E get( int index ) {
    if ( index < 0 ) {
       throw new IndexOutOfBoundsException();
    }
    return get( first, index );
}
private E get( Node<E> p, int index ) {
    if ( p == null ) {
       throw new IndexOutOfBoundsException();
    }
    if ( index == 0 ) {
       return p.value;
    }
    return get( p.next, index - 1 );
}
```

indexOf(E obj)

La méthode **indexOf** retourne la position de l'occurrence la plus à gauche de **obj** dans cette liste, et -1 si la valeur ne s'y trouve pas. La numérotation des éléments débute à zéro.

Selon la stratégie ≪head + tail≫, le cas général comportera un appel récursif tel que celui-ci :

```
int s = indexOf( p.next, obj );
```

Que représente la valeur de **s** ?

C'est la position d'**obj** dans la liste désignée par **p.next**.

Par rapport à la liste courante, celle désignée par **p**, quelle est la position d'**obj** ?
$1 + s$.

indexOf(Node<E> p, E obj)

Que signifie **s** == -1 ?

obj absent de la liste débutant à la position **p.next**.

Quel cas n'a pas encore été traité ? Ah oui, il se peut qu'**obj** se trouve à la position courante. Quelle est la valeur de l'index retourné ?

```
if ( p.value.equals( obj ) ) {
    result = 0;
} else if ( s == -1 ) {
    result = s;
} else {
    result = 1 + s;
}
```

indexOf(Node<E> p, E obj)

Qu'elle est le cas de base ?

La plus petite liste est la liste vide, elle ne contient pas l'élément recherché, il suffit de retourner la valeur spéciale -1.

```
if ( p == null ) {
    return -1;
}
```

int indexOf(Node<E> p, E obj)

```
private int indexOf( Node<E> p, E o ) {

    if ( p == null ) {
        return -1;
    }

    int result = indexOf( p.next, o );

    if ( p.value.equals( o ) ) {
        return 0;
    }

    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

private int indexOf(Node<E> p, E obj)

Est-ce que ça fonctionne ? Oui, mais cette solution n'est pas efficace. Pourquoi ?

private int indexOf(Node<E> p, E obj)

La récursivité devrait s'arrêter dès que l'élément recherché a été trouvé. Ouais, mais comment ?

```
private int indexOf( Node<E> p, E o ) {
    // Cas de base:
    if ( p == null ) {
        return -1;
    }
    if ( p.value.equals( o ) ) {
        return 0;
    }
    // Cas général
    int result = indexOf( p.next, o );
    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

p.value.equals(o) is a base case !

private int indexOf(Node<E> p, E obj)

Cette implémentation est très ≪déclarative≫.

```
private int indexOf( Node<E> p, E o ) {
    // Cas de base
    if ( p == null ) {
        return -1;
    }
    if ( p.value.equals( o ) ) {
        return 0;
    }
    // Cas général
    int result = indexOf( p.next, o );
    if ( result == -1 ) {
        return result;
    }
    return result + 1;
}
```

private int indexOfLast(Node<E> p, E obj)

La méthode **indexOfLast** retourne la position de la dernière occurrence (celle la plus à droite) de **obj**, et -1 sinon.

Quels sont les changements à apporter ?

Premièrement, est **p.value.equals(o)** peut faire partie du cas base ? Non, la récursivité doit parcourir la liste en entier.

Deuxièmement, comment allez vous combiner le résultat de **indexOfLast(p.next, obj)** et l'élément courant ?

int indexOfLast(Node<E> p, E obj)

```
public int indexOfLast( E obj ) {
    return indexOfLast( first, obj );
}
private int indexOfLast( Node<E> p, E obj ) {
    if (p == null) {
        return -1;
    }
    int result = indexOfLast( p.next, obj );
    if ( result > -1 ) {
        return result + 1;
    } else if ( obj.equals( p.value ) ) {
        return 0;
    } else {
        return -1;
    }
}
```

boolean contains(E o)

La méthode contains retourne **true** si la liste contient la valeur **o**, i.e. s'il y a un noeud tel que **p.value.equals(o)**.

La méthode auxiliaire,

```
public boolean contains( E o ) {
    return contains( first, o );
}
```

private boolean contains(Node<E> p, E o)

La signature de la méthode récursive sera,

```
private boolean contains( Node<E> p, E o );
```

Appliquons la stratégie ≪head+tail≫.

Cas de base ?

```
if ( p == null ) {
    return false;
}
```

Cas général ?

```
boolean result = contains( p.next, o );
```

contains(E o)

De même que la méthode **indexOf**, **contains** doit s'arrêter dès que la valeur a été trouvée.

```
private boolean contains( Node<E> p, E o ) {
    if ( p == null ) {
        return false;
    }
    if ( p.value.equals( o ) ) {
        return true;
    }
    return contains( p.next, o );
}
```

Scénarios plus complexes

Les méthodes **size**, **indexOf** et **contains** ne traitent qu'un élément à la fois.

Considérons une implémentation récursive de la méthode **isIncreasing**.

L'idée générale consiste regarder chaque paire consécutive et de retourner la valeur **false** dès qu'une paire n'est pas croissante.

Si la méthode atteint la fin de liste alors la liste est croissante !

boolean isIncreasing()

```
public boolean isIncreasing() {
    return isIncreasign( first );
}
```

boolean isIncreasing(Node<E> p)

Cas de base : quelle est la plus petite liste valide. La liste vide et le singleton sont croissante.

```
if ( ( p == null ) || ( p.next == null ) ) {
    return true;
}
```

boolean isIncreasing(Node<E> p)

Cas général : 1) devrait-on faire un appel récursive et traiter le résultat ou encore 2) traiter la position courante puis faire un appel récursif (si nécessaire) ?

```
if ( p.value.compareTo( p.next.value ) > 0) {
    return false;
} else {
    return isIncreasing( p.next );
}
```

boolean isIncreasing(Node<E> p)

```
private boolean isIncreasing( Node<E> p ) {

    if ( ( p == null ) || ( p.next == null ) ) {
        return true;
    }

    if ( p.value.compareTo( p.next.value ) > 0 ) {
        return false;
    }

    return isIncreasing( p.next );
}
```

Piège !

```
private boolean isIncreasing( Node<E> p ) {

    if ( ( p == null ) || ( p.next == null ) ) {
        return true;
    }

    if ( p.value.compareTo( p.next.value ) > 0 ) {
        return false;
    }

    return isIncreasing( p.next.next );
}
```

Exercices

Pour une liste simplement chaînée, implémentez les méthodes suivantes.

void addLast(E o) ; ajoute un élément à la fin de la liste.

boolean equals(OrderedList<E> other) ; compare deux listes et retourne **true** si les deux listes sont équivalentes, de même longueur et contiennent les mêmes éléments dans le même ordre.

Modifier la structure de la liste

Nous considérons maintenant des méthodes qui transforment la structure de la liste.

Pour les méthodes **indexOf** et **contains**, la conséquence principale des appels récursifs supplémentaires est l'inefficacité de la méthode.

Par contre, les méthodes récursives qui transforment les listes peuvent engendrer des problèmes plus sérieux.

Considérez l'exemple de la méthode **remove**, qui retire la première occurrence d'un objet dans la liste.

public void remove(E o)

void remove(E o) ; retire la première occurrence (celle la plus à gauche) de l'objet **o**.

Organisation générale de la méthode ?

– Traverser la liste ;
– Une fois l'élément trouvé, on le retire.

Difficultés (2) ?

On se rappellera que lors du parcours d'une liste simplement chaînée à l'aide d'une boucle **while**, nous devions nous arrêter une position avant l'élément à retirer, puisque c'est la variable **next** de l'élément qui précède qu'il faut changer.

Il y a aussi une seconde difficulté, en fait un cas spécial, quel est-il ? Pour retirer le premier élément il faut modifier la variable **first** de l'entête, et la variable **next** du noeud qui précède.

public void remove(E o)

Considérons la méthode ≪**public**≫ **non récursive** d'abord.

Quelles sont les préconditions ? La liste n'est pas vide.

Lorsqu'une méthode peut changer la structure de la liste, la méthode ≪**public**≫ **non récursive** a souvent un cas spécial : le retrait du premier élément.

```
public void remove( E o ) {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    if ( first.value.equals( o ) ) {
        first = first.next;
    } else {
        remove( first, o );
    }
}
```

Exercice : ≪scrubbing the memory≫.

private void remove(Node<E> p, E o)

Remarques : pour le premier appel à la méthode **remove(Node<E> p, E o)**, nous savons que **p.value.equals(o)** est faux. En effet, **p == first** et le cas **first.value.equals(o)** a été traiter la méthode non-récursive.

La méthode récursive préservera cette propriété, elle vérifie si l'élément recherché, **o**, se trouve à la position qui suit, **p.next**, et si oui retire ce noeud et termine, sinon elle poursuit sa recherche.

private void remove(Node<E> p, E o)

Cas général : quel scénario semble le mieux approprié : **1)** faire un appel récursif suivi d'un posttraitement ou **2)** faire un prétraitement suivi d'un appel récursif (si nécessaire) ?

Puisque nous devons retirer l'élément le plus à gauche, la stratégie 2) est celle qu'il faut suivre.

Quel est le prétraitement nécessaire ? Si **o** se trouve à la position suivante on retire le noeud suivant et termine l'exécution.

Quel est le cas de base ? Le singleton (**p.next == null**). Que fait-on ? Lance l'exception **NoSuchElementException**.

private void remove(Node<E> p, E o)

```
private void remove( Node<E> p, E o ) {

    if ( p.next == null ) {
        throw new NoSuchElementException();
    }

    // General case
    if ( p.next.value.equals( o ) ) {
        p.next = p.next.next;
    } else {
        remove( p.next, o );
    }
}
```

public void remove(E o)

```
public void remove( E o ) {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    if ( first.value.equals( o ) ) {
        first = first.next;
    } else {
        remove( first, o );
    }
}
private void remove( E o, Node<E> p ) {
    if ( p.next == null ) {
        throw new NoSuchElementException();
    }
    if ( p.next.value.equals( o ) ) {
        p.next = p.next.next;
    } else {
       remove( p.next, o );
    }
}
```

Exercices

void add(E c) ; ajout d'un élément tout en préservant l'ordre des éléments ;

void removeLast() ; retrait du dernier élément ;

void removeLast(E o) ; retire l'occurrence la plus à droite de **o** (c'est peut-être plus difficile que ça ne le semble).

void removeAll(E o) ; retire toutes les occurrences de **o**.

void remove(int pos) ; retire l'élément à la position **pos**.

LinkedList<E> subList(int fromIndex, int toIndex)

Voici des méthodes retournant un ensemble de valeurs sous forme de liste.

En particulier, la méthode **LinkedList<E> subList(int fromIndex, int toIndex)** retournera un nouvelle liste contenant les éléments situés aux positions **fromIndex** jusqu'à **toIndex** de la liste originale, sans la changée.

La difficulté principale réside dans l'élaboration de notre stratégie afin de construire le résultat. Suggestions ?

Je vous propose deux stratégies, pour chacune il faut connaître l'index de la position courante dans la liste. Suggestions ?

Stratégie 1 : traverser la liste jusqu'à l'index le plus élevé et retourner une liste contenant seulement la valeur se trouvant à cette position, en suite, sur le retour des appels récursifs, ajouter l'élément courant au début de la liste, si sa position fait partie de l'intervalle.

Stratégie 2 : une liste vide est passée en paramètre, tout en traversant la liste ajouter l'élément courant à la fin de la liste des résultats si sa position fait partie de l'intervalle.

Stratégie 1

Les appels récursifs traversent la liste de gauche à droite, la récursivité s'arrête lorsque l'index **toIndex** est atteint.

Cas de base :

```
LinkedList<E> result;

if ( index == toIndex ) {
   result = new LinkedList<E>();
   result.addFirst( p.value );
}
```

Stratégie 1

Cas général :

```
result = subList( p.next, index + 1, fromIndex, toIndex );
```

Que contient **result** ? Quelle est la prochaine étape ?

```
if ( index > fromIndex ) {
   result.addFirst( p.value );
}
```

Stratégie 1

```
private LinkedList<E> subList( Node<E> p, int index, int fromIndex, int toIndex ) {
    LinkedList<E> result;

    if ( index == toIndex ) {
        result = new LinkedList<E>();
        result.addFirst( p.value );
    } else {
        result = subList(  p.next, index + 1, fromIndex, toIndex );
        if ( index >= fromIndex ) {
           result.addFirst( p.value );
        }
    }
    return result;
```

Bien que plus complexe, cette solution suit le modèle proposé (“head+tail”).

Stratégie 1

```
public LinkedList<E> subList( int fromIndex, int toIndex ) {
    return subList( first, 0, fromIndex, toIndex );
}
```

Le traitement des préconditions (intervalle de valeurs illégal) est laissé comme exercice.

Stratégie 2

Pour la seconde stratégie, la liste des résultats est créée dès le départ et les éléments y sont insérés tout en traversant la liste.

```
public LinkedList<E> subList( int fromIndex, int toIndex ) {
    LinkedList result = new LinkedList<E>();
    subList( first, 0, result, fromIndex, toIndex );
    return result;
}
```

Stratégie 2

Cas de base :

```
if ( index == toIndex ) {
   result.addLast( p.value );
}
```

result.addLast(p.value) or **result.addFirst(p.value)** ?

Stratégie 2

Cas général :

```
if ( index >= fromIndex ) {
   result.addLast( p.value );
}
subList( p.next, index + 1, result, fromIndex, toIndex );
```

Stratégie 2

```
private void subList( Node<E> p, int i, LinkedList r, int f, int t ) {

    if ( i == t ) {
        r.addLast( p.value );
    } else {
        if ( i >= f ) {
           r.addLast( p.value );
        }
        subList( p.next, i + 1, r, f, t );
    }
```

ou **f = fromIndex**, **t = toIndex**, **i = index**, **r = result**.

LinkedList filterLessThan(E c)

LinkedList<E> filterLessThan(E c) retourne une liste (**LinkedList<E>**) contenant tous les éléments plus petits que **c**.

Stratégie ?

(Cas de base :) À quel moment doit-on s'arrêter ? À la fin de liste, tous les éléments doivent être visités.

(Cas général :) Que contient **result** ?

```
result = filterLessThan( p.next, c );
```

Tous les éléments qui sont plus petits que **c** dans la liste débutant à la position **p.next**.

Prochaine étape ? Ajouter l'élément courant à la liste des résultats s'il est plus petit que **c**.

LinkedList<E> filterLessThan(Node<E> p, LinkedList<E> result, E c)

```
private void filterLessThan( Node<E> p, LinkedList<E> result, E c ) {
  if ( p == null ) {
      return;
  }
  filterLessThan( p.next, result, c );
  if ( p.value.compareTo( c ) < 0 )
      result.addFirst( p.value );
}
```

LinkedList<E> filterLessThan(E c)

```
public LinkedList<E> filterLessThan( E c ) {
  LinkedList<E> result = new LinkedList<E>();
  filterLessThan( first, result, c );
  return result;
}
private void filterLessThan( Node<E> p, LinkedList<E> result, E c ) {
  if ( p == null ) {
      return;
  }
  filterLessThan( p.next, result, c );
  if ( p.value.compareTo( c ) < 0 ) {
      result.addFirst( p.value );
  }
}
```

LinkedList<E> filterLessThan(E c) (prise 2)

```
public LinkedList filterLessThan( E c ) {
  return filterLessThan( first, c );
}
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

LinkedList<E> filter(Predicate<E> f)

```
private LinkedList<E> filterLessThan( Node<E> p, E c ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filterLessThan( p.next, c );
      if ( p.value.compareTo( c ) < 0 ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

Qu'est-ce qu'il faudrait changer afin d'implémenter **filterGreaterThan** ? Pour **filterGenderMale** ? Pour `...` ?

Predicate

```
public interface Predicate<E> {
    public abstract boolean evaluate( E arg );
}
```

IsNegative

```
public class IsNegative implements Predicate<Integer> {

    public boolean evaluate( Integer arg ) {
        return args.intValue() < 0;
    }
}
```

LinkedList<E> filter(Predicate<E> f)

```
public LinkedList<E> filter( Predicate<E> f ) {
  return filter( first, f );
}
private LinkedList<E> filter( Node<E> p, Predicate<E> f ) {
  LinkedList<E> result;
  if ( p == null ) {
      result = new LinkedList<E>();
  } else {
      result = filter( p.next, f );
      if ( f.evaluate( p.value ) ) {
          result.addFirst( p.value );
      }
  }
  return result;
}
```

IsNegative

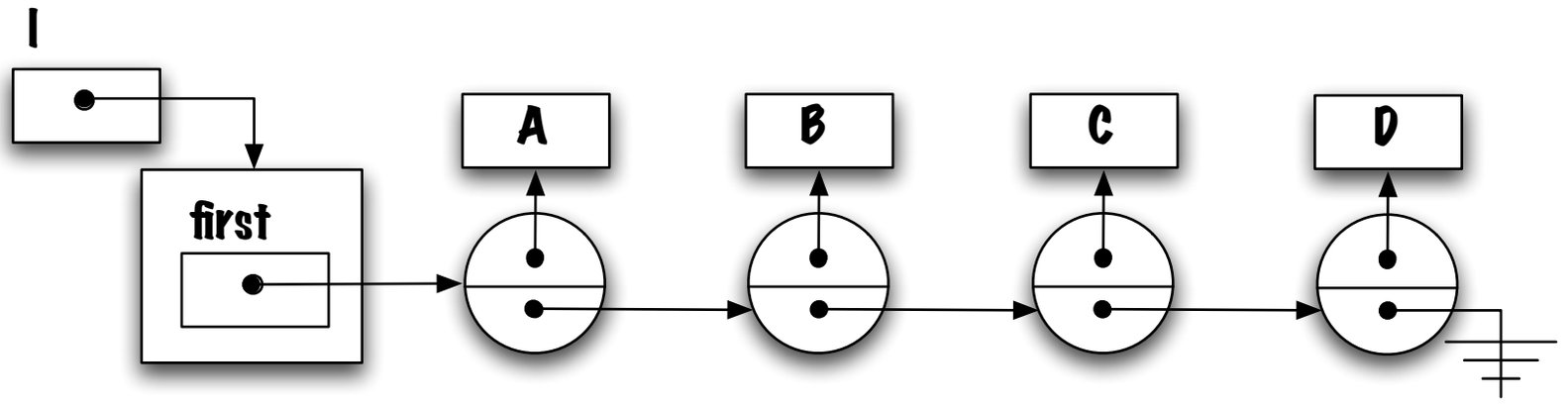
```
l2 = l1.filter( new IsNegative() );
```

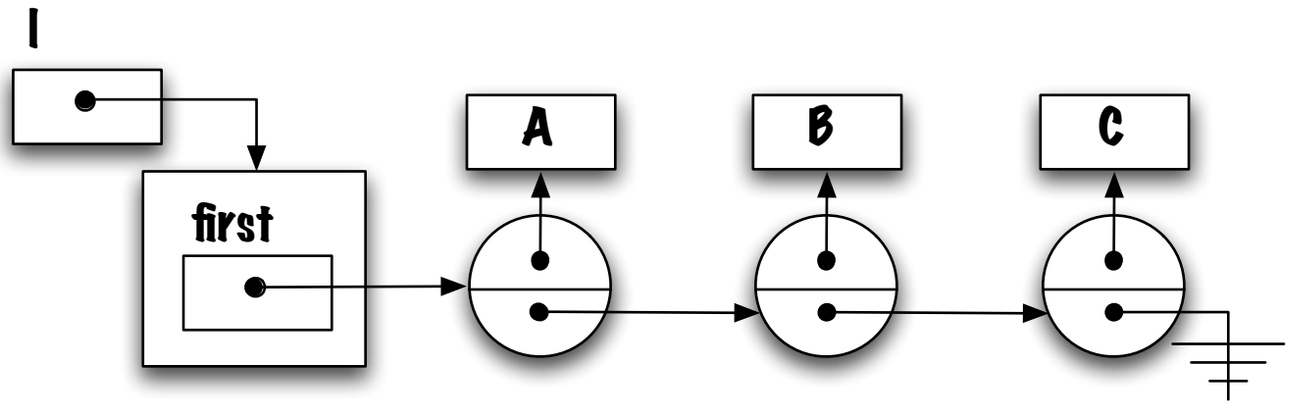
IsPositive (anonyme)

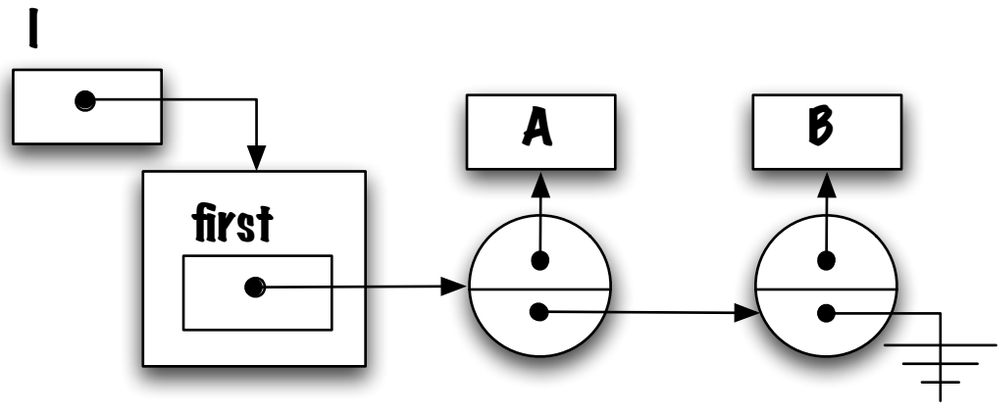
```
l2 = l1.filter( new Predicate<E>() {
    public boolean evaluate( Integer arg ) {
        return args.intValue() > 0;
    }
});
```

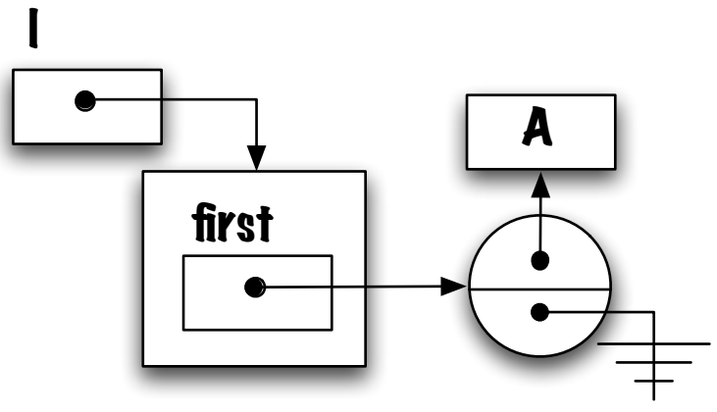
“Head + tail”

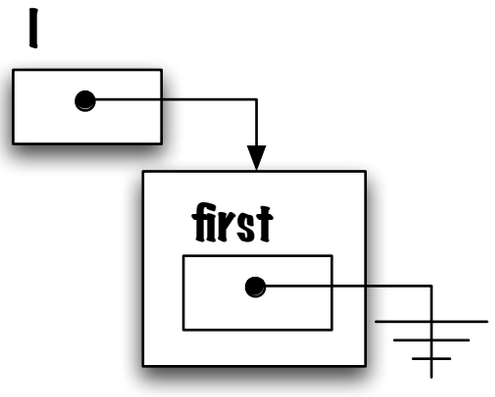
```
if ( ... ) { // Cas de base

   Calcul du résultat sans appel récursif

} else { // Cas général

  // prétraitement

  s = method( p.next ); // appel récursif

  // posttraitement

}
```


l
first
A
B
C
D


l
first
A
B
C


l
first
A
B


l
A
first


l
first