

ITI 1521. Introduction à l'informatique II

Liste : traitement récursif des listes chaînées.

by

Marcel Turcotte

Version du 22 mars 2020

Préambule

Préambule

Aperçu

Liste : traitement récursif des listes chaînées.

Nous revisitons le concept de récursivité, cette fois dans le contexte du traitement des listes chaînées. Nous développons une stratégie générale, « head & tail », que vous pourrez appliquer à tous les problèmes abordés dans ce cours.

Objectif général :

- Cette semaine, vous serez en mesure de concevoir des méthodes récursives pour le traitement des listes chaînées.

«*To iterate is human, to recurse divine*»

L. Peter Deutsch

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- ❖ **Reconnaitre** les problèmes pour lesquels la récursivité est appropriée.
- ❖ **Discuter** l'efficacité du traitement récursif des listes en Java, notamment par rapport à la consommation de mémoire.
- ❖ **Expliquer** le rôle des paramètres pour contrôler le flot d'exécution des programmes récursifs.
- ❖ **Paraphraser** la stratégie « head & tail » discutée en classe pour le traitement récursif des listes de données.
- ❖ **Utiliser** la stratégie « head & tail » afin de concevoir une méthode récursive pour le traitement d'une liste chaînée.

Lectures :

- ❖ Pages 233-238 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

- 1 Préambule
- 2 Théorie
- 3 Implémentation
- 4 Principes
- 5 Prologue

Théorie

Discussion

- ✚ Quels **problèmes** avez-vous résolu à l'aide de la récursivité ?

Discussion

- ❖ Quels **problèmes** avez-vous résolu à l'aide de la récursivité ?
- ❖ Qu'ont **en commun** ces problèmes ?

- ❖ La **solution** d'un problème donné peut être construite **à partir des solutions** de **sous problèmes** ;

- ❖ La **solution** d'un problème donné peut être construite **à partir des solutions** de **sous problèmes** ;
- ❖ Ces sous problèmes sont **de même nature** et peuvent donc être **résolus de la même manière** ;

- ❖ La **solution** d'un problème donné peut être construite **à partir des solutions** de **sous problèmes** ;
- ❖ Ces sous problèmes sont **de même nature** et peuvent donc être **résolus de la même manière** ;
- ❖ Les sous problèmes sont **de plus en plus petits (convergence)** ;

- ❖ La **solution** d'un problème donné peut être construite **à partir des solutions** de **sous problèmes** ;
- ❖ Ces sous problèmes sont **de même nature** et peuvent donc être **résolus de la même manière** ;
- ❖ Les sous problèmes sont **de plus en plus petits (convergence)** ;
- ❖ Finalement, il existe une taille de problèmes pouvant être résolus de façon **triviale**, sans recours à des appels récursifs, ce sont les **cas de base**.

Remarques

- ✚ La **récurtivité** et l'**itération** sont de même nature.

Remarques

- ❖ La **récurtivité** et l'**itération** sont de même nature.
- ❖ Il est important que la **taille des problèmes à traiter diminue**, sinon il y aurait **un nombre infini d'appels récursifs** (équivalent de la boucle infinie); en pratique le programme terminera lorsque toute la mémoire réservée pour les appels de méthodes est épuisée.

Remarques

- ❖ La **récurtivité** et l'**itération** sont de même nature.
- ❖ Il est important que la **taille des problèmes à traiter diminue**, sinon il y aurait **un nombre infini d'appels récursifs** (équivalent de la boucle infinie); en pratique le programme terminera lorsque toute la mémoire réservée pour les appels de méthodes est épuisée.
- ❖ Il faut donc qu'il y ait **une taille de problèmes pouvant être résolus sans appel récursifs**, afin de stopper la récursivité.

Remarques

- ❖ La **récurtivité** et l'**itération** sont de même nature.
- ❖ Il est important que la **taille des problèmes à traiter diminue**, sinon il y aurait **un nombre infini d'appels récursifs** (équivalent de la boucle infinie); en pratique le programme terminera lorsque toute la mémoire réservée pour les appels de méthodes est épuisée.
- ❖ Il faut donc qu'il y ait **une taille de problèmes pouvant être résolus sans appel récursifs**, afin de stopper la récursivité.
- ❖ **Ce sont les cas de base**. Il faut qu'il y en ait au moins un, mais il peut y en avoir plusieurs.

Remarques

- ❖ La **récurtivité** et l'**itération** sont de même nature.
- ❖ Il est important que la **taille des problèmes à traiter diminue**, sinon il y aurait **un nombre infini d'appels récursifs** (équivalent de la boucle infinie); en pratique le programme terminera lorsque toute la mémoire réservée pour les appels de méthodes est épuisée.
- ❖ Il faut donc qu'il y ait **une taille de problèmes pouvant être résolus sans appel récursifs**, afin de stopper la récursivité.
- ❖ **Ce sont les cas de base**. Il faut qu'il y en ait au moins un, mais il peut y en avoir plusieurs.
- ❖ Les **cas de base doivent être traités en premier** afin de stopper la récursivité, si nécessaire.

Remarques (suite)

- ❖ Les langages de programmation **Lisp**, **Prolog** et **Haskell**, pour en nommer que quelques-uns, n'ont aucun énoncé de contrôle itératif, tout se fait à l'aide de la récursivité.

Remarques (suite)

- ❖ Les langages de programmation **Lisp**, **Prolog** et **Haskell**, pour en nommer que quelques-uns, n'ont aucun énoncé de contrôle itératif, tout se fait à l'aide de la récursivité.
 - ❖ Les compilateurs transforment automatiquement certaines formes de récursivité en itération.

Remarques (suite)

- ❖ Les langages de programmation **Lisp**, **Prolog** et **Haskell**, pour en nommer que quelques-uns, n'ont aucun énoncé de contrôle itératif, tout se fait à l'aide de la récursivité.
 - ❖ Les compilateurs transforment automatiquement certaines formes de récursivité en itération.
- ❖ La technologie des **Transformations XSLT** que l'on utilise notamment pour certaines applications Web repose sur le concept de récursivité.

Remarques (suite)

- ❖ Les langages de programmation **Lisp**, **Prolog** et **Haskell**, pour en nommer que quelques-uns, n'ont aucun énoncé de contrôle itératif, tout se fait à l'aide de la récursivité.
 - ❖ Les compilateurs transforment automatiquement certaines formes de récursivité en itération.
- ❖ La technologie des **Transformations XSLT** que l'on utilise notamment pour certaines applications Web repose sur le concept de récursivité.
- ❖ Certains traitements des **arbres binaires de recherche** seront très simples à exprimer à l'aide de la récursivité, mais très complexes sinon.

Théorie

Patron

Patron

```
type method(parameters) {
  type result;
  if (test) { // base case
    result = calculating the result // no recursive call
  } else { // general case
    // pre-processing:
    // (partitioning the data)
    result = method(sub-problem); // recursive call
    // post-processing:
    // (combine the result)
  }
  return result;
}
```

Théorie

Factorielle

Factorielle

```
public static int factorial(int n) {  
    int s, result;  
    if (n<=1) { // base case  
        result = 1;  
    } else {    // general case  
        int n1 = n-1;  
        s = factorial(n1);  
        result = n * s;  
    }  
    return result;  
}
```

Factorielle

```
public static int factorial(int n) {  
    int s, result;  
    if (n<=1) { // base case  
        result = 1;  
    } else { // general case  
        int n1 = n-1;  
        s = factorial(n1);  
        result = n * s;  
    }  
    return result;  
}
```

- ❖ La méthode ci-haut correspond au modèle proposé :
 - ❖ On vérifie d'abord le **cas de base**, son résultat est calculé sans appel récursif (la récursivité s'arrête ici !);
 - ❖ Le **cas général** crée des sous-problèmes de plus en plus petits.

Factorielle — implémentation plus compacte

```
public static int factorial(int n) {  
    if (n<=1) {  
        return 1;  
    }  
  
    return n * factorial(n-1);  
}
```

Factorielle — implémentation plus compacte

```
public static int factorial(int n) {  
  
    if (n<=1) {  
        return 1;  
    }  
  
    return n * factorial(n-1);  
}
```

- ❖ L'énoncé **return** retourne le contrôle à l'appelant, il stop l'exécution de la méthode, **aucun autre énoncé de cet appel ne sera exécuté.**

Factorielle — implémentation plus compacte

```
public static int factorial(int n) {  
  
    if (n < 0) {  
        throw new IllegalArgumentException(Integer.toString(n));  
    }  
  
    if (n <= 1) {  
        return 1;  
    }  
  
    return n * factorial(n-1);  
}
```

Théorie : «*head & tail*»

- ❖ Développons une **stratégie générale** pour le **traitement récursif des listes**.

*Ici, **head** et **tail** ne sont pas des variables d'instance.

Théorie : «*head & tail*»

- ❖ Développons une **stratégie générale** pour le **traitement récursif des listes**.
- ❖ **Divisons la liste en deux parties**, le premier élément (**head**) et le reste de la liste (**tail**)*.

*Ici, **head** et **tail** ne sont pas des variables d'instance.

Implémentation

Implémentation de la classe LinkedList

- ✚ Nous utiliserons une liste **simplement** chaînée.

```
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;  
  
    // ...  
}
```

Implémentation

size

Calculer la taille d'une liste

Considérons d'abord le calcul de la **taille** d'une liste (**size**).

- ✦ Soit **current**, une variable de type **Node** désignant un élément de la liste.

Calculer la taille d'une liste

Considérons d'abord le calcul de la **taille** d'une liste (**size**).

- ❖ Soit **current**, une variable de type **Node** désignant un élément de la liste.
- ❖ Sachant que la taille de la liste débutant avec l'élément désigné par **current.next** est **n**,

Calculer la taille d'une liste

Considérons d'abord le calcul de la **taille** d'une liste (**size**).

- ❖ Soit **current**, une variable de type **Node** désignant un élément de la liste.
- ❖ Sachant que la taille de la liste débutant avec l'élément désigné par **current.next** est **n**,
 - ❖ quelle est la taille de la liste débutant avec l'élément désigné par **current** ?

Calculer la taille d'une liste

Considérons d'abord le calcul de la **taille** d'une liste (**size**).

- ❖ Soit **current**, une variable de type **Node** désignant un élément de la liste.
- ❖ Sachant que la taille de la liste débutant avec l'élément désigné par **current.next** est **n**,
 - ❖ quelle est la taille de la liste débutant avec l'élément désigné par **current** ?
- ❖ La taille de la liste débutant par l'élément désigné par **current** est **n+1**.

Calculer la taille d'une liste

- ❖ La stratégie «*head & tail*» suggère que l'on commence la réflexion en posant l'**appel récursif**, passant **en paramètre `current.next`**.

```
int n = size(current.next);
```

Calculer la taille d'une liste

- ❖ La stratégie «*head & tail*» suggère que l'on commence la réflexion en posant l'**appel récursif**, passant **en paramètre `current.next`**.

```
int n = size(current.next);
```

- ❖ Quelle est la valeur de **n**? Que signifie **n**?

Calculer la taille d'une liste

- ❖ La stratégie «*head & tail*» suggère que l'on commence la réflexion en posant l'**appel récursif**, passant **en paramètre `current.next`**.

```
int n = size(current.next);
```

- ❖ Quelle est la valeur de **n**? Que signifie **n**?
 - ❖ C'est la longueur de la liste débutant par l'élément désigné par **`current.next`**.

Calculer la taille d'une liste

- ❖ La stratégie «*head & tail*» suggère que l'on commence la réflexion en posant l'**appel récursif**, passant **en paramètre `current.next`**.

```
int n = size(current.next);
```

- ❖ Quelle est la valeur de **n**? Que signifie **n**?
 - ❖ C'est la longueur de la liste débutant par l'élément désigné par **`current.next`**.
- ❖ Quelle est la taille de la liste débutant avec l'élément désigné par **`current`**?

Calculer la taille d'une liste

- ❖ La stratégie «*head & tail*» suggère que l'on commence la réflexion en posant l'**appel récursif**, passant **en paramètre `current.next`**.

```
int n = size(current.next);
```

- ❖ Quelle est la valeur de **n**? Que signifie **n**?
 - ❖ C'est la longueur de la liste débutant par l'élément désigné par **`current.next`**.
- ❖ Quelle est la taille de la liste débutant avec l'élément désigné par **`current`**?
 - ❖ La longueur de la liste débutant par l'élément désigné par **`current`** est **`n+1`**.

Calculer la taille d'une liste

- ✚ Quelle est la **plus petite liste valide** et quelle est sa **longueur** ?

Calculer la taille d'une liste

- ✚ Quelle est la **plus petite liste valide** et quelle est sa **longueur** ?
 - ✚ C'est la **liste vide** et sa longueur est **0**.

Calculer la taille d'une liste

- ❖ Quelle est la **plus petite liste valide** et quelle est sa **longueur** ?
 - ❖ C'est la **liste vide** et sa longueur est **0**.
- ❖ Quelle est la valeur de **current** si la liste est vide ?

Calculer la taille d'une liste

- ❖ Quelle est la **plus petite liste valide** et quelle est sa **longueur** ?
 - ❖ C'est la **liste vide** et sa longueur est **0**.
- ❖ Quelle est la valeur de **current** si la liste est vide ?
 - ❖ La valeur de **current** est **null**.

Calculer la taille d'une liste

- ▣ Ce qui suggère l'implémentation partielle suivante :

```
int n;  
  
if (current == null) {  
    n = 0;  
}  
else {  
    n = 1 + size(current.next);  
}
```

Calculer la taille d'une liste

- ❖ Ce qui suggère l'implémentation partielle suivante :

```
int n;  
  
if (current == null) {  
    n = 0;  
}  
else {  
    n = 1 + size(current.next);  
}
```

- ❖ Quel est le **type** du paramètre de la méthode **size** ?

Calculer la taille d'une liste

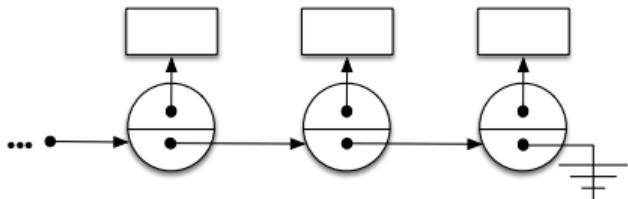
- ❖ Ce qui suggère l'implémentation partielle suivante :

```
int n;  
  
if (current == null) {  
    n = 0;  
}  
else {  
    n = 1 + size(current.next);  
}
```

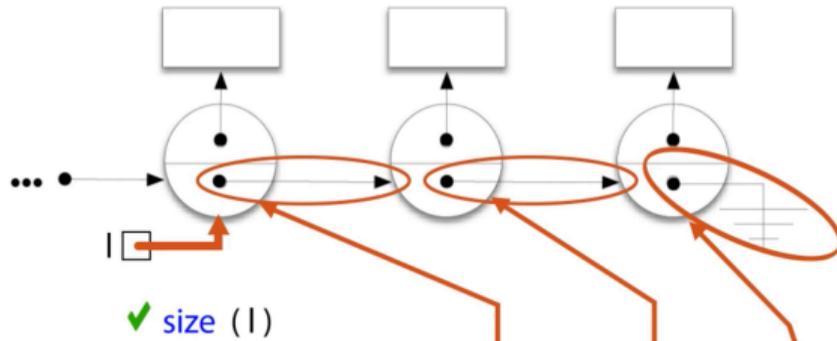
- ❖ Quel est le **type** du paramètre de la méthode **size** ?
 - ❖ Le **type** du paramètre est **Node**.

Calculer la taille d'une liste

```
int size(Node<E> current) {  
  
    int n;  
  
    if (current == null) {  
        n = 0;  
    } else {  
        n = 1 + size(current.next);  
    }  
  
    return n;  
}
```



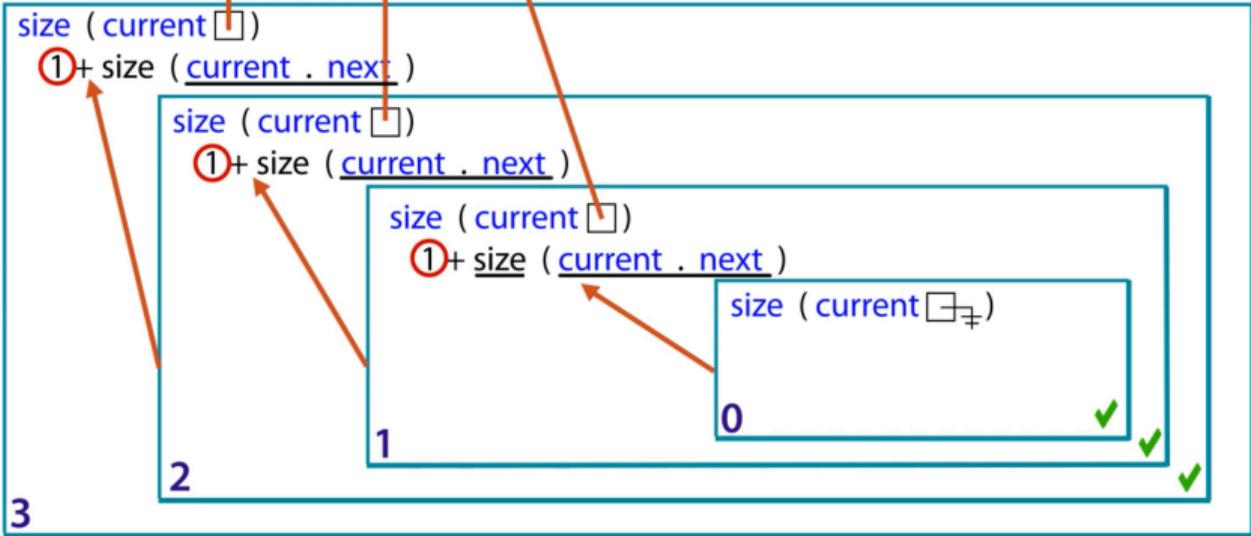
```
int size(Node<E> current) {  
    if (current == null) {  
        return 0;  
    }  
    return 1 + size(current.next);  
}
```



```

int size ( Node<E> current ) {
    if ( current == null ) {
        return 0;
    }
    return 1 + size ( current . next );
}

```



Remarques

- ❖ Remarquez que la méthode **size** n'utilise **aucune variable d'instance** !

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

Remarques

- ❖ Remarquez que la méthode **size** n'utilise **aucune variable d'instance** !
 - ❖ On **contrôle la récursivité à l'aide du paramètre**.

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

Remarques

- ❖ Remarquez que la méthode **size** n'utilise **aucune variable d'instance** !
 - ❖ On **contrôle la récursivité à l'aide du paramètre**.
 - ❖ Chaque **appel** possède sa **mémoire de travail** (bloc d'activation) et donc ses propres copies des variables locales et des paramètres.

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

Remarques

- ❖ Remarquez que la méthode **size** n'utilise **aucune variable d'instance** !
 - ❖ On **contrôle la récursivité à l'aide du paramètre**.
 - ❖ Chaque **appel** possède sa **mémoire de travail** (bloc d'activation) et donc ses propres copies des variables locales et des paramètres.

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

- ❖ Le **cas de base** est validé en premier.

Lancer la méthode size

- ❖ **Comment** utilise-t-on cette méthode afin de calculer la taille de la liste débutant par le noeud désigné par **head** ?

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

Lancer la méthode size

- ❖ **Comment** utilise-t-on cette méthode afin de calculer la taille de la liste débutant par le noeud désigné par **head** ?

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

```
int size() {  
    return size(head);  
}
```

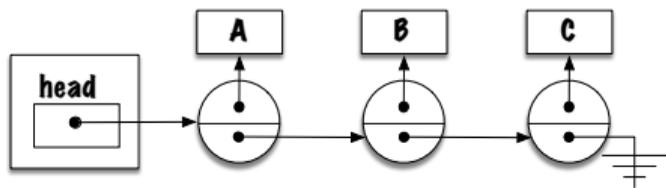
La méthode récursive est un compagnon

- Le premier appel est initié par une méthode de visibilité **public**. On passe la valeur de **head** en paramètre.

```
public int size() {  
    return size(head);  
}
```

- La méthode **récursive** doit être de visibilité **privée** puisque son paramètre est de type **Node**.

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



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

- ❖ Chaque appel a son propre **bloc d'activation** (mémoire de travail) sur la pile des appels, ainsi la taille de la pile système sera proportionnelle à la taille de la liste.

Implémentation

Résumé

Résumé

```
type method(Node<E> current) {  
  
    type result;  
  
    if (current ...) {           // base case  
        calculating the result   // no recursive call  
    } else {                     // general case  
        // pre-processing  
        s = method(current.next); // recursion  
        // post-processing  
    }  
  
    return result;  
}
```

«*head & tail*»

Étapes :

- ▣ Que signifie `méthode(current.next)` ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?
 - ❖ Quelle est la **plus petite liste valide** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?
 - ❖ Quelle est la **plus petite liste valide** ?
 - ❖ Quel est le **résultat** ?

Implémentation

`findMax`

LinkedList

- Utilisons maintenant une liste dont les éléments possèdent une méthode **compareTo**.

```
public class LinkedList<E extends Comparable<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;  
  
    // ...  
}
```

findMax

- **Appliquons** la stratégie telle que proposée.

```
result = findMax(current.next);
```

findMax

- ❖ **Appliquons** la stratégie telle que proposée.

```
result = findMax(current.next);
```

- ❖ Quelle est la valeur de **result** ? Que signifie **result** ?

findMax

- ❖ **Appliquons** la stratégie telle que proposée.

```
result = findMax(current.next);
```

- ❖ Quelle est la valeur de **result** ? Que signifie **result** ?
 - ❖ La **plus grande valeur** pour la liste débutant avec l'élément désigné par **current.next**

findMax

- ❖ **Appliquons** la stratégie telle que proposée.

```
result = findMax(current.next);
```

- ❖ Quelle est la valeur de **result** ? Que signifie **result** ?
 - ❖ La **plus grande valeur** pour la liste débutant avec l'élément désigné par **current.next**
- ❖ Que fait-on si **result** est plus grand que **current.value** ?

findMax

- ❖ Appliquons la stratégie telle que proposée.

```
result = findMax(current.next);
```

- ❖ Quelle est la valeur de **result** ? Que signifie **result** ?
 - ❖ La **plus grande valeur** pour la liste débutant avec l'élément désigné par **current.next**
- ❖ Que fait-on si **result** est plus grand que **current.value** ?

```
if (result.compareTo(current.value) > 0) {  
    return result;  
} else {  
    return current.value;  
}
```

- ❖ Ce processus construit des problèmes de plus en plus petits, **quelle est la plus petite liste valide ?**

- ❖ Ce processus construit des problèmes de plus en plus petits, **quelle est la plus petite liste valide ?**
 - ❖ **Non, pas la liste vide**, mais la liste contenant **un seul élément**.

findMax

- ❖ Ce processus construit des problèmes de plus en plus petits, **quelle est la plus petite liste valide ?**
 - ❖ **Non, pas la liste vide**, mais la liste contenant **un seul élément**.
- ❖ Quelle sera la **valeur retournée ?**

```
if (current.next == null) {  
    return current.value;  
}
```

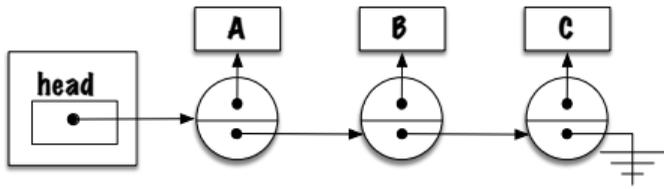
```
public E findMax() {
    if (head == null) {
        throw new NoSuchElementException();
    }
    return findMax(head);
}

private E findMax(Node<E> current) {

    if (current.next == null) {
        return current.value;
    }

    E result = findMax(current.next);

    if (result.compareTo(current.value) > 0) {
        return result;
    } else {
        return current.value;
    }
}
```



```
private E findMax(Node<E> p) {  
    if (p.next == null) {  
        return p.value;  
    }  
    E r = findMax(p.next);  
    if (r.compareTo(p.value) > 0) {  
        return r;  
    } else {  
        return p.value;  
    }  
}
```

Chaque **exemple** qui suit introduit une **nouvelle problématique.**

Implémentation

`E get(int index)`

E get(int index)

- La méthode **E get(int index)** retourne l'élément à la valeur spécifiée (**index**) de la liste.

E get(int index)

- ❖ La méthode **E get(int index)** retourne l'élément à la valeur spécifiée (**index**) de la liste.
- ❖ Quelle était la **stratégie** adoptée pour la méthode **non réursive** ?

E get(int index)

- ❖ La méthode **E get(int index)** retourne l'élément à la valeur spécifiée (**index**) de la liste.
- ❖ Quelle était la **stratégie** adoptée pour la méthode **non réursive** ?
 - ❖ Il fallait compter le nombre de noeuds visités et terminer l'exécution de la boucle **while** après avoir visité **index** noeuds.

E get(int index)

- ❖ La méthode **E get(int index)** retourne l'élément à la valeur spécifiée (**index**) de la liste.
- ❖ Quelle était la **stratégie** adoptée pour la méthode **non réursive** ?
 - ❖ Il fallait compter le nombre de noeuds visités et terminer l'exécution de la boucle **while** après avoir visité **index** noeuds.
- ❖ Pour une méthode **réursive**, comment **détermine-t-on le nombre de noeuds visités** ?

E get(int index)

- ❖ La méthode **E get(int index)** retourne l'élément à la valeur spécifiée (**index**) de la liste.
- ❖ Quelle était la **stratégie** adoptée pour la méthode **non réursive** ?
 - ❖ Il fallait compter le nombre de noeuds visités et terminer l'exécution de la boucle **while** après avoir visité **index** noeuds.
- ❖ Pour une méthode **réursive**, comment **détermine-t-on le nombre de noeuds visités** ?
 - ❖ On pourrait ajouter un **paramètre** afin de compter le nombre de noeuds visités. Initialement **0**, puis **1**, **2**, **etc.**

❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Si **index** représente la position de l'élément par rapport à la liste débutant à la position **current**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **current.next** ?

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

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

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Si **index** représente la position de l'élément par rapport à la liste débutant à la position **current**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **current.next** ?
 - ❖ C'est bien ça, **index-1**.
- ❖ Que fera la méthode si la valeur de l'**index est 0** ?

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Si **index** représente la position de l'élément par rapport à la liste débutant à la position **current**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **current.next** ?
 - ❖ C'est bien ça, **index-1**.
- ❖ Que fera la méthode si la valeur de l'**index est 0** ?
 - ❖ Elle doit retourner **current.value**.

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Si **index** représente la position de l'élément par rapport à la liste débutant à la position **current**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **current.next** ?
 - ❖ C'est bien ça, **index-1**.
- ❖ Que fera la méthode si la valeur de l'**index est 0** ?
 - ❖ Elle doit retourner **current.value**.
 - ❖ **Aucun** appel récursif.

- ❖ Étudiez l'implémentation partielle suivante :

```
public E get(int index) {  
    return get(head, index);  
}  
  
private E get(Node<E> current, int index) {  
    ...  
}
```

- ❖ Si **index** représente la position de l'élément par rapport à la liste débutant à la position **current**, quelle est la position de l'élément recherché par rapport à la liste débutant à la position **current.next** ?
 - ❖ C'est bien ça, **index-1**.
- ❖ Que fera la méthode si la valeur de l'**index est 0** ?
 - ❖ Elle doit retourner **current.value**.
 - ❖ **Aucun** appel récursif.
 - ❖ C'est le **cas de base**.

E get(int index)

```
private E get(Node<E> current, int index) {  
    if (index == 0) {  
        return current.value;  
    }  
  
    return get(current.next, index - 1);  
}
```

E get(int index)

```
private E get(Node<E> current, int index) {  
    if (index == 0) {  
        return current.value;  
    }  
  
    return get(current.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 ?

E get(int index)

```
private E get(Node<E> current, int index) {  
    if (index == 0) {  
        return current.value;  
    }  
  
    return get(current.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 **current == null**

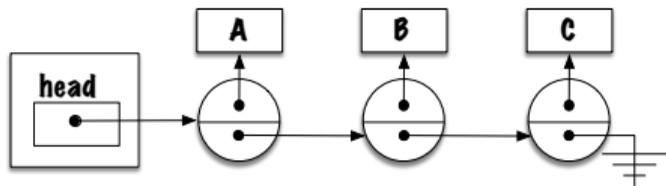
E get(int index)

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

E get(int index)

```
public E get(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    }  
    return get(head, index);  
}
```

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



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

Implémentation

```
int indexOf(E element)
```

int indexOf(E element)

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

int indexOf(E element)

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

int indexOf(E element)

- ❖ Selon la stratégie «*head & tail*», le cas général comportera un appel récursif tel que celui-ci :

```
s = indexOf(current.next, element);
```

int indexOf(E element)

- ❖ Selon la stratégie «*head & tail*», le cas général comportera un appel récursif tel que celui-ci :

```
s = indexOf(current.next, element);
```

- ❖ Que représente la valeur de s ?

int indexOf(E element)

- ❖ Selon la stratégie «*head & tail*», le cas général comportera un appel récursif tel que celui-ci :

```
s = indexOf(current.next, element);
```

- ❖ Que représente la valeur de **s**?
 - ❖ C'est la position de l'**élément** dans la liste désignée par **current.next**.

int indexOf(E element)

- ❖ Selon la stratégie «*head & tail*», le cas général comportera un appel récursif tel que celui-ci :

```
s = indexOf(current.next, element);
```

- ❖ Que représente la valeur de **s**?
 - ❖ C'est la position de l'**élément** dans la liste désignée par **current.next**.
- ❖ Par rapport à la liste courante, celle désignée par **current**, quelle est la position d'**élément** ?

int indexOf(E element)

- ❖ Selon la stratégie «*head & tail*», le cas général comportera un appel récursif tel que celui-ci :

```
s = indexOf(current.next, element);
```

- ❖ Que représente la valeur de **s**?
 - ❖ C'est la position de l'**élément** dans la liste désignée par **current.next**.
- ❖ Par rapport à la liste courante, celle désignée par **current**, quelle est la position d'**element**?
 - ❖ **s + 1**

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?
 - ❖ L'élément était **absent** du reste de la liste.

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?
 - ❖ L'élément était **absent** du reste de la liste.
- ❖ Quel cas **n'a pas** été traité ?

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?
 - ❖ L'élément était **absent** du reste de la liste.
- ❖ Quel cas **n'a pas** été traité ?
 - ❖ **current.value.equals(element)**

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?
 - ❖ L'élément était **absent** du reste de la liste.
- ❖ Quel cas **n'a pas** été traité ?
 - ❖ **current.value.equals(element)**
 - ❖ Quelle **valeur** doit-on alors **retourner** ?

int indexOf(E element)

- ❖ Si la valeur de **s** est **plus grande ou égale à zéro**, **s** est la **position de l'élément** dans le **reste de la liste**.
- ❖ Que signifie **s == -1** ?
 - ❖ L'élément était **absent** du reste de la liste.
- ❖ Quel cas **n'a pas** été traité ?
 - ❖ **current.value.equals(element)**
 - ❖ Quelle **valeur** doit-on alors **retourner** ?
 - 0

int indexOf(E element)

```
s = indexOf(current.next, element);  
  
if (current.value.equals(element) ) {  
    result = 0;  
} else if (s == -1) {  
    result = s;  
} else {  
    result = 1 + s;  
}
```

int indexOf(E element)

- ❖ Qu'elle est le **cas de base** ?

int indexOf(E element)

- ❖ 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**.

int indexOf(E element)

❖ 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 (current == null) {  
    return -1;  
}
```

int indexOf(E element)

```
private int indexOf(Node<E> current , E element) {  
  
    if (current == null) {  
        return -1;  
    }  
  
    int result = indexOf(current.next , element);  
  
    if (current.value.equals(element)) {  
        return 0;  
    }  
  
    if (result == -1) {  
        return result;  
    }  
  
    return result + 1;  
}
```

int indexOf(E element)

- Est-ce que ça fonctionne ?

int indexOf(E element)

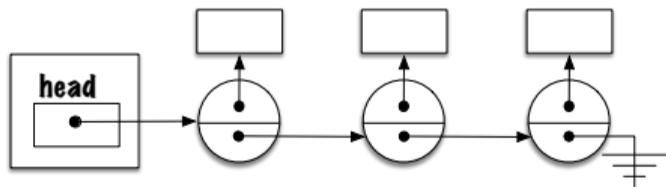
- Est-ce que ça fonctionne?
 - Oui.

int indexOf(E element)

- ❖ Est-ce que **ça fonctionne** ?
 - ❖ Oui.
- ❖ Il tout de même un **problème** avec cette implémentation.

int indexOf(E element)

- ❖ Est-ce que **ça fonctionne** ?
 - ❖ Oui.
- ❖ Il tout de même un **problème** avec cette implémentation.
 - ❖ **Quel** est-il ?



```
int indexOf(Node<E> p, E e) {  
    if (p == null) return -1;  
    int r = indexOf(p.next, e);  
    if (p.value.equals(e)) return 0;  
    if (r == -1) return r;  
    return r + 1;  
}
```

int indexOf(E element)

- Comment **freiner** les appels récursifs dès que la valeur recherchée a été trouvée?

```
private int indexOf(Node<E> current, E element) {  
    if (current == null) {  
        return -1;  
    }  
    int result = indexOf(current.next, element);  
    if (current.value.equals(element)) {  
        return 0;  
    }  
    if (result == -1) {  
        return result;  
    }  
    return result + 1;  
}
```

int indexOf(E element)

```
private int indexOf(Node<E> current, E element) {  
  
    if (current == null) {  
        return -1;  
    }  
  
    if (current.value.equals(element)) {  
        return 0;  
    }  
  
    int result = indexOf(current.next, element);  
  
    if (result == -1) {  
        return result;  
    }  
  
    return result + 1;  
}
```

Implémentation

E indexOfLast(E element)

E indexOfLast(E element)

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

E indexOfLast(E element)

- ❖ La méthode **indexOfLast** retourne la position de la dernière occurrence (celle la plus à droite) de l'**élément**, et -1 sinon.
- ❖ Quels sont les **changements** à apporter ?

E indexOfLast(E element)

- ❖ La méthode **indexOfLast** retourne la position de la dernière occurrence (celle la plus à droite) de l'**élément**, et -1 sinon.
- ❖ Quels sont les **changements** à apporter ?
- ❖ Est-ce que **current.value.equals(element)** peut faire partie du cas base ?

E indexOfLast(E element)

- ❖ La méthode **indexOfLast** retourne la position de la dernière occurrence (celle la plus à droite) de l'**élément**, et -1 sinon.
- ❖ Quels sont les **changements** à apporter ?
- ❖ Est-ce que **current.value.equals(element)** peut faire partie du cas base ?
 - ❖ **Non**, la récursivité doit parcourir la liste en entier.

E indexOfLast(E element)

- ❖ La méthode **indexOfLast** retourne la position de la dernière occurrence (celle la plus à droite) de l'**élément**, et -1 sinon.
- ❖ Quels sont les **changements** à apporter ?
- ❖ Est-ce que **current.value.equals(element)** peut faire partie du cas base ?
 - ❖ **Non**, la récursivité doit parcourir la liste en entier.
- ❖ Comment traiter le résultat **indexOfLast(current.next, element)** ?

```
public int indexOfLast(E element) {
    return indexOfLast(head, element);
}

private int indexOfLast(Node<E> current, E element) {

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

    int result = indexOfLast(current.next, element);

    if (result > -1) {
        return result + 1;
    } else if (element.equals(current.value)) {
        return 0;
    }

    return -1;
}
```

Implémentation

boolean contains(E element)

Exercice

❖ `boolean contains(E element)`

Implémentation

`boolean isIncreasing()`

boolean `isIncreasing()`

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

boolean `isIncreasing()`

- ❖ Les méthodes **size**, **indexOf** et **contains** ne traitent qu'un élément à la fois.
- ❖ Considérons une implémentation récursive de la méthode **isIncreasing**.

boolean isIncreasing()

- ❖ Les méthodes **size**, **indexOf** et **contains** ne traitent qu'un élément à la fois.
- ❖ Considérons une implémentation récursive de la méthode **isIncreasing**.
- ❖ **Examiner** chaque paire consécutive et de retourner la valeur **false** dès qu'une paire n'est pas croissante.

boolean isIncreasing()

- ❖ Les méthodes **size**, **indexOf** et **contains** ne traitent qu'un élément à la fois.
- ❖ Considérons une implémentation récursive de la méthode **isIncreasing**.
- ❖ **Examiner** 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 isIncreasing(head);  
}
```

boolean isIncreasing()

- ❖ Quel est le **cas de base** ?

boolean isIncreasing()

- ❖ Quel est le **cas de base** ?
 - ❖ Quelle est la **plus petite liste** valide.

boolean isIncreasing()

- ❖ Quel est le **cas de base** ?
 - ❖ Quelle est la **plus petite liste** valide.
 - ❖ La liste **vide** et le **singleton** sont croissants.

boolean isIncreasing()

- ❖ Quel est le **cas de base** ?
 - ❖ Quelle est la **plus petite liste** valide.
 - ❖ La liste **vide** et le **singleton** sont croissants.

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

boolean isIncreasing()

Cas général.

- ▣ Quelle approche est **préférable** ?

boolean isIncreasing()

Cas général.

- ❖ Quelle approche est **préférable** ?
 1. Faire un **appel récursif**, puis **traiter le résultat**.

boolean isIncreasing()

Cas général.

- ❖ Quelle approche est **préférable** ?
 1. Faire un **appel récursif**, puis **traiter le résultat**.
 2. Traiter la **position courante**, puis faire un **appel récursif**.

boolean isIncreasing()

Cas général.

- ❖ Quelle approche est **préférable** ?
 1. Faire un **appel récursif**, puis **traiter le résultat**.
 2. Traiter la **position courante**, puis faire un **appel récursif**.

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

boolean isIncreasing()

```
private boolean isIncreasing(Node<E> current) {  
    if ((current == null) || (current.next == null)) {  
        return true;  
    }  
  
    if (current.value.compareTo(current.next.value) > 0 ) {  
        return false;  
    }  
  
    return isIncreasing(current.next);  
}
```

Implémentation

Exercices

Exercices

- ❖ `void addLast(E element)`
- ❖ `boolean equals(LinkedList<E> other)`

Implémentation

```
void remove(E element)
```

void remove(E element)

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

void remove(E element)

- ❖ 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.

void remove(E element)

- ❖ 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.

void remove(E element)

- ❖ 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.

void remove(E element)

- Donnez la **stratégie** à haut niveau

void remove(E element)

- Donnez la **stratégie à haut niveau**
 - **Traverser** la liste.

void remove(E element)

- ❖ Donnez la **stratégie à haut niveau**
 - ❖ **Traverser** la liste.
 - ❖ **Trouver** l'élément.

void remove(E element)

- ❖ Donnez la **stratégie à haut niveau**
 - ❖ **Traverser** la liste.
 - ❖ **Trouver** l'élément.
 - ❖ **Retirer** l'élément.

public void remove(E element)

- ❖ Quelles seront les **difficultés** ?

public void remove(E element)

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

public void remove(E element)

- ❖ Quelles seront les **difficultés** ?
 - ❖ On se rappellera que lors du parcours d'une liste simplement chaînée à l'aide d'une boucle **while**, nous devons nous arrêter une position avant l'élément à retirer, puisque c'est la variable **next** de l'élément qui précède qu'il faut changer.
 - ❖ Pour retirer le premier élément, il faut modifier la variable **head** de l'entête, et non la variable **next** du noeud qui précède.

public void remove(E element)

- ✦ Quelles sont les **préconditions** ?

public void remove(E element)

- ❖ Quelles sont les **préconditions**?
 - ❖ `element != null`

public void remove(E element)

- ❖ Quelles sont les **préconditions** ?
 - ❖ **element != null**
 - ❖ La liste n'est pas vide.

public void remove(E element)

- ❖ Quelles sont les **préconditions** ?
 - ❖ **element != null**
 - ❖ La liste n'est pas vide.
- ❖ Quels sont les **cas spéciaux** ?

public void remove(E element)

- ❖ Quelles sont les **préconditions**?
 - ❖ **element != null**
 - ❖ La liste n'est pas vide.
- ❖ Quels sont les **cas spéciaux**?
 - ❖ L'élément recherché est en **première position**.

public void remove(E element)

```
public void remove(E element) {  
  
    if (element == null) {  
        throw new NullPointerException("Illegal argument");  
    }  
  
    if (head == null) {  
        throw new NoSuchElementException();  
    }  
  
    if (head.value.equals(element) ) {  
        head = head.next;  
    } else {  
        remove(head, element);  
    }  
  
}
```

Remarques

- ✦ Pour le premier appel à la méthode `remove(Node<E> current, E element)`, nous savons que `current.value.equals(element)` est faux. Pourquoi ?

Remarques

- ✚ Pour le premier appel à la méthode `remove(Node<E> current, E element)`, nous savons que `current.value.equals(element)` est faux. Pourquoi ?
 - ✚ C'est le premier appel, `current == head`.

Remarques

- ❖ Pour le premier appel à la méthode `remove(Node<E> current, E element)`, nous savons que `current.value.equals(element)` est faux. **Pourquoi ?**
 - ❖ C'est le premier appel, `current == head`.
 - ❖ Si `head.value.equals(element)` était vrai au moment de l'appel à la méthode `public`, alors il n'y aurait pas eu d'appel à la méthode `private`.

Remarques

- ❖ Pour le premier appel à la méthode **remove(Node<E> current, E element)**, nous savons que **current.value.equals(element)** est faux. **Pourquoi ?**
 - ❖ C'est le premier appel, **current == head**.
 - ❖ Si **head.value.equals(element)** était vrai au moment de l'appel à la méthode **public**, alors il n'y aurait pas eu d'appel à la méthode **private**.
- ❖ La méthode récursive préservera cette propriété, elle vérifie si l'élément recherché, **element**, se trouve à la position qui suit, **current.next**, et si oui retire ce noeud et termine, sinon elle poursuit sa recherche.

remove(Node<E> current, E element)

Cas général : Quel **scénario** semble le mieux **approprié** :

remove(Node<E> current, E element)

Cas général : Quel **scénario** semble le mieux **approprié** :

1. Faire un **appel récursif**, suivi d'un **post-traitement** ?

remove(Node<E> current, E element)

Cas général : Quel **scénario** semble le mieux **approprié** :

1. Faire un **appel récursif**, suivi d'un **post-traitement** ?
2. Faire un **pré-traitement**, suivi d'un **appel récursif** (si nécessaire) ?

remove(Node<E> current, E element)

Cas général : Quel **scénario** semble le mieux **approprié** :

1. Faire un **appel récursif**, suivi d'un **post-traitement** ?
2. Faire un **pré-traitement**, suivi d'un **appel récursif** (si nécessaire) ?

remove(Node<E> current, E element)

Cas général : Quel **scénario** semble le mieux **approprié** :

1. Faire un **appel récursif**, suivi d'un **post-traitement** ?
2. Faire un **pré-traitement**, suivi d'un **appel récursif** (si nécessaire) ?

Puisque nous devons retirer l'**élément le plus à gauche**, il faut faire un **pré-traitement** suivi d'un appel récursif (si nécessaire) ? (Stratégie 2)

remove(Node<E> current, E element)

- ❖ Quel est le **prétraitement** nécessaire ?
 - ❖ Si **current.next.value.equals(element)** on retire le prochain élément.
 - ❖ Sinon, il faut traiter le reste de la liste (**appel récursif**).

remove(Node<E> current, E element)

- ❖ Quel est le **cas de base** ?
 - ❖ **Singleton.**
 - ❖ Que fait-on ?
 - ❖ On lance l'exception **NoSuchElementException**.

remove(Node<E> current, E element)

```
private void remove(Node<E> current , E element) {  
  
    if ( current.next == null ) {  
        throw new NoSuchElementException();  
    }  
  
    if ( current.next.value.equals(element) ) {  
        current.next = current.next.next; // base case  
    } else {  
        remove(current.next , element); // general case  
    }  
}
```

```
public void remove(E element) {
    if (element == null) {
        throw new NullPointerException("Illegal argument");
    }
    if (head == null) {
        throw new NoSuchElementException();
    }
    if (head.value.equals(element) ) {
        head = head.next; // special case
    } else {
        remove(head, element);
    }
}
```

```
private void remove(Node<E> current, E element) {
    if (current.next == null ) {
        throw new NoSuchElementException();
    }
    if (current.next.value.equals(element)) {
        current.next = current.next.next; // base case
    } else {
        remove(current.next, element); // general case
    }
}
```

Exercices

- ❖ `void removeLast()`
- ❖ `void removeLast(E element)`
- ❖ `void removeAll(E element)`
- ❖ `void remove(int pos)`

Implémentation

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

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

- ❖ La méthode retournera un **nouvelle liste** contenant les éléments situés entre les positions **fromIndex** et **toIndex** de la liste originale, sans la changée.

Discussion

Proposez une stratégie afin de construire le résultat.

Discussion

Proposez une stratégie afin de construire le résultat.

1. **Post-treatment**

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-treatment

- ✚ **Traverser** la liste jusqu'à l'index le plus élevé ;

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ **Traverser** la liste jusqu'à l'index le plus élevé ;
- ❖ **Retourner** une liste contenant seulement la valeur se trouvant à cette position ;

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ **Traverser** la liste jusqu'à l'index le plus élevé ;
- ❖ **Retourner** une liste contenant seulement la valeur se trouvant à cette position ;
- ❖ **Ajouter** l'élément courant au **début** de la liste, si sa position fait partie de l'intervalle.

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ **Traverser** la liste jusqu'à l'index le plus élevé ;
- ❖ **Retourner** une liste contenant seulement la valeur se trouvant à cette position ;
- ❖ **Ajouter** l'élément courant au **début** de la liste, si sa position fait partie de l'intervalle.

2. Pré-traitement

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ **Traverser** la liste jusqu'à l'index le plus élevé ;
- ❖ **Retourner** une liste contenant seulement la valeur se trouvant à cette position ;
- ❖ **Ajouter** l'élément courant au **début** de la liste, si sa position fait partie de l'intervalle.

2. Pré-traitement

- ❖ Une **liste vide** est passée en **paramètre** du premier appel ;

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ Traverser la liste jusqu'à l'index le plus élevé ;
- ❖ Retourner une liste contenant seulement la valeur se trouvant à cette position ;
- ❖ Ajouter l'élément courant au **début** de la liste, si sa position fait partie de l'intervalle.

2. Pré-traitement

- ❖ Une **liste vide** est passée en **paramètre** du premier appel ;
- ❖ **Ajouter** l'élément courant à la **fin** de la liste, si la position courante fait partie de l'intervalle ;

Discussion

Proposez une stratégie afin de construire le résultat.

1. Post-traitement

- ❖ Traverser la liste jusqu'à l'index le plus élevé ;
- ❖ Retourner une liste contenant seulement la valeur se trouvant à cette position ;
- ❖ Ajouter l'élément courant au **début** de la liste, si sa position fait partie de l'intervalle.

2. Pré-traitement

- ❖ Une **liste vide** est passée en **paramètre** du premier appel ;
- ❖ **Ajouter** l'élément courant à la **fin** de la liste, si la position courante fait partie de l'intervalle ;
- ❖ **Appel** récursif.

Stratégie 1

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

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(current.value);  
}
```

Stratégie 1

❖ Cas général :

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

Stratégie 1

❖ Cas général :

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

❖ Que contient **result** ?

Stratégie 1

❖ Cas général :

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

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

Stratégie 1

❖ Cas général :

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

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

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

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

private LinkedList<E> subList(Node<E> current, int index, int fromIndex, int toIndex) {

    LinkedList<E> result;

    if (index == toIndex) {

        result = new LinkedList<E>();
        result.addFirst(current.value);

    } else {

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

        if (index >= fromIndex) {
            result.addFirst(current.value);
        }

    }

    return result;
}
```

- ❖ Le traitement des **préconditions** (intervalle de valeurs illégales) 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(head, 0, result, fromIndex, toIndex);  
    return result;  
}
```

Stratégie 2

❖ Cas de base :

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

Stratégie 2

❖ Cas de base :

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

`result.addLast(current.value)` ou `result.addFirst(current.value)` ?

Stratégie 2

❖ Cas général :

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

Stratégie 2

❖ Cas général :

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

`result.addLast(current.value)` ou `result.addFirst(current.value)` ?

```
public LinkedList<E> subList(int fromIndex, int toIndex) {  
    LinkedList result = new LinkedList<E>();  
    subList(head, 0, result, fromIndex, toIndex);  
    return result;  
}  
  
private void subList(Node<E> current, int index, LinkedList<E> result,  
                    int fromIndex, int toIndex) {  
    if (index == toIndex ) {  
        result.addLast(current.value);  
    } else {  
        if (index >= fromIndex) {  
            result.addLast(current.value);  
        }  
        subList(current.next, index+1, result, fromIndex, toIndex);  
    }  
}
```

Principes

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node, current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node, current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`
 - ❖ `current.next == null`

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`
 - ❖ `current.next == null`
 - ❖ `current.value.equals(element)`

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`
 - ❖ `current.next == null`
 - ❖ `current.value.equals(element)`
 - ❖ `current.next.value.equals(element)`

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`
 - ❖ `current.next == null`
 - ❖ `current.value.equals(element)`
 - ❖ `current.next.value.equals(element)`
 - ❖ Cas général :

Principes

Les **paramètres** jouent un **rôle essentiel** pour l'écriture de méthodes **récurives**.

- ❖ Un paramètre de type **Node**, **current**, tient un rôle clé pour **contrôler l'exécution** de la méthode.
 - ❖ Exemples de tests pour le **cas de base** :
 - ❖ `current == null`
 - ❖ `current.next == null`
 - ❖ `current.value.equals(element)`
 - ❖ `current.next.value.equals(element)`
 - ❖ **Cas général** :
 - ❖ On passe la valeur de **current.next** en paramètre pour l'**appel récursif** afin de traiter le reste de la liste.

Principes

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

- ❖ **Un compteur** : Si chaque appel doit connaître sa position dans la liste, on passe en paramètre la valeur d'un compteur, et chaque appel passe au suivant cette valeur plus un.

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

- ❖ **Un compteur** : Si chaque appel doit connaître sa position dans la liste, on passe en paramètre la valeur d'un compteur, et chaque appel passe au suivant cette valeur plus un.
 - ❖ On pourrait aussi décrémenter la valeur à chaque appel.

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

- ❖ **Un compteur** : Si chaque appel doit connaître sa position dans la liste, on passe en paramètre la valeur d'un compteur, et chaque appel passe au suivant cette valeur plus un.
 - ❖ On pourrait aussi décrémenter la valeur à chaque appel.

2. Valeur de retour :

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

- ❖ **Un compteur** : Si chaque appel doit connaître sa position dans la liste, on passe en paramètre la valeur d'un compteur, et chaque appel passe au suivant cette valeur plus un.
 - ❖ On pourrait aussi décrémenter la valeur à chaque appel.

2. Valeur de retour :

- ❖ La méthode **size** retourne la taille de la liste à partir de l'élément désigné par son paramètre **current**.

Deux mécanismes pour **passer des informations** entre les appels.

1. Paramètres :

- ❖ **Un compteur** : Si chaque appel doit connaître sa position dans la liste, on passe en paramètre la valeur d'un compteur, et chaque appel passe au suivant cette valeur plus un.
 - ❖ On pourrait aussi décrémenter la valeur à chaque appel.

2. Valeur de retour :

- ❖ La méthode **size** retourne la taille de la liste à partir de l'élément désigné par son paramètre **current**.
 - ❖ Pour l'appelant, la taille de la liste est un de plus que la valeur retournée.

```
type method(Node<E> current) {  
  
    type result;  
  
    if (current ...) {           // base case  
        calculating the result // no recursive call  
    } else {                     // general case  
                                // pre-processing  
        s = method(current.next); // recursion  
                                // post-processing  
    }  
  
    return result;  
}
```

«*head & tail*»

Étapes :

- ▣ Que signifie `méthode(current.next)` ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?
 - ❖ Quelle est la **plus petite liste valide** ?

«*head & tail*»

Étapes :

- ❖ Que signifie **méthode(current.next)** ?
 - ❖ La solution d'un problème, **plus petit d'un élément**.
- ❖ Comment allons-nous **utiliser ce résultat** afin de construire la solution du problème pour une liste débutant par l'élément **current** ?
- ❖ Quels sont les **cas de base** ?
 - ❖ Quelle est la **plus petite liste valide** ?
 - ❖ Quel est le **résultat** ?

Prologue

- ❖ Nous avons proposé la stratégie «*head & tail*»

- ❖ Nous avons proposé la stratégie «*head & tail*»
 - ❖ **Cas de base** : généralement un test impliquant la valeur de **current**

- ❖ Nous avons proposé la stratégie «*head & tail*»
 - ❖ **Cas de base** : généralement un test impliquant la valeur de **current**
 - ❖ **Cas général** : appel récursif passant la valeur de **current.next** en paramètre

- ❖ Nous avons proposé la stratégie «*head & tail*»
 - ❖ **Cas de base** : généralement un test impliquant la valeur de **current**
 - ❖ **Cas général** : appel récursif passant la valeur de **current.next** en paramètre
- ❖ On contrôle la récursivité à l'aide des **paramètres**

- ✦ **Arbres binaires de recherche** : concept

References I



Koffman, E. B. and T., W. P. A. (2016).

Data Structures : Abstraction and Design Using Java.

John Wiley & Sons, 3e edition.



Marcel Turcotte

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (**SIGE**)
Université d'Ottawa