Algorithmique Avancée

Nicolas Labroche

Cours 1 : objectifs et organisation du cours, listes chaînées

Université de Tours

Introduction

Objectifs de l'enseignement

- comprendre et manipuler des structures de données dynamiques
- différencier l'interface d'une structure de son implémentation
- programmer des algorithmes plus complexes itératifs et récursifs
- comprendre les principaux algorithmes de tri
- prendre conscience des problématiques de complexité

Contenu de l'enseignement

6 CM, 10 TD, 12 TP

1. Rappels sur les structures de données statiques : tableaux en Java ou semi-dynamiques comme les vecteurs dynamiques (`ArrayList`)
2. Structures séquentielles simples : listes chaînées
3. Structures séquentielles complexes : listes doubles, circulaires, piles, files,
4. Arbres binaires
5. Arbres binaires de recherche et arbres généraux
6. Tri et complexité $\Rightarrow$ préparation à la L3

Comment aborder le cours ? Petite FAQ

- *À quoi sert le cours d'AAV ?*
  Apprendre à abstraire une solution implémentable à partir d'un raisonnement en utilisant des structures dynamiques (listes, arbres). Prendre conscience que toute opération a un coût.

- *Pourquoi en Java ?*
  Car ici le langage n'est pas l'objectif mais un moyen pour travailler sur les structures dynamiques sans se préoccuper d'apprendre de nouveaux mécanismes propres à un nouveau language. Cela permettra aussi de renforcer vos connaissances en programmation objet (légèrement, ce n'est pas l'objectif premier).

Comment aborder le cours ? Petite FAQ

- *Est-ce qu'il y a des limites à utiliser Java ?*
  Oui : Java n'est pas un langage fonctionnel, il n'est donc pas aisé de passer une méthode en argument d'une méthode. Nous verrons comment procéder pour certains aspects fonctionnel avec les interfaces `Function`, `BiFunction` et les lambda expressions de Java 8.

Méthodologie de travail

Le cours :

- accessible sur l'ENT
- doit être relu (compris) d'une semaine sur l'autre
- réponse aux questions en début de séance suivante
- ne pas hésiter à prendre des notes complémentaires (schémas)

Méthodologie de travail

Les TD :

- accessibles sur l'ENT
- discussions possibles avec ses voisins à voix basse
- en profiter pour poser des questions
- à finir chez soi d'une semaine sur l'autre
- commencer par réfléchir sur papier avec un schéma
- implémenter les exercices sur machine une fois terminés "sur papier"

Méthodologie de travail

Les TP :

- accessibles sur l'ENT
- travail en monôme de préférence (selon le nombre d'ordinateurs disponibles)
- spécifier toutes les fonctions que vous écrivez
- commenter le code abondamment
- tester : programmer pour chaque fonction une méthode de test sans argument
- une séance de TP sera évaluée (voir après)

Évaluation

Note finale = 60% examen final + 40% CC

- Examen : exercices couvrant l'ensemble du programme
  - seul document autorisé : feuille A4 recto-verso
  - rattrapage selon les mêmes modalités
- CC : contrôle continu
  - 2 interrogations en TD
  - 1 TP noté sur les algorithmes de tri (à finir à la maison)

Organisation et groupes

Cette année : 2 groupes de TD, 3 groupes de TP

CM	Nicolas Labroche (resp.)
TD et TP	Nicolas Labroche
	Ben Crulis
TP	Fodil Benali

Calendrier

Semaine		CM	Heure	Instr.	TD	Heure	Instr.	TP machine	Heure	Instr.	TOTAL
			9			15			18		
03/01/2022	1	Intro, listes chaînées Java	1,5	NL							1,5
10/01/2022	2	Dictionnaire, tables de hachage, tuples	1,5	NL	Listes chaînées	1,5	NL + BC	Lises chaînées	1,5	NL + BC + FB	4,5
17/01/2022	3				Listes chaînées	1,5	NL + BC	Listes chaînées	1,5	NL + BC + FB	3
24/01/2022	4	EGC 2022 Conference									
31/01/2022	5	Liste doublement chaînées et circulaires	1,5	NL	Dictionnaire, tables de hachage, tuples	1,5	NL + BC	Dictionnaire, tables de hachage, tuples	1,5	NL + BC + FB	3
07/02/2022	6				Dictionnaire, tables de hachage, tuples	1,5	NL + BC	Dictionnaire, tables de hachage, tuples	1,5	NL + BC + FB	4,5
14/02/2022		PAUSE									
21/02/2022	7	Map / Filter / Reduce	1,5	NL	Listes doublement chaînées, pile, file	1,5	NL + BC	Listes doublement chaînées, pile, file	1,5	NL + BC + FB	4,5
28/02/2022	8	EDBT 2022 Conference									
07/03/2022	9	Arbres binaires et arbres binaires de recherche	1,5	NL	Parcours AB	1,5	NL + BC	Parcours AB	1,5	NL + BC + FB	4,5
14/03/2022	10	Arbres généraux	1,5	NL	Arbres binaires de recherche	1,5	NL + BC	Arbres binaires de recherche	1,5	NL + BC + FB	4,5
21/03/2022	11				ressources	1,5	NL + BC	Arbres généraux	1,5	NL + BC + FB	3
28/03/2022	12				Arbres généraux	1,5	NL + BC	Arbres généraux	1,5	NL + BC + FB	3
04/04/2022	13					1,5	NL + BC	Algorithmes de tri	1,5	NL + BC + FB	3
11/04/2022	14							Algorithmes de tri	3	NL + BC + FB	3

Prérequis

Prérequis

Pour bien comprendre ce cours il est nécessaire de connaître :

- les notions étudiées en L1 :
  - les types de données et les structures de contrôle
  - les structures statiques comme les tableaux
  - les appels de fonction et avoir des notions de récursivité
  - en L1 ces notions sont présentées par le biais du langage Java
- quelques notions vues en L2 :
  - la programmation objet simple avec encapsulation et constructeur
  - les types génériques `<T>`
  - la notion d'adresse mémoire pour accéder à un objet
  - la notion d'interface pour les barrières d'abstraction

Introduction aux structures dynamiques

Les structures de données statiques

Vous utilisez des structures de données statiques depuis la L1 !

- tableaux 1D et 2D

```
int[] tab = new int[4];
String[][] occurrences = new String[3][4];
```

- représentation de données dont la taille est connue à l'avance
  - par exemple pour représenter des vecteurs tableaux, matrices

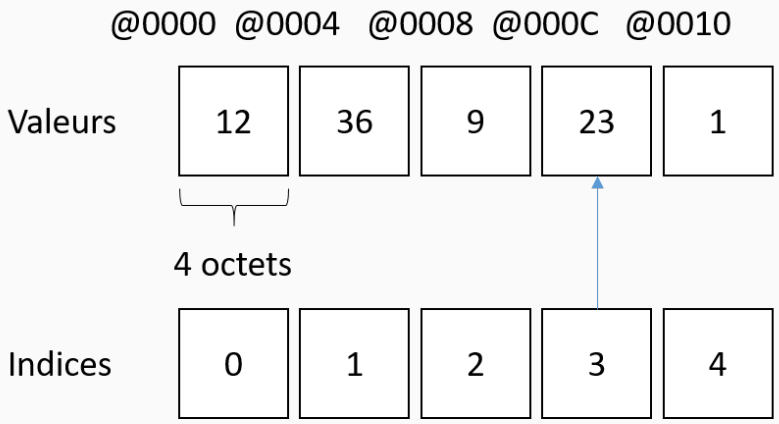
E_1	E_2	E_3	E_4

E_{11}	E_{12}	E_{13}	E_{14}
E_{21}	E_{22}	E_{23}	E_{24}
E_{31}	E_{32}	E_{33}	E_{34}

Avantages des tableaux en Java

- lorsqu'un tableau est créé en Java : toutes les valeurs sont placées côte à côte en mémoire

```
int[] t = {12, 36, 9, 23, 1};
```

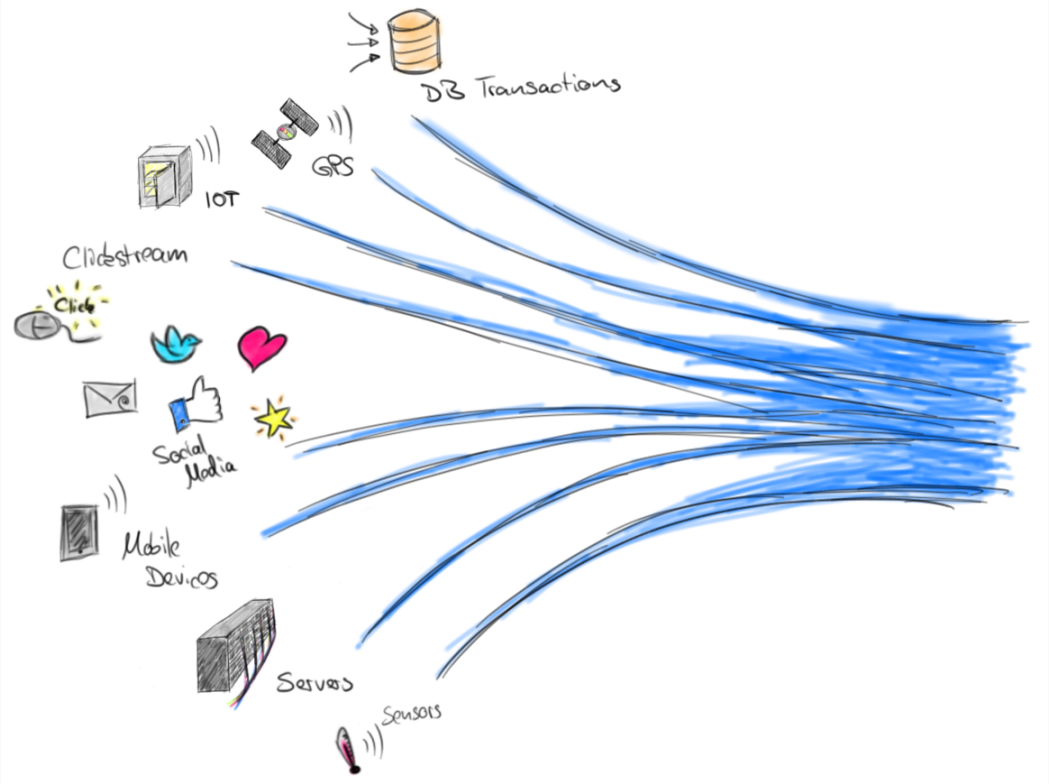


L'élément t[3] est à l'adresse
@t[3] = @t[0] + 3 * taille élément t
@t[3] = 0000 + 3 * 4 = 000C
(en héxadécimal C = 12)

- les tableaux sont très efficaces pour accéder non séquentiellement à une information en lecture ou écriture

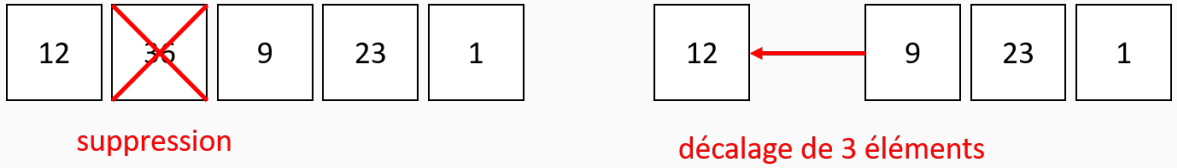
Limite des structures de données statiques

- Dans la pratique, besoin de représenter des objets de taille inconnue initialement ou de taille variable au cours du temps
- Problématiques Volume et Velocity des Big Data
  (illustration de Apache Flink)

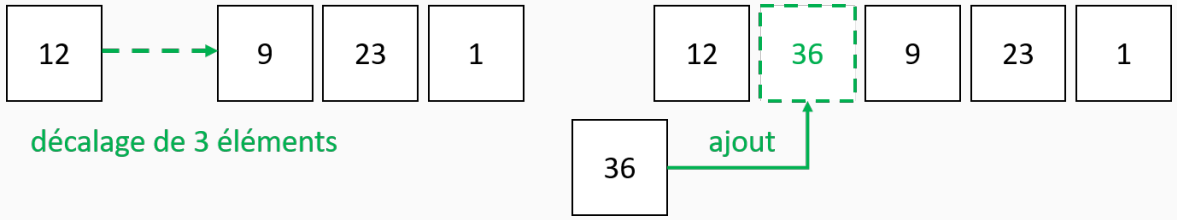


Limite des tableaux

- les limites concernent les insertions ou suppressions d'éléments
- par exemple, dans le cas de la suppression d'un élément



- par exemple, dans le cas de l'ajout d'un élément



- problème encore plus important s'il faut redimensionner le tableau

Solutions possibles avec un tableau

1. utiliser un tableau surdimensionné
   - ne garantit pas qu'il soit suffisamment grand
   - consommation inutile de ressources (mémoire)
2. reconstruire le tableau dynamiquement
   - quand la taille maximale est atteinte
   - construire un tableau plus grand
   - recopier tous les éléments dans le nouveau tableau
   - c'est le principe des vecteurs dynamiques ou `ArrayList` en Java
3. utiliser une structure dynamique
   - taille croissante au fur et à mesure de la lecture des valeurs

Principe des structures de données dynamiques

Structure de donnée dynamique
$\Rightarrow$ qui peut évoluer pour s'adapter à la représentation de ses objets

- Nécessité de suivre les évolutions de la structure
  - attribuer de la place en mémoire quand elle grandit
  - récupérer la place en mémoire quand elle diminue

- Mise en œuvre
  1. allocation et libération dynamique d'espace mémoire
  2. structure auto-référentielle / récursive

Allocation dynamique et pointeur

- Les procédures de **réservation** et de **libération** de mémoire travaillent sur des pointeurs

- Un pointeur = variable dont le contenu indique l'emplacement mémoire (l'adresse) d'une autre variable

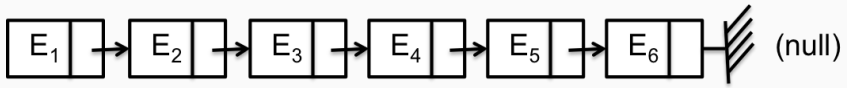
Mécanisme de réservation et de libération mémoire

- En C (méthode traditionnelle) :
  - réservation explicite avec nombre d'octets nécessaires : `malloc()`
  - libération explicite : `free()`
- En Java (méthode actuelle) :
  - opérateur `new` pour réserver l'espace mémoire
  - les objets sont contenus dans une zone mémoire appelé heap (tas)
  - les variables objets contiennent l'adresse de l'objet dans le tas
    - il s'agit de pointeurs implicites !
  - la libération mémoire est effectuée par le `garbage collector`
- Une variable pointeur `p` qui ne pointe sur rien : `p = null`
  - `null` est une valeur particulière d'adresse qui indique qu'il n'y a pas d'adresse

Structures auto-référentielles ou récursives

Structure dont au moins un des champs contient un pointeur vers une structure de même type

- Création d'éléments (appelés nœud ou lien) contenant des données reliées entre elles par des liens logiques / pointeurs



- Possibilité d'insérer ou de supprimer dynamiquement des éléments (voir après)
- Contrairement à un tableau, les données peuvent être éparpillées en mémoire

Les collections en Java

Implémentation Java des structures dynamiques

- Depuis le JDK 5.0, Java contient l'interface `Collection<E>`
- Une interface regroupe des méthodes communes à plusieurs classes
- `E` désigne le type des éléments contenus dans les collections
  - par exemple : `Collection<Integer>`, `Collection<String>`
- Une interface n'est pas un objet = pas instanciable !
- Pour utiliser une collection, il faut instancier un objet qui implémente l'interface `Collection<E>`

Classes implémentant l'interface `Collection<E>`

- les vecteurs dynamiques : `ArrayList` et `Vector`
- les listes : `LinkedList`
- les ensembles : `HashSet` et `TreeSet`
- les queues (file) : avec priorité `PriorityQueue` et à double entrée `ArrayDeque`

Pour représenter les listes, il y a deux possibilités :

- les `ArrayList`
- les `LinkedList`

Parcours d'une `Collection<E>`

Utilisation des `Iterator<E>` unidirectionnels :

```
Iterator<E> iter = c.iterator(); // collection c
while (iter.hasNext()) {
  E obj = iter.next(); // objet courant
}
```

Parcours en utilisant une boucle `for each` :

```
for (E obj : c) {...}
```

$\Rightarrow$ mais solution inexploitable si modification de la collection par les méthodes `remove` et `add`

Ajout et suppression d'un élément dans une `Collection<E>` (1)

- Méthodes propres à la collection

```
// Ajout en fin pour les ArrayList et LinkedList
c.add(elem);
// Ajout de tous les elements de la collection col
c.addAll(col);

// Suppression de l'element
c.remove(elem);
// Suppression de tous les elements de col
c.removeAll(col);
```

Ajout et suppression d'un élément dans une `Collection<E>` (2)

- Méthodes issues des itérateurs

```
Iterator<E> iter = c.iterator();
ListIterator<E> iter2 = c.iterator(); // pour ArrayList↩
    et LinkedList
```

- Ajout uniquement pour les `ListIterator<E>`
  - avant la position de l'élément qui devrait être retourné par `next()`

```
iter2.add(elem);
```

- Suppression

```
iter.remove(elem);
iter2.remove(elem);
```

Les tableaux dynamiques `ArrayList<E>`

- Fonctionnalités d'accès rapide à un élément comme un tableau
  - utilisation de plages mémoires contigues
- Plus souple qu'un tableau car possibilité d'ajouter des éléments
- Mais insertion / supression à la position courante coûteuse du fait de la structure en mémoire
- Ajout et suppression issues de la `Collection`

Les tableaux dynamiques `ArrayList<E>`

```
ArrayList<String> mots = new ArrayList();
mots.add("Hello"); mots.remove("world");
mots.add(3, "Holiday"); // ajout indice 3
```

- Suppression d'une plage définie par des indices de début et de fin

```
mots.removeRange(3,8);
```

Les listes chaînées `LinkedList<E>`

- Représentation et manipulation de listes (doublement !) chaînées
  - références vers les éléments ‘suivant’ et ‘précédent’
- Utilisation possible de l’itérateur `ListIterator`
  - insertion à la position courante en $O(1)$ avec `iter.add(elem);`
  - idem pour la suppression avec `iter.remove(elem);`
- Insertion / suppression en début et fin en $O(1)$ avec les méthodes de `LinkedList<E>`

```
LinkedList<Integer> l = new LinkedList();
l.addFirst(12); l.removeFirst(12);
l.addLast(13); l.removeLast(12);
```

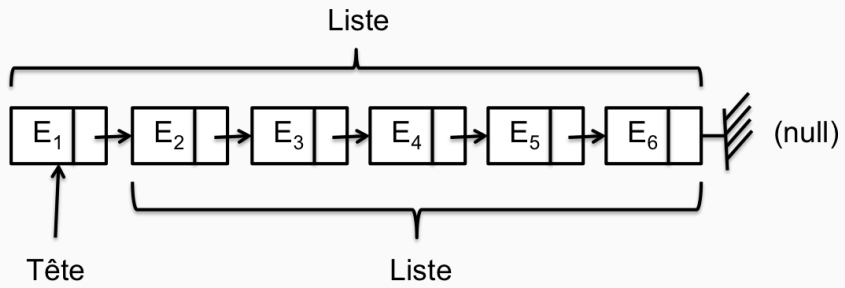
- Méthodes `add` et `remove` issues de la `Collection` peu efficaces

Listes chaînées

Définition d'une liste

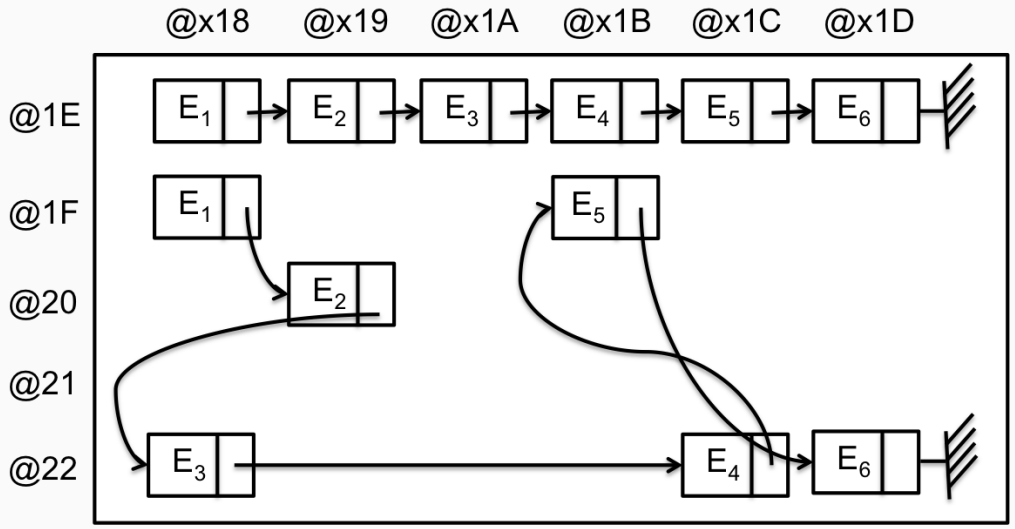
- Une liste est une structure de données qui regroupe une séquence d'éléments de même type
- Une liste est une collection homogène ordonnée
- Un élément peut figurer plusieurs fois dans une liste ($\neq$ ensemble)

- Une liste est une structure récursive
  - une liste = un élément appelé tête + la suite de la liste



Représentation mémoire d'une liste

- Une liste chaînée est une suite d'éléments, dans laquelle les éléments sont rangés linéairement
- Linéarité purement virtuelle : à la différence du tableau, les éléments n'ont aucune raison d'être contigus ni ordonnés en mémoire



Représentation en Java d'une liste chaînée

Dans ce cours, nous considérons une **barrière d'abstraction** pour les listes chaînées :

- ensemble des méthodes et fonctionnalités attachées aux listes
- la barrière d'abstraction précise le **quoi** et pas le **comment**
- avantage : indépendance des traitements demandés de la manière dont ils sont réellement implémentés algorithmiquement
- inconvénient : problème de lisibilité, parfois on croit utiliser une liste chaînée et finalement on utilise un tableau amélioré (cf. `ArrayList` en Java)

$\Rightarrow$ utilisation d'une interface en Java pour représenter la barrière d'abstraction des listes

Opérations possibles sur les listes chaînées

Les opérations minimales effectuées sur une liste chaînée

- le parcours de la liste (affichage, calcul de la longueur)
- l'insertion en tête de liste, en fin de liste
- la lecture / modification d'un élément
- la suppression d'un élément
- la recherche d'un élément

Barrière d'abstraction d'une liste chaînée

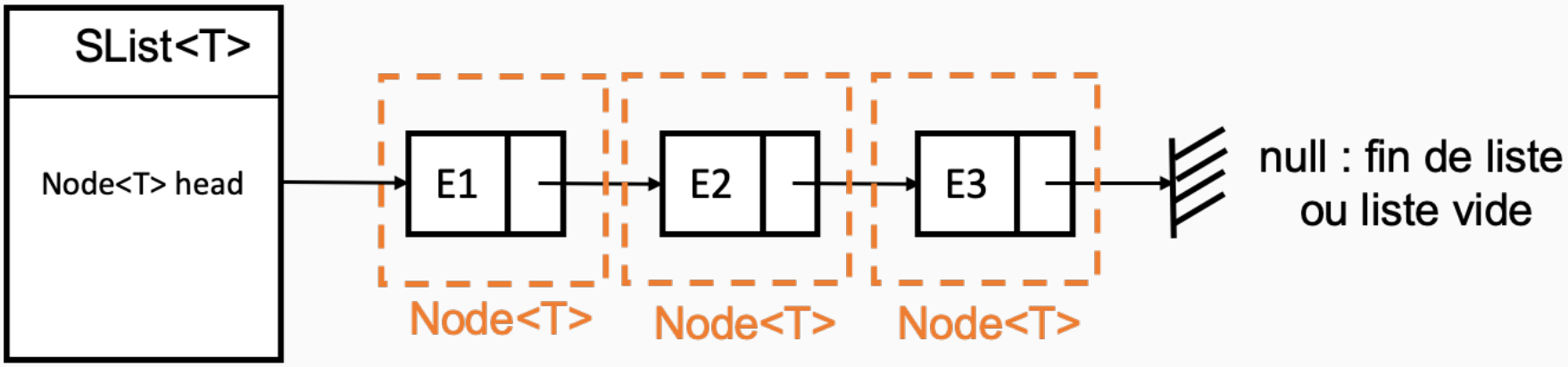
```
public interface LList<T>
{
    void add(T elem);                    // ajout en fin de liste
    void add(T elem, int index);         // ajout a un index
    void clear();                        // vider la liste
    boolean contains(T elem);            // contient un element ?
    boolean isEmpty();                   // liste vide ?
    int indexOf(T elem);                 // indice element
    void remove(int index);              // supprimer position index
    T get(int index);                    // valeur position index
    void set(T elem, int index);         // definir position index
    int size();                          // taille liste
    String toString();                   // chaine de caracteres
}
```

Représentation en Java d'une liste chaînée (2)

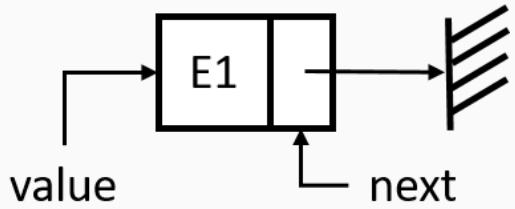
Nous proposons dans ce cours, une représentation à deux niveaux d'une liste chaînée

- un objet `Node<T>` qui suit la définition récursive d'un élément d'une liste chaînée
  - un *Node* est soit vide (= null), soit un élément de type `T` suivi d'un autre nœud *Node*
- un objet `SList<T>` pour *Simple Linked List* qui représente la liste chaînée
  - une liste possède une tête de liste qui est un `Node<T>`
  - et toutes les opérations que l'on peut effectuer dessus
    - ajout, suppression, recherche d'un élément
    - affichage
    - tri, comparaison de listes
    - extraction de sous-listes

Représentation en Java d'une liste chaînée (3)



Représentation d'un nœud en Java




```
public static class Node<T> {
    T value;
    Node<T> next;

    public Node(T elem){
        this.value = elem;
        this.next = null;
    }

    public Node(T elem, Node<T> _next){
        this.value = elem;
        this.next = _next;
    }
}
```

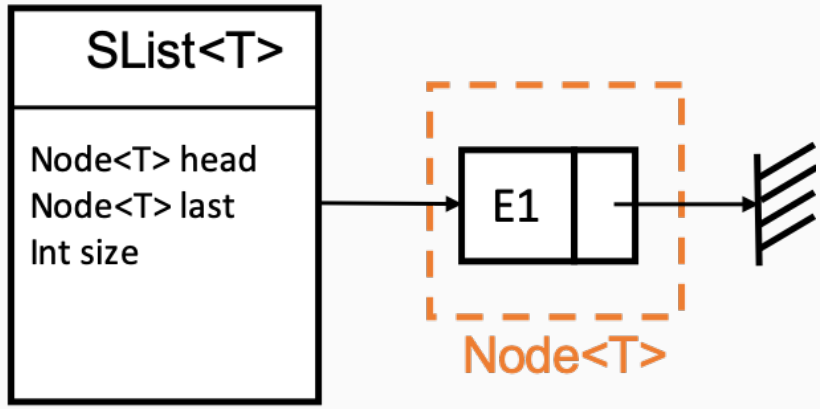
Quels choix d'implémentation pour le Node ?

- Classe `public Node<T>`
  - permet de réutiliser `Node<T>` dans les classes dérivées de `SList<T>`
  - permet aussi d'utiliser des classes dérivées de `Node<T>`
- Classe `static inner Node<T>`
  - classe intégrée au code d'une autre classe qui n'a de sens que dans le contexte de l'autre classe (`Node` serait alors indissociable de `List`)
  - se comporte comme une classe normale, sauf qu'elle apparaît dans l'espace de nommage de sa classe encadrante
  - comme elle est `static` : pas besoin de créer une instance de la classe encadrante, mais ne peut accéder qu'aux champs `static` de la classe encadrante

Construction d'une liste `SList<T>` en Java

- plusieurs constructeurs : liste vide ou avec un seul élément comme en Java

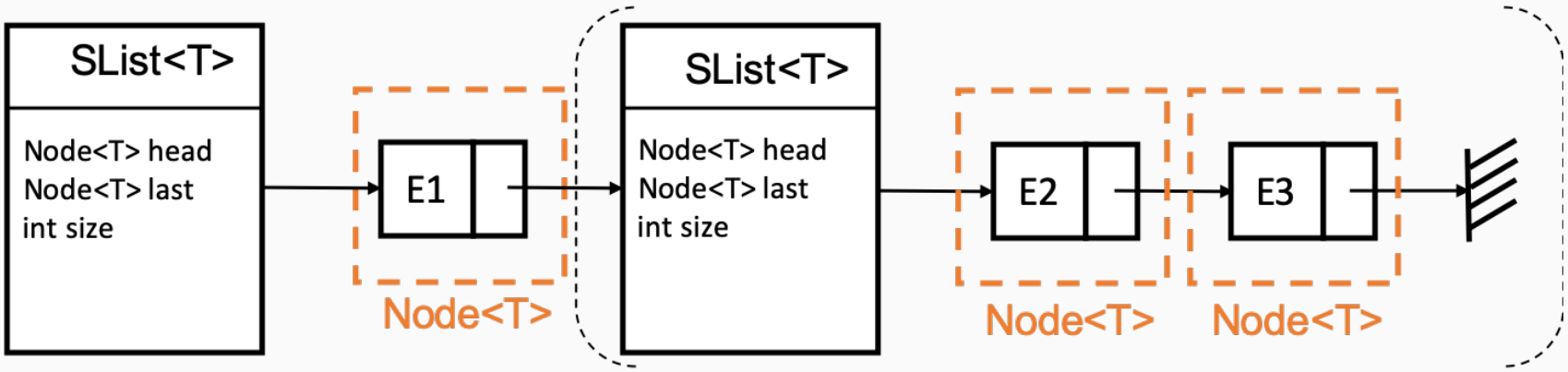
```
public class SList<T> ↩
    implements LList<T> {

    protected Node<T> head;
    protected Node<T> last;
    protected int size;

    /**
     * Create an empty list
     */
    public SList(){
        head = null;
        last = null;
        size = 0;
    }
```



Construction d'une liste `SList<T>` en Java (2)

- ou constructeur de philosophie plus récursive à partir d'une liste existante et d'une valeur

```
// liste a partir d'une tete de liste et d'une queue de liste
public SList(T head, SList<T> tail) {
    this.head = new Node<T>(head, tail.head) ;
    this.size = 1 + tail.size;
    this.last = tail.last;
}
```



Construction d'une liste `SList<T>` en Java (3)

On peut réécrire la méthode précédente en parcourant explicitement la liste `tail` pour mettre à jour le pointeur `last` et la taille `size` de la liste.

```
public SList(T head, SList<T> tail) {
    this .head = new Node<T>(head, tail.head) ;
    Node<T> p = this.head; // sauvegarde debut liste head
    size = 1; // au moins 1 element dans la liste
    while (p.next != null){
        p = p.next;
        size ++;
    }
    // on s'arrete sur le dernier element
    last = p;
}
```

Parcours d'une liste

- Les éléments ne sont pas indexés dans une liste
  - pas d'accès direct à un élément
- Chaque élément est lié à son successeur sauf le dernier
- Accès à un élément uniquement en passant par le premier élément et en parcourant tous les éléments jusqu'à atteindre celui qui est recherché

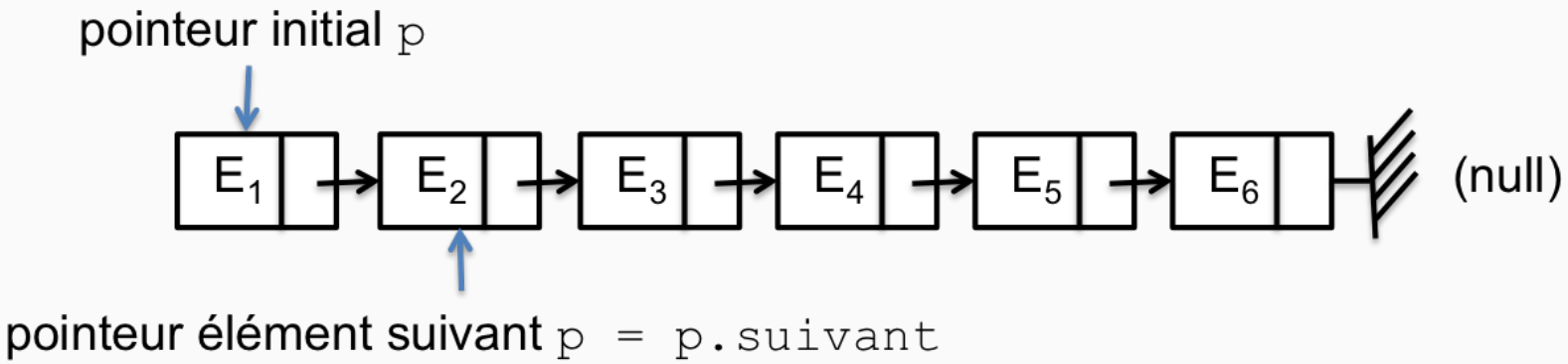
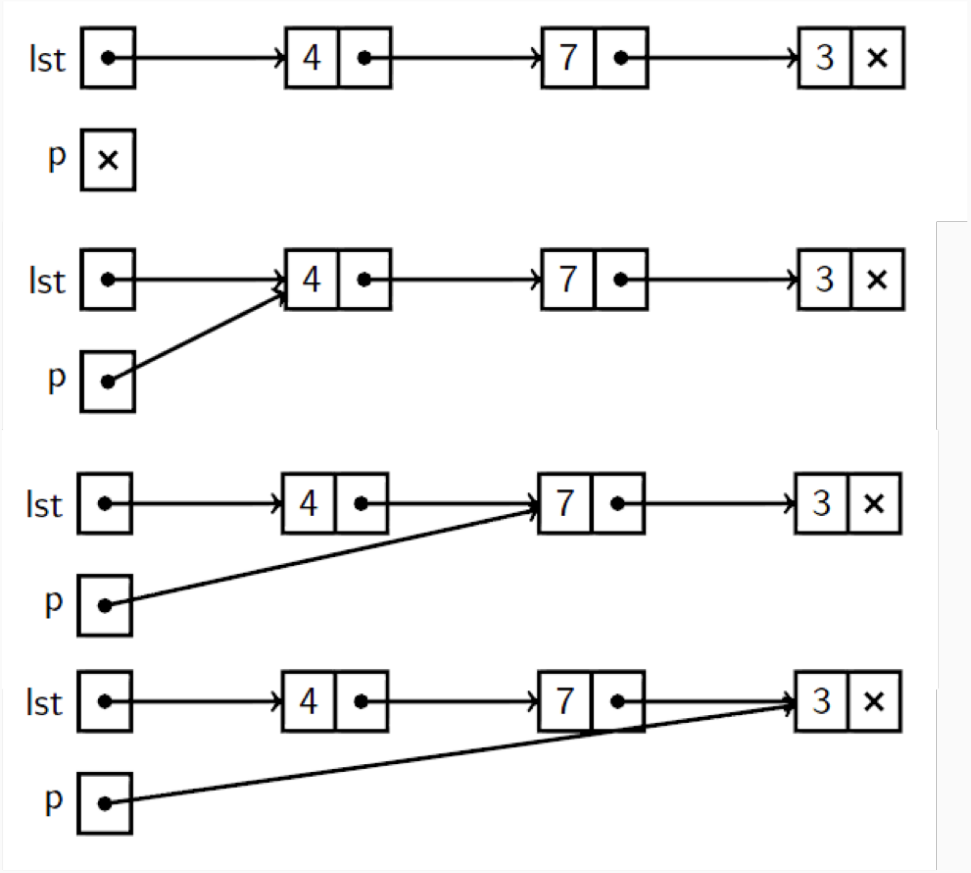


Illustration du parcours d'une liste

- on utilise un pointeur temporaire **p** qui indique la position courante dans la liste **lst**



$\Rightarrow$ affiche `4, 7, 3`

Affichage des éléments d'une liste

- construire un pointeur temporaire **p** et le placer en début de liste (= `head`)
- tant que la liste n'est pas vide : afficher la valeur et décaler le pointeur p vers le suivant
- implémentations itératives / récursives possibles

```
public String toString() {
    StringBuffer sb = new StringBuffer("(");
    Node<T> p = head;
    while (p != null){
        sb.append(p.value);
        if (p.next != null) sb.append(",");
        p = p.next;
    }
    sb.append(")");
    return sb.toString();
}
```

Longueur d'une liste

- première solution : utiliser un champs `size` pour mémoriser le nombre d'élément

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

- comme précédemment, plusieurs parcours possibles, dans une classe ou l'autre, itératif ou récursif

Longueur d'une liste - parcours itératif

```
public int size() {
    if (head == null) return 0;
    else {
        int i = 1;
        Node<T> p = head;
        while (p.next != null) {
            p = p.next;
            i++;
        }
        return i;
    }
}
```

Longueur d'une liste - parcours récursif

On utilise la relation de récurrence suivante :

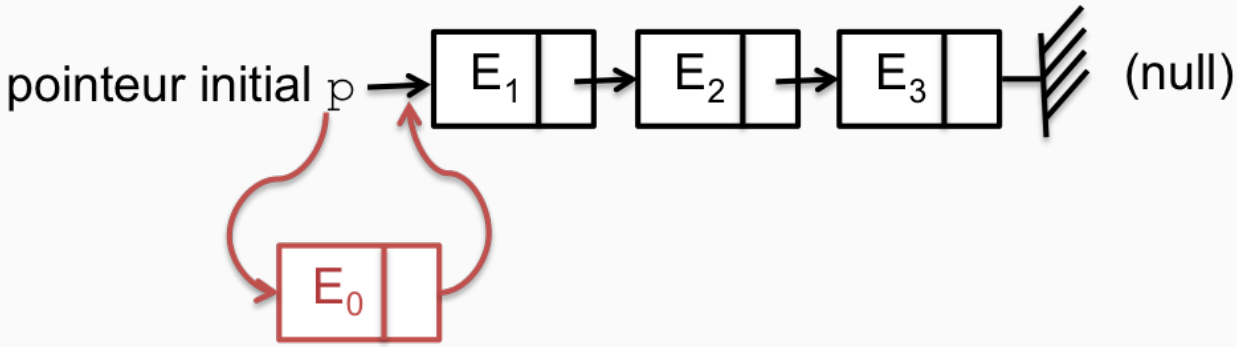
- si la liste est vide, la longueur = 0
- sinon la longueur = 1 (la tête) + la longueur de la suite de la liste

```
private int nbNodes(Node<T> node) {
  if (node == null) return 0;
  else return 1 + nbNodes(node.next);
}

public int sizeRec() {
  return this.nbNodes(head);
}
```

- Attention ici la méthode prend en argument le `Node<T>` qui est une structure récursive

Insertion d'un élément en tête de liste

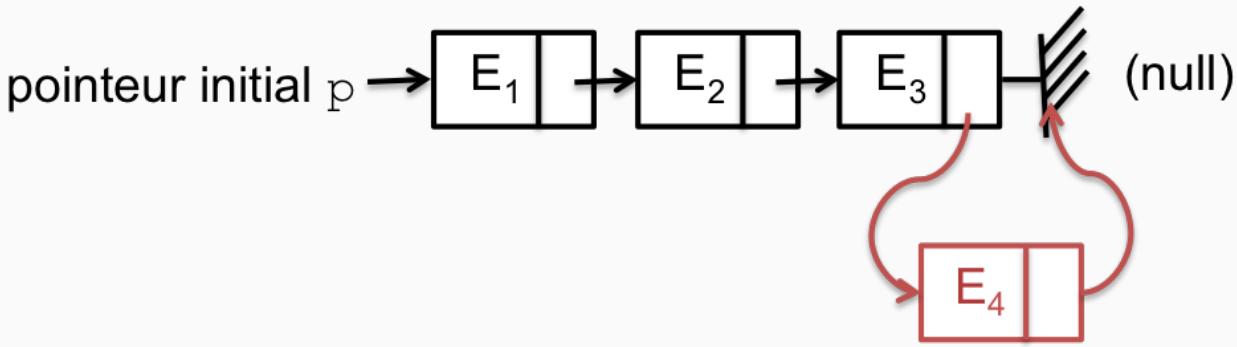
- On utilise simplement le constructeur de notre classe `Node<T>`
- Pas de mise à jour du pointeur `last` dans ce cas




```
public void push(T elem) {
  this.head = new Node<T>(elem, this.head);
  size ++;
}
```

Insertion d'un élément en fin de liste

- En itératif, remplacer le suivant lorsqu'il est égal à `null`
- Ici ajouter suite au pointeur `last` : on ne fait qu'une opération environ au lieu de parcourir toute la liste



Insertion d'un élément en fin de liste

```
public void add(T elem) {
    if (isEmpty()){
        this.head = new Node<T>(elem);
        this.last = head;
        size = 1;
    } else {
        last.next = new Node<T>(elem);
        last = last.next;
        size +=1;
    }
}
```

Attention à la gestion des pointeurs !

- Pensez à conserver les références vers les données importantes de la liste avant de changer les pointeurs

Quelques exercices

1. comment réaliser l'affichage d'une liste récursivement ?
2. comment réaliser l'insertion en fin de liste de manière récursive ?

Affichage récursif d'une liste

- l'idée consiste à utiliser le caractère récursif de `Node<T>`
- écrire une méthode récursive qui prend en argument un `Node<T>`
- au 1er appel on passera la tête de la liste en argument de cette méthode
- on encapsule l'appel à cette méthode privée dans une méthode publique au niveau de `SList<T>`

Affichage récursif d'une liste

```
public String toStringNode() {
  if (head == null) return "()";
  else return this.nodeToString(head);
}

private String nodeToString(Node<T> p) {
  if (p.next != null) {
    return p.value.toString() + ", " + nodeToString(p.next);
  } else return p.value.toString();
}
```

Insertion récursive d'un élément en fin de liste

- même principe : travailler au niveau de `Node<T>` en interne
- attention ajouter un élément en récursif implique de reconstruire intégralement le chaînage de la liste

Insertion récursive d'un élément en fin de liste

```
public void enqueueRec(T elem) {
    head = enqueue(elem, head);
    size ++;
    last = last.next;
}

// Add in the end (like enqueue for a queue structure)
private Node<T> enqueue(T elem, Node<T> L) {
    if (L == null) {
        return new Node<T>(elem);
    }
    else return new Node<T>(L.value, enqueue(elem, L.next));
}
```