ITI 1521. Introduction à l'informatique II*

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

Version du 14 mars 2011

Résumé

– Type abstrait de données (TAD) :
  – Liste

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

Definitions

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 (est 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.

Les opérations de base sont :

int size() : retourne le nombre d'éléments sauvegardés ; la liste vide a une taille 0 ;

int get(int index) : l'accès aux éléments se fait par position (ou par contenu). Quel sera l'index du premier élément de la liste, 0 ou 1 ? Comme pour les tableaux, le premier élément se trouve à la position 0 ;

void add(int index, E elem) : ajout d'un élément à une position quelconque ;

void remove(int index) : retrait d'un élément par position (ou contenu).

⇒ Les listes sont donc plus générales que les piles et les files. Piles et files peuvent être implémentées à l'aide d'une liste.

```
public interface List<E> {

    public abstract void add( int index, E elem );
    public abstract boolean add( E elem );
    public abstract E remove( int index );
    public abstract boolean remove( E o );
    public abstract E get( int index );
    public abstract E set( int index, E element );
    public abstract int indexOf( E o );
    public abstract int lastIndexOf( E o );
    public abstract boolean contains( E o );
    public abstract int size();
    public abstract boolean isEmpty();
}
```

⇒ L'interface ci-haut déclare un sous-ensemble des méthodes de l'interface **java.util.List**.

Implémentations

– ArrayList ;
– LinkedList :
  – Listes simplement chaînées ;
  – Listes doublement chaînées ;
  – Noeud factice (dummy node) ;
  – Traitement itératif (Iterator) ;
  – Traitement récursif.

De nouveaux concepts seront introduits au besoin afin d'améliorer l'efficacité des implémentations.

Efficacité par rapport au **temps d'exécution** et/ou **consommation de mémoire** ; nous nous intéresserons surtout à la vitesse d'exécution.

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<E> {
    private E value;
    private Node<E> next;
    private Node( E value, Node<E> next ) {
        this.value = value;
        this.next = next;
    }
}
```

La classe **SinglyLinkedList** a une variable d'instance qui désigne le premier élément de la liste, que nous nommerons **head**.

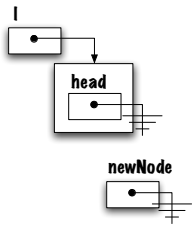
⇒ La classe imbriquée est parfois nommée **Elem** ou **Entry**.

addFirst(E o) (1/2)

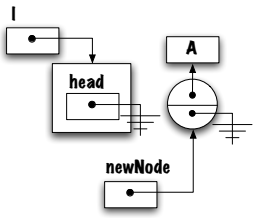
L'insertion d'un élément en tête de liste nécessite 1) la création d'un nouveau noeud, ainsi que 2) l'ajout de l'élément à la liste.

```
public void addFirst( E elem ) {

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

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

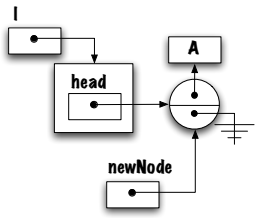
Cas spécial : addFirst(E o) (1/2)



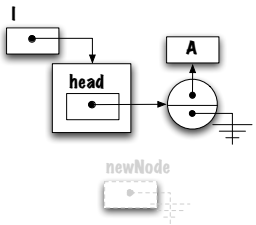
Cas spécial : addFirst(E o) (1/2)



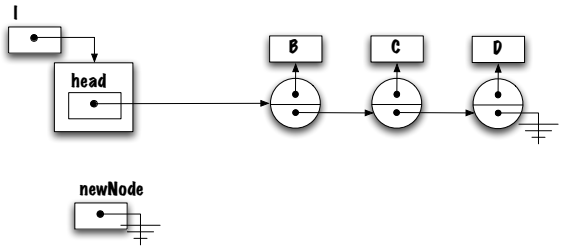
Cas spécial : addFirst(E o) (1/2)



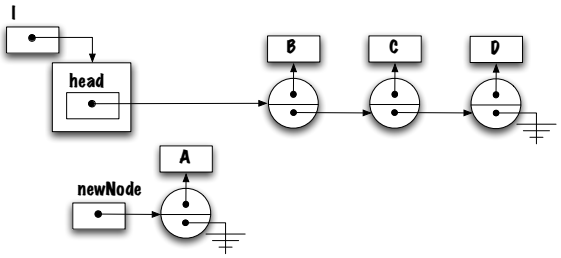
Cas spécial : addFirst(E o) (1/2)



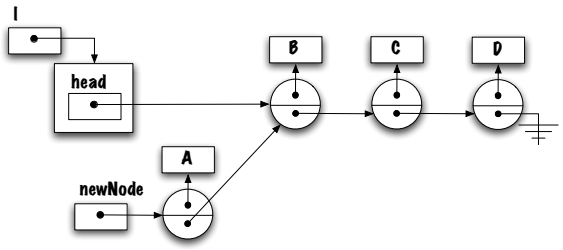
Cas général : addFirst(E o) (1/2)



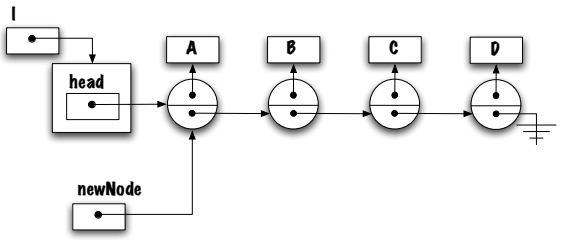
Cas général : addFirst(E o) (1/2)



Cas général : addFirst(E o) (1/2)



Cas général : addFirst(E o) (1/2)



Est-ce que cette distinction entre le cas de la liste vide et le cas de la liste ayant des éléments est vraiment nécessaire ?

Que pensez-vous de cette implémentation ?

```
public void addFirst( E elem ) {
    head = new Node<E>( elem, head );
}
```

Ça fonctionne pour les deux cas (spécial et général) ?

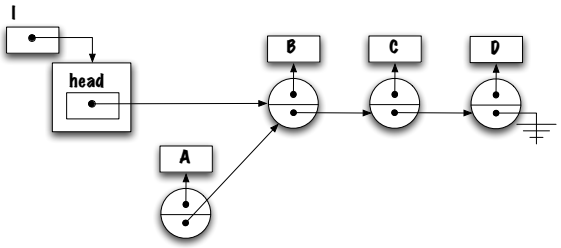
Oui, ça fonctionne. Pourquoi ?

Parce que Java évalue d'abord le côté droit de l'expression.

addFirst(E o) (2/2)

Évaluation du côté droit.

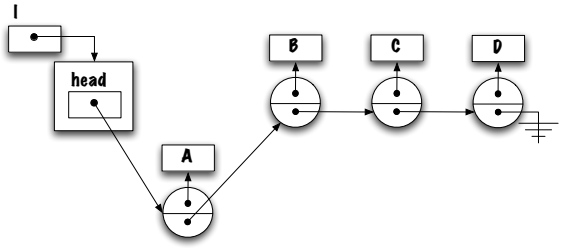
```
head = new Node<E>( elem, head );
```



addFirst(E o) (2/2)

Le résultat (une référence vers l'élément nouvellement créé) est affecté à la variable **head**.

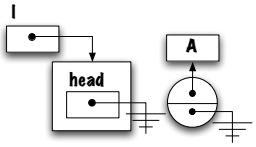
```
head = new Node( o, head );
```



addFirst(E elem) (2/2)

De même, le résultat de l'évaluation du côté droit, ici **head** est **null**, est affecté à la variable d'instance **next** du noeud.

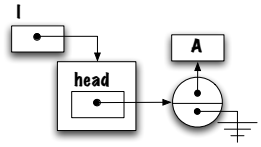
```
head = new Node<E>( elem, head );
```



addFirst(E elem) (2/2)

Le résultat (une référence vers l'élément nouvellement créé) est affecté à la variable **head**.

```
head = new Node<E>( elem, head );
```



add(E elem) (1/3)

Pour ajouter un élément à la fin de la liste, il faut traverser la liste et modifier la variable **next** du dernier noeud de la liste.

```
public void add( Elem elem ) {

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

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

⇒ Quelque chose ne va pas, mais quoi ?

add(E elem) (2/3)

Après l'exécution de la boucle, la valeur de **p** est **null**, une exception de type **NullPointerException** sera lancée lors de l'exécution de **p.next = newNode**.

```
public void add( E elem ) {
    Node<E> newNode = new Node<E>( elem, null );

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

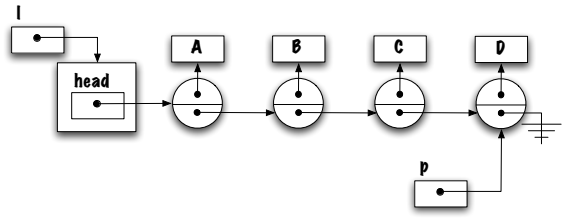
add(E elem) (1/3) (prise 2)

Après l'exécution de la boucle, la valeur de **p** est **null**, une exception de type **NullPointerException** sera lancée lors de l'exécution de **p.next = newNode**.

```
public void add( E elem ) {

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

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p = newNode;
}
```

⇒ Qu'en pensez-vous ?

add(E elem) (2/3)



Comment pourrait-on s'arrêter sur le dernier noeud ?

Il faut s'arrêter lorsque **p.next == null** est **true**.

add(E elem) (2/3)

```
public void add( E elem ) {
    Node<E> newNode = new Node<E>( elem, null );

    Node<E> p = head;
    while ( p.next != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

⇒ Il y a encore un problème, quel est-il ?

add(E elem) (2/3)

Qu'arrivera-t-il si la liste est vide ?

```
public void add( E elem ) {
    Node<E> newNode = new Node<E>( elem, null );

    Node<E> p = head;
    while ( p.next != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

1) Qu'arrivera-t-il au moment de l'exécution de **p.next != null** ? 2) Qu'est-ce qui change lorsqu'on insère un élément dans une liste vide ?

add(E elem) (3/3)

```
public void add( E elem ) {
    Node<E> newNode = new Node<E>( elem, null );

    if ( head == null ) {
        head = newNode;
    } else {
        Node<E> p = head;
        while ( p.next != null )
            p = p.next;
        p.next = newNode;
    }
}
```

Le traitement de la liste vide est bien souvent particulier : il faut évider tout accès de la forme **head.value** ou **head.next** (**head** ne désigne aucun objet), il faut aussi modifier la variable **head** et non la variable **next** d'un des éléments de la liste.

removeFirst()

```
public E removeFirst() {

    // pre-condition?

    Node<E> first = head;

    E savedValue = first.value;

    head = head.next;

    first.value = null;
    first.next = null;

    return savedValue;
}
```

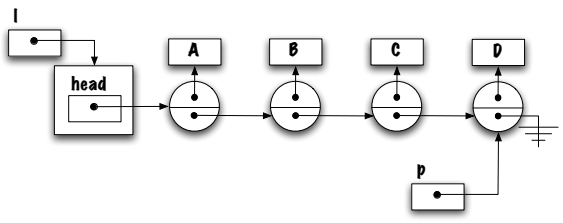
removeLast() (1/4)

Il nous faut traverser la liste, mais nous savons qu'il faut faire attention afin de s'arrêter au bon moment !

```
Node<E> p = head;
while (p.next != null) {
    p = p.next;
}
```

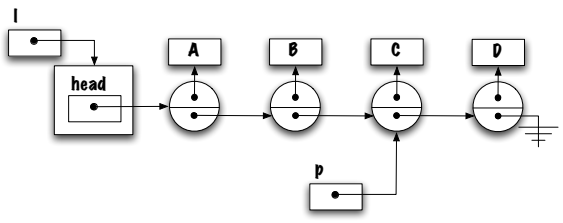
⇒ Qu'en pensez-vous ?

removeLast() (2/4)



La variable locale **p** désigne le dernier élément de la liste, comment accède-t-on à la variable **next** de son prédécesseur ?

removeLast() (2/4)



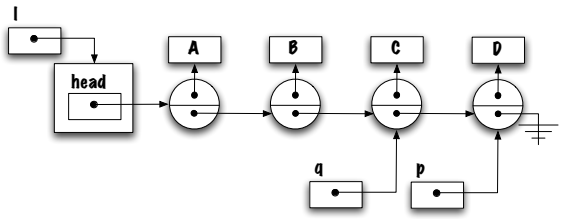
L'itération se termine lorsque **p** désigne l'avant dernier élément de la liste.

removeLast() (2/4)

Nous pourrions utiliser une seconde boucle qui s'arrêterait lorsque la variable **next** de l'élément courant désigne le même objet que **p**.

```
Node<E> p = head;
while ( p.next != null ) {
    p = p.next;
}


Node q = head;
while ( q.next != p ) {
    q = q.next;
}
```

removeLast() (2/4)



⇒ Qu'en pensez-vous ?

removeLast() (3/4)

Nous favoriserons une approche avec une seule boucle, s'arrêtant au bon moment, pour des raisons d'efficacité. Comment ?

```
Node<E> p = head;
while ( p.next.next != null ) {
    p = p.next;
}
Node<E> last;
last = p.next;
p.next = null;

result = last.next.value;
last.value = null;
```

⇒ Y-a-t-il des cas spéciaux ?

removeLast() (4/4)

```
public E removeLast() {
    // pre-condition?
    E result;
    Node<E> last;
    if ( head.next == null ) {
        last = head;
        head = null;
    } else {
        Node<E> p = head;
        while ( p.next.next != null ) {
            p = p.next;
        }
        last = p.next;
        p.next = null;
    }
    result = last.value;
    last.value = null;
    return result;
}
```

remove(E elem)

Accès aux éléments par contenu.

Retourne **true** is **elem** a été retiré et **false** sinon.

Quels sont les éléments de cette méthode ?

1. Traverser la liste ;
2. Critère d'arrêt ?
3. Retrait.

remove(E elem)

Qu'en pensez-vous ?

```
public boolean remove( E elem ) {

        Node<E> p = head, toDelete;

        while ( p != null && ! p.value.equals( elem ) ) {
            p = p.next;
        }
        toDelete = p;

        ...

    return true;
}
```

```
public boolean remove( E elem ) {

    Node<E> toDelete = null;
    Node<E> p = head;

    while ( p.next != null && ! p.next.value.equals( elem ) ) {
      p = p.next;
    }

    toDelete = p.next;
    p.next = toDelete.next;
    toDelete.value = null;
    toDelete.next = null;

    return true;
}
```

Problèmes ? Qu'arrivera-t-il si l'élément est absent de liste ?

Que fait-on si la liste est vide ?

```
public boolean remove( E elem ) {
    if ( head == null ) {
        return false;
    }
    Node<E> toDelete = null;
    if ( head.value.equals( elem ) ) {
        toDelete = head;
        head = head.next;
    } else {
        Node<E> p = head;
        while ( p.next != null && ! p.next.value.equals( elem ) ) {
            p = p.next;
        }
        if ( p.next == null ) {
            return false;
        }
        toDelete = p.next;
        p.next = toDelete.next;
    }
    toDelete.value = toDelete.next = null;
    return true;
```

```
public boolean remove( E elem ) {
    Node<E> toDelete = null;
    Node<E> p = head;

    while ( p.next != null && ! p.next.value.equals( elem ) ) {
        p = p.next;
    }
    if ( p.next == null ) {
        return false;
    }
    toDelete = p.next;
    p.next = toDelete.next;
    toDelete.value = null;
    toDelete.next = null;
    return true;
}
```

```
}
```

get(int pos)

Accès aux éléments par position.

Pour être conformes à l'implémentation à base de tableaux, nous devons désigner le premier élément par la position 0. Ainsi, du point de vue de l'utilisateur de cette classe, il importe peu que l'implémentation utilise un tableau ou une liste chaînée.

Il nous faudra traverser la liste !

Il faudra aussi s'arrêter tôt.

Comment déterminer l'endroit ou s'arrêter ?

Il suffit de compter le nombre d'éléments visités.

Il nous faut donc un compteur.

get(int pos)

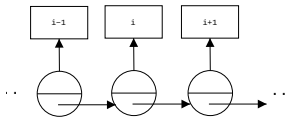
```
public E get( int pos ) {

    Node<E> p = head;

    for ( int i=0; i<pos; i++ ) {
        p = p.next;
    }

    return p.value;
}
```

⇒ Exercice : ajoutez les traitements d'exception.

remove(int pos)

Le retrait d'un élément a beaucoup en commun avec l'accès ; il faut traverser la liste jusqu'à un certain élément.

Considérons le cas général d'abord. L'élément i doit être retiré. Dessinez le diagramme de mémoire représentant cette situation.



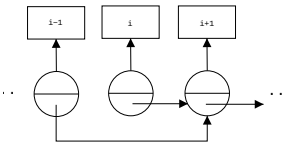
⇒ Quel **.next** doit-on modifier ?

remove(int pos) 1/2

```
public E remove( int pos ) {
    // pre-conditions: ?
    E savedValue;
    Node<E> toDelete;
    Node<E> p = head;
    for ( int i=0; i<( pos-1 ); i++) {
        p = p.next;
    }
    toDelete = p.next;
    p.next = toDelete.next;
    savedValue = toDelete.value;
    toDelete.value = null;
    toDelete.next = null;
    return savedValue;
}
```

⇒ Est-ce que la méthode est suffisamment générale ? Y-a-t-il des cas spéciaux ?

La variable d'instance **next** de l'élément qui précède doit être modifiée.



Supposons que **p** désigne le noeud qui précède,

```
p.next = p.next.next;
```

effectue la transformation nécessaire.

remove(int pos) 2/2

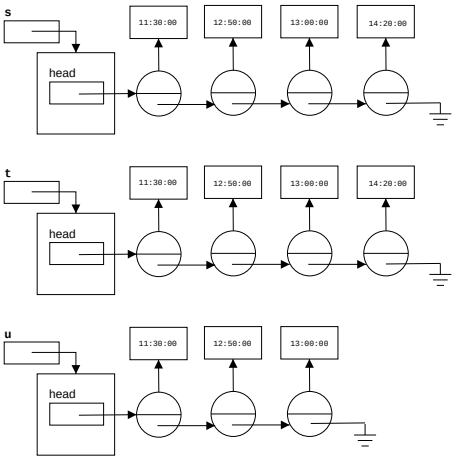
```
public E remove( int pos ) {
    E savedValue;
    Node<E> toDelete;
    if ( pos == 0 ) {
        toDelete = head;
        head = head.next;
    } else {
        Node<E> p = head;
        for ( int i=0; i<( pos-1 ); i++) {
            p = p.next;
        }
        toDelete = p.next;
        p.next = toDelete.next;
    }
    savedValue = toDelete.value;
    toDelete.value = toDelete.next = null;
    return result;
}
```

⇒ Retirer le premier élément, modification de la variable **head** (plutôt que la variable **next** d'un noeud précédent).

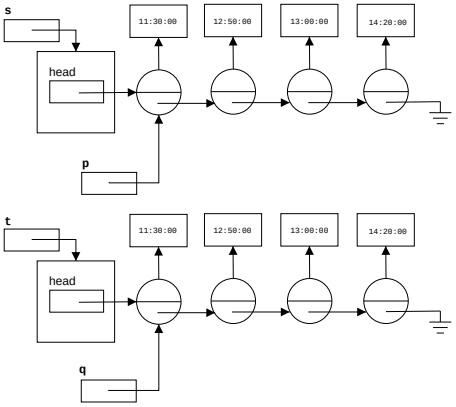
Implémenter la méthode equals

On souhaite implémenter une méthode qui retournera la valeur vrai si deux listes ont un contenu équivalent, c'est-à-dire, que les éléments des deux listes sont équivalents et dans le même ordre.

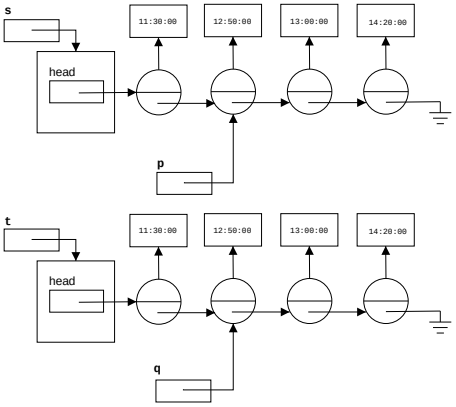
Bien sûr, les listes doivent être de même longueur !



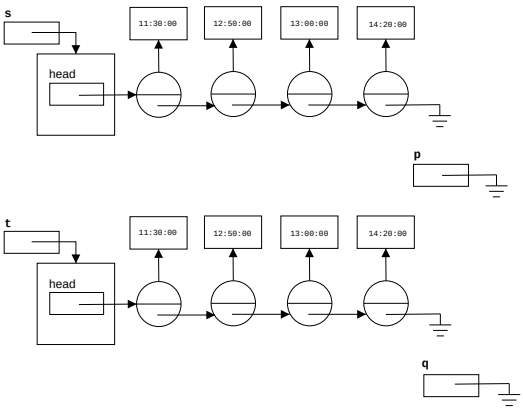

```
s.equals(t); // vrai

s.equals(u); // faux
```



⇒ Qu'est-ce qu'il nous faut ?



⇒ `p = p.next` et `q = q.next`



⇒ éventuellement, la fin de l'une des listes ou les deux sera atteinte.

La méthode infaillible

Utilisés de façon systématique les diagrammes de mémoire peuvent faciliter l'écriture de méthodes modifiant une structure de données ; pas seulement les structures chaînées mais aussi les tableaux.

Si vous **prêtez attention aux détails**, il est presqu'impossible de se tromper !

Afin d'illustrer la technique, nous allons créer une méthode **removeFirst()**.

L'opération **removeFirst()** fait 2 choses : elle retire et retourne le premier élément de la liste.

9 étapes

1. Identifier les entrées/sorties
2. Diagrammes de mémoire AVANT/APRÈS
3. Généralisation
4. Trouver les cas spécifiques
5. Traitement des cas spécifiques
6. Écriture des tests
7. Écriture des blocs d'énoncés
8. Écriture de la méthode (non-optimisée)
9. Optimisation

Étape 0 : identifier les entrées/sorties

removeFirst() est une méthode d'instance qui modifie l'état ; par exemple, **l.removeFirst()** modifie (l'état de) la liste désignée par **l**.

removeFirst() retourne la valeur du premier élément et le retire.

Étape 1 : diagrammes AVANT/APRÈS

Dessinez les diagrammes AVANT et APRÈS pour une entrée typique ; le cas général.

Le diagramme **AVANT** est un diagramme de mémoire qui illustre l'état de la liste (et toutes les autres variables données en entrée, modifiées ou associées aux résultats) au moment qui précède l'exécution de la méthode.

Le diagramme **APRÈS** est un diagramme de mémoire qui illustre l'état de la mémoire immédiatement après l'exécution de la méthode.

Diagramme de mémoire AVANT,

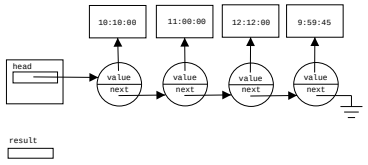
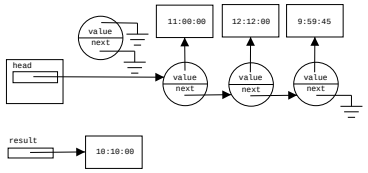


Diagramme de mémoire APRÈS,



⇒ Il est préférable de ne pas regarder le diagramme AVANT afin de dessiner le diagramme APRÈS (ça évite des erreurs).

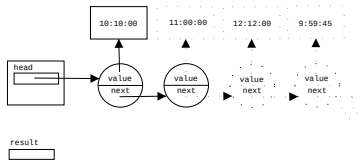
Étape 2 : généralisation

Généralisation des diagrammes AVANT/APRÈS.

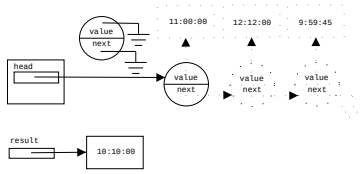
(a) Éliminez des diagrammes tout ce qui n'est pas changé ou utilisé par la méthode.

(b) Changer les constantes par des variables.

AVANT :

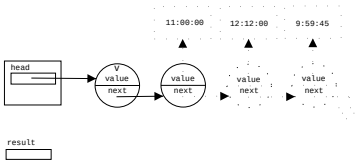


APRÈS :

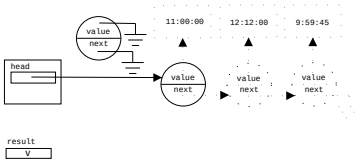


⇒ il faut trouver tous les éléments qui ne sont pas directement impliqués.

AVANT :

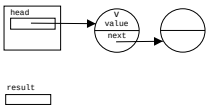


APRÈS :

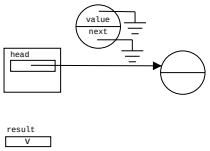


⇒ on associe une variable à chacune des constantes.

AVANT :



APRÈS :



⇒ diagrammes AVANT/APRÈS pour le cas général.

Étape 3 : trouver tous les cas particuliers

Il y a toujours une ou deux exceptions au cas général ; en particulier dans le cas des structures chaînées.

Quelles entrées ne correspondent pas au cas général ?

Suggestions ?

Le cas général, décrit toutes les listes ayant deux éléments ou plus !

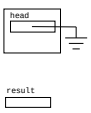
Les cas particuliers sont donc : la liste vide et la liste d'un seul élément.

Les cas particuliers peuvent être classifiés en 2 catégories :

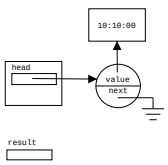
illégaux : une situation qui ne doit pas se présentée, par exemple, retirer un élément d'une liste vide.
Ces cas doivent être proprement documentés et serviront de contre-exemples lors de tests.

spéciaux : une situation légale/valide mais qui n'est pas traitée par le cas général. Par exemple, le cas de la liste d'un seul élément, singleton, est un cas valide mais qui n'est pas traité par le cas général.

Cas illégal, condition d'erreur, **removeFirst()** d'une liste vide ;



Cas particulier, la liste ne contient qu'un seul élément, singleton :



Étape 4 : traitement des cas spéciaux

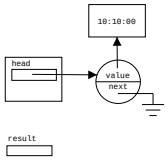
Il y a deux façons de traiter les cas spéciaux :

a) Modifier les diagrammes pour le cas général de sorte qu'ils représentent aussi le ou les cas spéciaux.
Si c'est possible, c'est aussi la meilleure solution, puisqu'elle simplifiera l'écriture de la méthode (moins de cas à traiter).

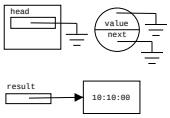
b) Répétez les étapes 1 (diagrammes AVANT/APRÈS) et 2 (généralisation) pour chacun des cas spéciaux.
Créez des diagrammes qui couvrent le plus grand nombre possible de cas spéciaux.
Répétez jusqu'à ce que tous les cas spéciaux soient représentés par l'une des paires de diagrammes.

removeFirst() d'une liste contenant un seul élément.

AVANT :

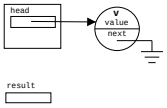


APRÈS :

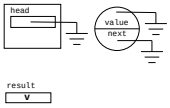


removeFirst() d'une liste contenant un seul élément, diagrammes de mémoire après généralisation.

AVANT :



APRÈS :



⇒ diagrammes AVANT/APRÈS pour le cas général.

Étape 5 : tests

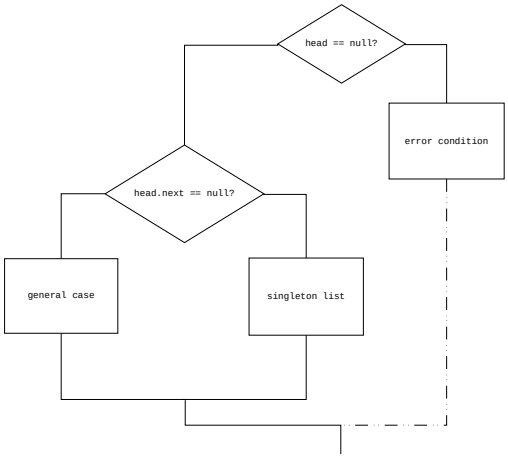
Écrivez l'ensemble des tests qui permettent de distinguer les différents cas, conditions d'erreur, cas particuliers, cas général.

En comparant entre eux les diagrammes AVANT pour chacun des cas, il devrait être possible de distinguer chacun d'eux.

Cas de la liste vide : **first == null** ?

Cas de la liste d'un seul élément ; **first.next == null** ?

Cas général : sinon.



⇒ Lorsqu'on détecte une condition d'erreur il faut lancer une exception.

Étape 6 : écriture des bloc d'énoncés

Pour chaque cas, nous allons écrire un bloc d'énoncés réalisant le traitement à faire, à l'étape 7 nous assemblerons ces blocs qui seront ensuite simplifiés à l'étape 8.

Comment ? Nous allons analyser attentivement les différences entre les diagrammes AVANT et APRÈS.

La première chose à faire est de déclarer une variable pour la valeur de retour ; on sait déjà que chaque bloc devra y accéder.

int returnValue ;

Il y a 3 sous-étapes :

a) i) Identifiez les ≪objets≫ impliqués à l'aide de noms logiques ; par exemple, ici chaque noeud de la liste auquel il nous faudra accéder, **nodeToDelete**, **newFirst**.

ii) Pour chaque nom logique, déclarez une variable et l'initialiser correctement.

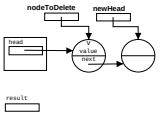
b) Sauvez la valeur de retour.

c) i) Identifiez toutes les différences entre les diagrammes AVANT/APRÈS.

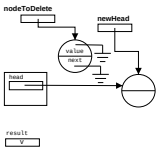
ii) Pour chaque différence, écrire les énoncés qui réalisent la transformation.
Si la sous-étape (a) a été faite correctement, les différences peuvent être traitées dans n'importe quel ordre.

Cas général,

AVANT :



APRÈS :



a) Node nodeToDelete = firstNode ;
Node newFirst = firstNode.next ;

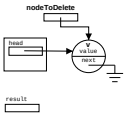
b) returnValue = nodeToDelete.value ;

c) i) – **firstNode** pointe sur **newFirst**,
– la valeur de la variable next de **nodeToDelete** est *null*,
– la valeur de la variable value de **nodeToDelete** est PROPRE.

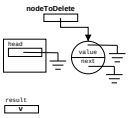
ii) – **firstNode = newFirst ;**
– **nodeToDelete.next = null ;**
– **nodeToDelete.value = PROPRE ;**

Cas particulier, la liste ne contient qu'un seul élément.

AVANT :



APRÈS :



a) Node nodeToDelete = firstNode ;

b) returnValue = nodeToDelete.value ;

c) i) – **firstNode** est **null**.
– la valeur de la variable value de **nodeToDelete** est PROPRE.

ii) – **firstNode = null ;**
– **nodeToDelete.value = PROPRE ;**

Étape 7 : écriture de la méthode

La structure des tests et les blocs pour chacun des cas sont mis ensemble.

```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    if ( firstNode.next == null ) { // singleton
      Node<E> nodeToDelete = firstNode;
      returnValue = nodeToDelete.value;
      firstNode = null;
      nodeToDelete.value = null;
    } else { // cas general
      Node<E> nodeToDelete = firstNode;
      Node<E> newFirst = firstNode.next;
      returnValue = nodeToDelete.value;
      firstNode = newFirst;
      nodeToDelete.next = null;
      nodeToDelete.value = PROPRE;
    }
  }
  return returnValue;
}
```

⇒ Nous avons maintenant une première version qui fonctionne.

Étape 8 : optimisation

Factorisation des énoncés communs :

```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    Node<E> nodeToDelete = firstNode;
    returnValue = nodeToDelete.value;
    nodeToDelete.next = null;
    if (firstNode.next == null) { // singleton
      firstNode = null;
    } else { // cas general
      Node<E> newFirst = firstNode.next;
      firstNode = newFirst;
      nodeToDelete.value = PROPRE;
    }
  }
  return returnValue;
}
```

Éliminez les variables qui ne sont utilisées qu'une seule fois.

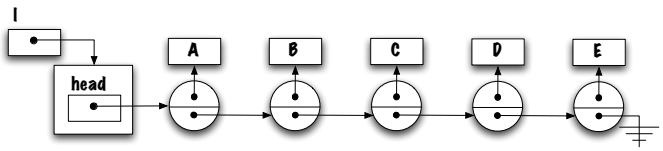
```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    Node<E> nodeToDelete = firstNode;
    returnValue = nodeToDelete.value;
    nodeToDelete.value = PROPRE;
    if ( firstNode.next == null ) { // singleton
      firstNode = null;
    } else { // cas general
      // Node<E> newFirst = firstNode.next;
      // firstNode = newFirst;
      firstNode = firstNode.next; // remplace les deux lignes qui prece
      nodeToDelete.next = null;
    }
  }
  return returnValue;
}
```

L'énoncé suivant, **firstNode = null**, du cas singleton, est équivalent à l'énononcé suivant, **firstNode = firstNode.next**, du cas général, puisque **nodeToDelete.next** vaut **null** ! Cet énoncé se retrouve aussi dans le cas général et peut donc être factorisé.

```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    Node<E> nodeToDelete = firstNode;
    returnValue = nodeToDelete.value;
    nodeToDelete.value = PROPRE;
    firstNode = firstNode.next;
    if (firstNode.next == null) { // singleton
    } else { // cas general
      nodeToDelete.next = null;
    }
  }
  return returnValue;
```

Dans le cas du singleton, nodeToDelete.next est null, mais il n'y a rien de mal à lui affecter la valeur null à nouveau, d'autant plus que ça nous permet de factoriser cette ligne.

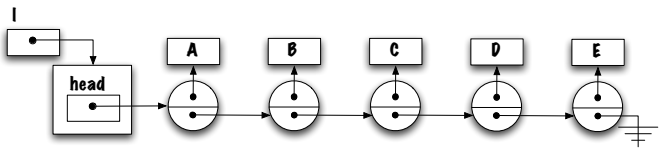
```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    Node<E> nodeToDelete = firstNode;
    returnValue = nodeToDelete.value;
    nodeToDelete.value = PROPRE;
    firstNode = firstNode.next;
    nodeToDelete.next = null;
    if (firstNode.next == null) { // singleton
    } else { // cas general
    }
  }
  return returnValue;
}
```

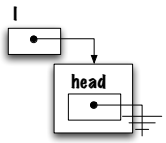



```
}
```

Le cas du singleton et le cas général sont maintenant vides et peuvent être éliminés.

```
public E removeFirst () {
  E returnValue;
  if ( firstNode == null ) {
    throw new IndexOutOfBoundsException();
  } else {
    Node<E> nodeToDelete = firstNode;
    returnValue = nodeToDelete.value;
    nodeToDelete.value = PROPRE;
    firstNode = firstNode.next;
    nodeToDelete.next = null;
  }
  return returnValue;
}
```




l
head