

Programmation et Algorithmique II

Ch.7 – Structures de données et Java Collections Framework

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

1. Introduction
2. *Java Collections Framework*
3. Tableaux dynamiques
4. Listes chaînées
5. Tables de hachage

Introduction

- **Objectifs**

- Les objectifs de ce chapitre sont
 - Introduire le **Java Collections Framework** (JCF) un groupe de *classes* et *algorithmes* de la bibliothèque Java permettant de manipuler facilement et efficacement des ensembles de données.
 - Comprendre le fonctionnement de certaines des structures de données mises à disposition par le JCF : les *listes chaînées*, les *tableaux dynamiques* et les *tables de hachage*.
 - Implémenter les structures de données en mettant en oeuvre les principes/mécanismes de P.O.O. de Java

Introduction

- **Collections**

- Une **Collection** est un terme générique désignant une *structure de données destinée à conserver un ensemble d'autres objets* (ou les références vers ces objets)
- Une collection possède des méthodes qui permettent typiquement
 - d'**ajouter un objet** à la collection
 - de **supprimer un objet** de la collection
 - de **recupérer un objet** ou tous les objets de la collection.
 - d'**obtenir la taille** (nombre d'éléments) de la collection
- Il existe plusieurs types de collections. Ceux-ci diffèrent selon les opérations disponibles et les contraintes imposées sur l'ensemble d'objets.

Introduction

- **Types de collections**

- Il existe différents types de collections qui sont distingués selon

- **Ordre**

- la collection garde-t-elle les éléments ordonnés ?

- **Type des éléments**

- les éléments de la collection peuvent-ils être de types différents (collections hétérogènes) ?

- **Duplication / null**

- les éléments *dupliqués* ou **null** sont-ils admis ?

- **Méthodes d'accès**

- l'accès est-il *séquentiel* ou *aléatoire* ?

- **Modifications concurrentes**

- la modification par plusieurs *threads* est-elle permise ?

Table des Matières

1. Introduction

2. *Java Collections Framework*

1. Collections

2. Associations

3. Tableaux dynamiques

4. Listes chaînées

5. Tables de hachage

Java Collections Framework

- **Introduction**

- Le **Java Collections Framework (JCF)** est un ensemble important d'interfaces, classes et méthodes destinées à manipuler des collections d'objets.
- Le JCF comprend
 - **Interfaces** : décrivent les services fournis par les collections
 - p.ex. List, Set, Map, Queue, Deque, ...
 - **Implémentations** : classes concrètes
 - p.ex. ArrayList, LinkedList, HashMap, ...
 - **Algorithmes** travaillant sur les collections
 - p.ex. tri, recherche, permutation aléatoire, min/max, ... (sort, binarySearch, shuffle, ...)
- Le JCF est découpé en deux grandes catégories : les collections et les associations (*maps*)

Table des Matières

1. Introduction

2. *Java Collections Framework*

1. Collections

2. Associations

3. Tableaux dynamiques

4. Listes chaînées

5. Tables de hachage

Java Collections Framework

- **Interface Collection**

- Le plus petit élément commun aux collections du JCF est l'interface **Collection** (`java.util`). Cette interface définit les opérations liées à la gestion d'un ensemble quelconque d'éléments.

```
public interface Collection<E>
    extends Iterable<E>
{
    boolean    add(E o);
    boolean    addAll(Collection<? extends E> c);
    void       clear();
    boolean    contains(E o);
    boolean    isEmpty();
    Iterator<E> iterator();
    boolean    remove(E o);
    boolean    removeAll(Collection<?> c);
    int        size();
    Object []  toArray();
    <T> T []   toArray(T [] a);
    /* ... */
}
```

Java Collections Framework

- **Exemple – tableau dynamique**

- L'exemple suivant illustre l'utilisation d'une liste, implémentée avec un tableau dynamique (`ArrayList`). Contrairement aux tableaux vus au Ch.2, il n'est pas nécessaire de spécifier la taille d'un tableau dynamique car elle est variable.

```
Collection<String> etudiants= new ArrayList<>();  
etudiants.add("Véronique");  
etudiants.add("Jef");  
etudiants.add("Tom");  
etudiants.add("Luc");
```

```
for (String s: etudiants)  
    System.out.println(s);
```

```
etudiants.remove("Luc");  
etudiants.add("Bruno");  
etudiants.add("Hadrien");  
etudiants.add("Souhaib");  
etudiants.add("Stéphane");
```

```
System.out.println(etudiants.size());
```

affiche

Véronique
Jef
Tom
Luc

affiche

7

Java Collections Framework

- Interfaces et classes génériques

- La notation `ArrayList<E>` permet de restreindre le type des instances ajoutées à l'`ArrayList`.
- `E` est un *paramètre de type*. Il est utilisé à l'intérieur de la classe, par exemple pour spécifier les arguments ou types de retour de méthodes.

```
public class ArrayList<E> ... {  
    public boolean add(E o) {...}  
    public E get(int i) {...}  
}
```

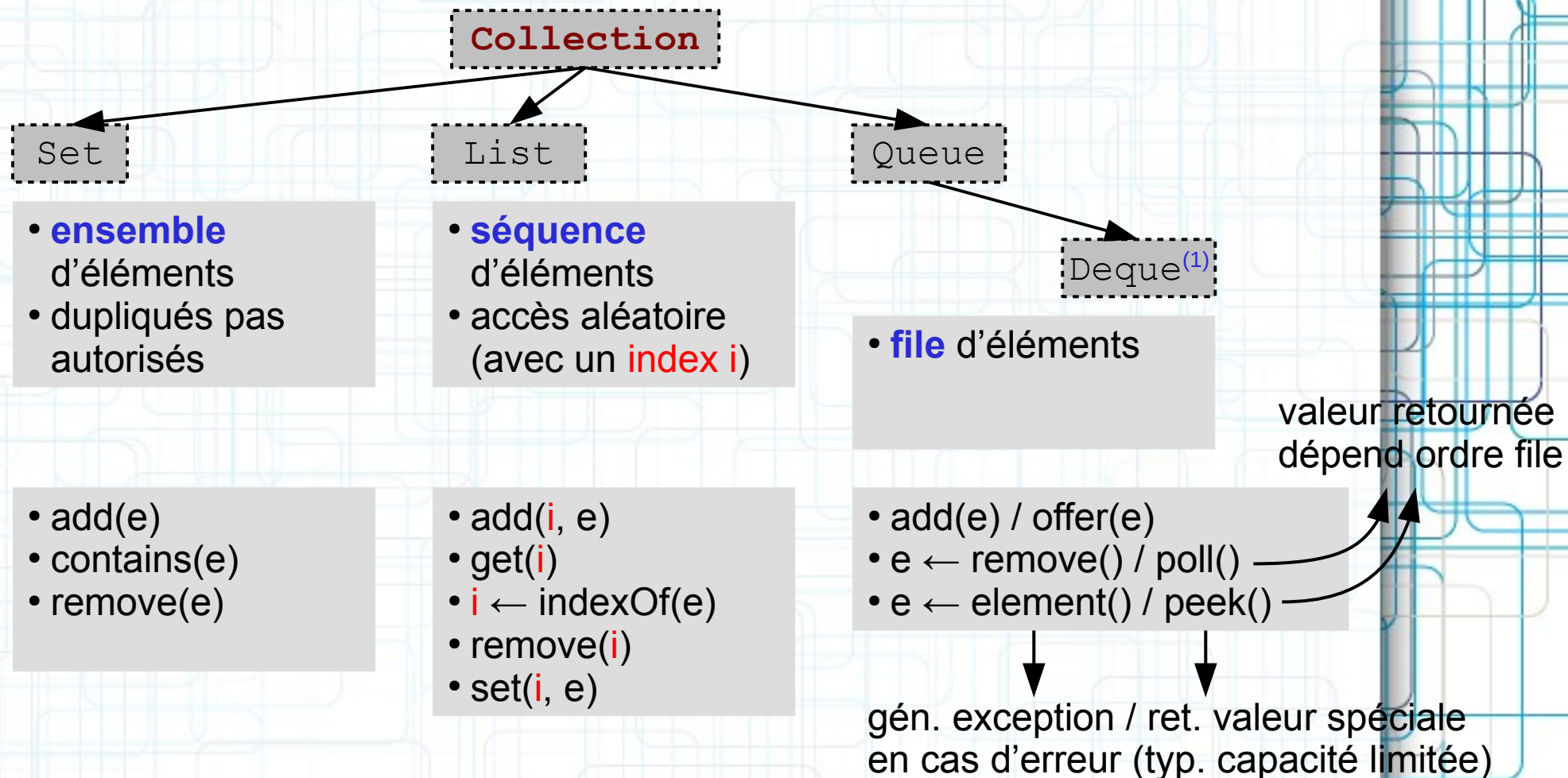


```
public class ArrayList<Carre> ... {  
    public boolean add(Carre o) {...}  
    public Carre get(int i) {...}  
}
```

- Le type `E` ne peut être un type primitif.
- Le compilateur vérifie que les instances passées comme paramètres aux méthodes d'`ArrayList` (p.ex. à la méthode `add`) sont compatibles avec `E`.
- On dit que la classe `ArrayList` est une **classe générique**. Le mécanisme *generics* sera discuté dans un chapitre ultérieur.

Java Collections Framework

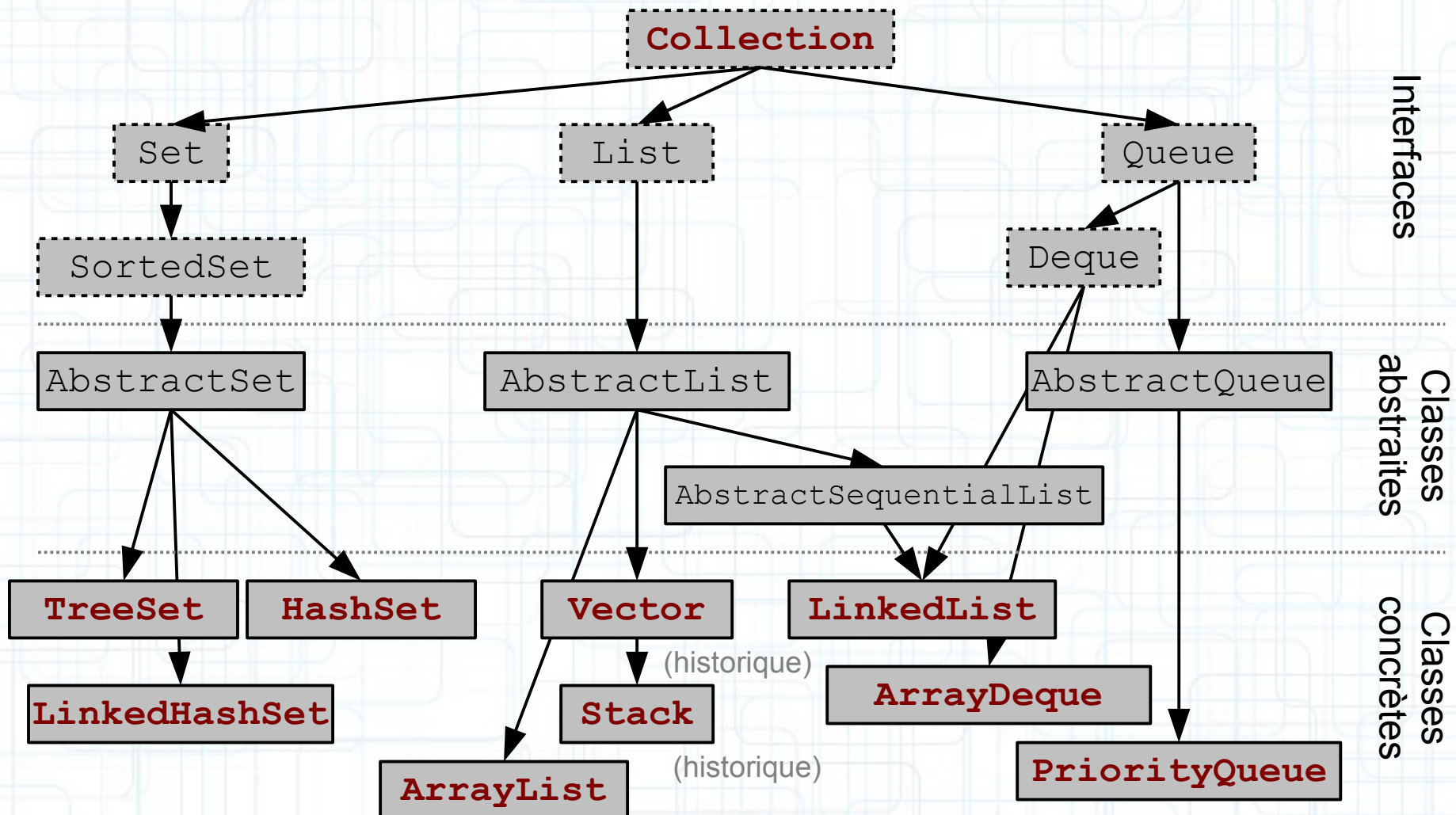
• Hiérarchie de classes Collection



(1) DeQue = *Double-ended Queue*

Java Collections Framework

- Hiérarchie de classes Collection



Java Collections Framework

- Hiérarchie de classes Collection

		dupli qués	null	ordon né	synch ronisé	remarques
Deque	List	ArrayList	✓	✓		tableau dynamique
		Vector	✓	✓	✓	tableau dynamique, (plus de contrôle que ArrayList)
		Stack	✓	✓	✓	LIFO (basé sur Vector)
Queue		LinkedList	✓	✓		liste doublement chaînée
		ArrayDeque	✓			FIFO, LIFO, ... (basé sur un tableau)
		PriorityQueue	✓			tas binaire
Set		HashSet				table de hachage
		TreeSet		✓ (1)		arbre binaire (rouge-noir)
		LinkedHashSet		✓ (2)		table de hachage + liste doublement chaînée

Table des Matières

1. Introduction

2. *Java Collections Framework*

1. Collections

2. Associations

3. Tableaux dynamiques

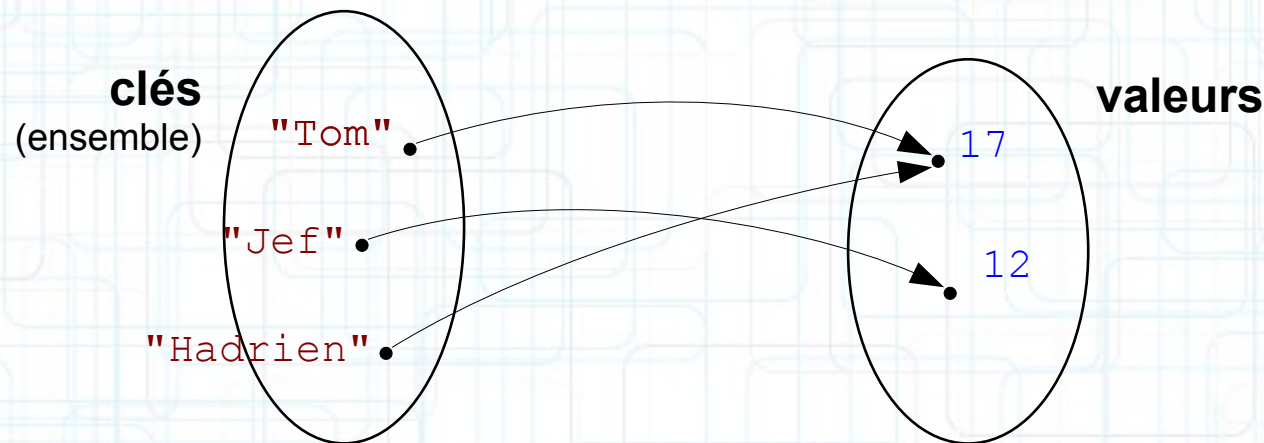
4. Listes chaînées

5. Tables de hachage

Java Collections Framework

- **Associations (Map)**

- Le JCF définit également des classes permettant de conserver l'**association** entre des paires d'éléments **clés** (*keys*) et **valeurs** (*values*).



- Il est possible de retrouver la valeur d'une association à l'aide de la clé qui lui est associée. Les clés ne peuvent être dupliquées (ensemble).
- Les clés et les valeurs des associations peuvent être récupérées indépendamment sous forme de collections.

Java Collections Framework

- **Interface Map**

- L'interface **Map** (`java.util`) définit les opérations liées à la gestion d'une association.

```
public interface Map<K,V>
{
    void                clear();
    boolean              containsKey(Object key);
    boolean              containsValue(Object value);
    Set<Map.Entry<K,V>> entrySet();
    V                    get(Object key);
    boolean              isEmpty();
    Set<K>               keySet();
    V                    put(K key, V value);
    void                putAll(Map<? extends K, ? extends V> t);
    V                    remove(Object key);
    int                 size();
    Collection<V>       values();
    ...
}
```

Java Collections Framework

- **Exemple – table de hachage**

- L'exemple suivant illustre l'utilisation d'une association implémentée avec une table de hachage (HashMap).

```
Map<String,Integer> resultats= new HashMap<>();
resultats.put("Bruno", 18);
resultats.put("Véronique", 16);
resultats.put("Jef", 12);
resultats.put("Hadrien", 17);
resultats.put("Tom", 17);

String key= "Tom";
if (resultats.containsKey(key)) {
    System.out.println("Résultat de \""+key+"\" : "+
        resultats.get(key));
}
```

clé

valeur

donne

Tom : 17

- Rappel : une clé est associée à une valeur unique !
 - utiliser 2 fois `put()` avec la même clé effectue un remplacement


Java Collections Framework

- **Exemple – table de hachage**

- L'exemple suivant illustre la récupération des ensembles de clés et de valeurs d'une association avec les méthodes `keySet()` et `values()`.


```
Map<String,Integer> resultats= new HashMap<>();  
/* ... */
```

```
Set<String> keys= resultats.keySet();  
for (String k: keys)  
    System.out.println(k);
```



Bruno
Véronique
Jef
Hadrien
Tom

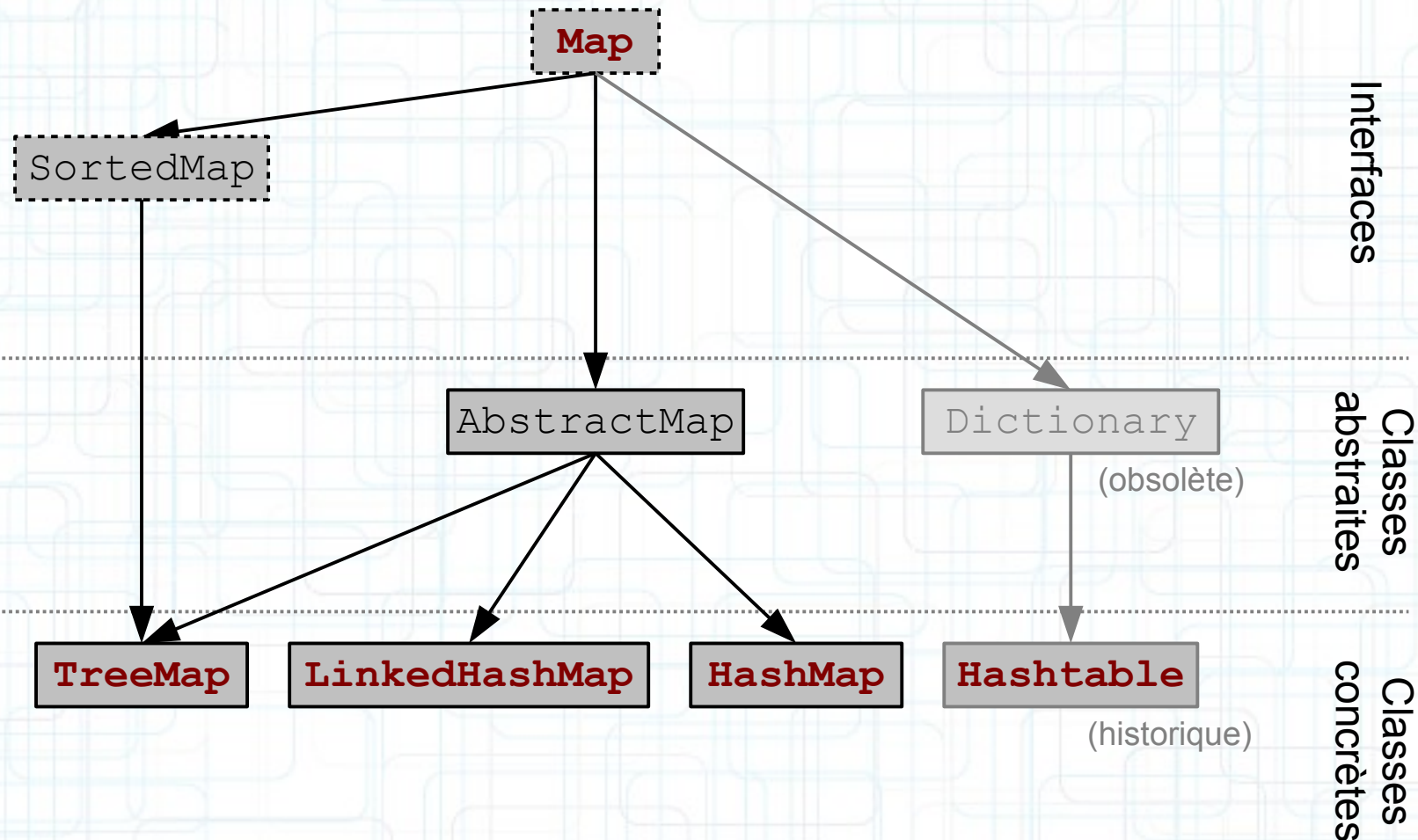
```
Collection<Integer> values= resultats.values();  
for (Integer v: values)  
    System.out.println(v);
```



18
15
12
17
17

Java Collections Framework

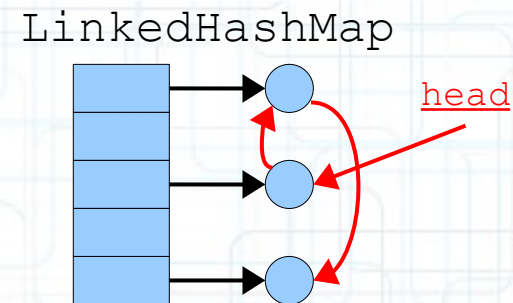
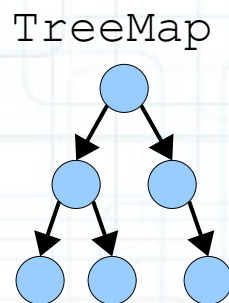
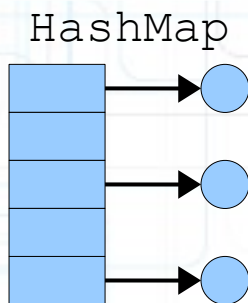
- Hiérarchie de classes Map



Java Collections Framework

- Hiérarchie de classes Map

	clés		valeurs	synchr onisé	remarques
	null	ordo nnée s	null		
HashMap	✓		✓		table de hachage
TreeMap		✓ (1)	✓		arbre binaire (rouge-noir)
LinkedHashMap	✓	✓ (2)	✓		table de hachage + liste doublement chaînée
Hashtable				✓	table de hachage



(1) ordre naturel ; (2) ordre d'insertion

Java Collections Framework

• Exemple – table de hachage

- Quel est l'ordre des clés ?
 - HashMap ne fournit aucune garantie sur cet ordre.
 - TreeMap utilise l'ordre naturel (i.e. Comparable)
 - LinkedHashMap conserve l'ordre d'insertion

```
Map<String,Integer> resultats= new (Hash|Tree|LinkedHash)Map<>();  
resultats.put("Bruno", 18);  
resultats.put("Véronique", 16);  
resultats.put("Jef", 12);  
resultats.put("Hadrien", 17);  
resultats.put("Tom", 17);
```

```
Set<String> keys= resultats.keySet();  
for (String k: keys)  
    System.out.println(k);
```

HashMap

Bruno
Tom
Hadrien
Véronique
Jef

TreeMap

Bruno
Hadrien
Jef
Tom
Véronique

LinkedHashMap

Bruno
Véronique
Jef
Hadrien
Tom

Table des Matières

1. Introduction
2. *Java Collections Framework*
- 3. Tableau dynamiques**
 - 1. Principe et implémentation**
 2. Stratégies de croissance
4. Listes chaînées
5. Tables de hachage

Listes

- **Introduction**

- Une **liste** est une séquence ordonnée d'éléments. Une liste peut contenir des éléments dupliqués.
- Une liste est un **type de données abstrait** (ADT – *Abstract Data Type*). Un ADT définit les services d'une structure de données sans imposer une implémentation particulière.
- Il existe plusieurs implémentations de l'ADT liste
 - `ArrayList` repose sur l'utilisation de *tableaux*.
 - `LinkedList` repose sur les *listes chaînées*.

Listes

- **Interface List**

- L'interface `List` du JCF définit les méthodes que toute implémentation d'une liste doit fournir. Un extrait de ces méthodes est fourni ci-dessous.

```
public interface List<E> extends Collection<E>
{
    boolean    add(E o);
    void       clear();
    boolean    contains(E o);
    E          get(int index);
    int        indexOf(E o);
    boolean    isEmpty();
    boolean    remove(int index);
    boolean    remove(E o);
    void       set(int index, E o);
    int        size();
}
```

Récupère un élément à partir de sa position absolue dans la liste (obligatoire).

Fournit le nombre d'éléments de la liste (obligatoire).

Notes :

- Toutes les méthodes ne doivent pas nécessairement être implémentées (voir documentation). Les méthodes optionnelles non implémentées doivent générer une `UnsupportedOperationException`.
- La méthode `add(E o)` retourne un booléen de façon à permettre de refuser l'ajout de certains éléments.

Tableaux Dynamiques

- Exemple

```
import java.util.ArrayList;
import java.util.List;

List<Carre> carres= new ArrayList<Carre>();
carres.add(new Carre(5, 7, -60, 5));
carres.add(new Carre(4, 1, 0, 2));
System.out.println("Taille = "+carres.size());

for (int i= 0; i < carres.size(); i++)
    carres.get(i).deplacer(Math.random(),
                           Math.random());
carres.set(0, new Carre(2, 2, 2, 2));
```

Le constructeur crée un tableau interne vide.

Note : il existe également un autre constructeur qui prend un paramètre **int** et crée un tableau de N éléments.

La méthode `size` retourne la taille actuelle du tableau.

La méthode `add` ajoute un élément à la fin du tableau. La taille du tableau interne est augmentée si nécessaire.

Le paramètre de type **E** est égal à **Carre**. Cela affecte le type de retour de `get`, ainsi que les arguments de `add` et `set`.

Tableaux Dynamiques

- **ArrayList vs tableau**

- **Tableau**

- Taille fixe
 - Cellules peuvent être de type primitif
 - Accès avec l'opérateur d'indexation []
 - Exception `ArrayIndexOutOfBoundsException`

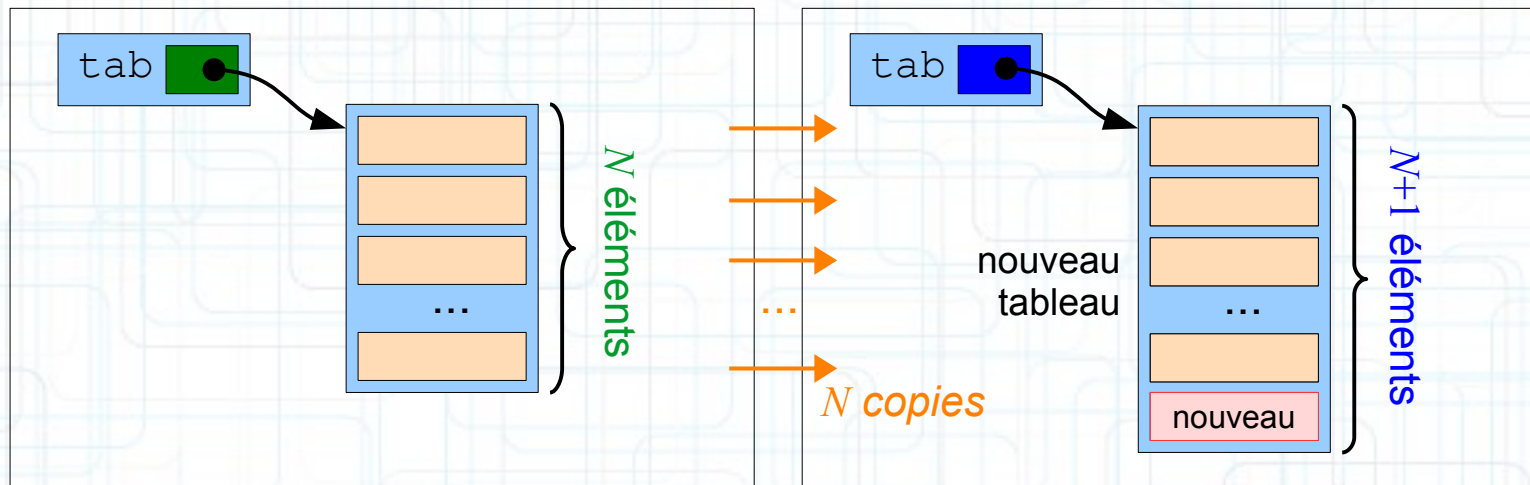
- **ArrayList**

- Taille variable
 - Éléments uniquement de type objet
 - Accès via `set()` et `get()`
 - Exception `IndexOutOfBoundsException`
 - Insertion/suppression d'un élément à une position arbitraire

Tableaux Dynamiques

- **Implémentation**

- Une tableau dynamique utilise un **tableau interne** (de taille fixe) pour stocker ses éléments.
- Lors de l'ajout d'un **nouvel élément**, il est possible qu'il n'y ait plus suffisamment de place dans le tableau interne. Auquel cas, il faut « **ré-allouer** » le **tableau** et y **copier** les éléments du tableau précédent.



- La ré-allocation et la copie des éléments prennent un temps proportionnel à la taille du tableau.

Tableaux Dynamiques

- Implémentation

```
public class MyArrayList {  
    protected Object[] tab = null; ..... référence vers  
    ..... tableau interne  
    public void add(Object o) {  
        int oldSize = size();  
        Object[] newTab = new Object[oldSize + 1]; ..... ré-allocation  
        if (oldSize > 0) .....  
            System.arraycopy(tab, 0, newTab, 0, oldSize); ..... copie  
        newTab[oldSize] = o;  
        tab = newTab;  
    }  
    public Object get(int i) {  
        if (tab == null)  
            throw new IndexOutOfBoundsException();  
        return tab[i];  
    }  
    public int size() {  
        if (tab == null)  
            return 0;  
        return tab.length;  
    }  
}
```


Tableaux Dynamiques

- **Complexité de l'ajout**

- Intéressons-nous à la complexité de l'ajout d'un élément à la fin d'un tableau dynamique.
- Prenons les hypothèses suivantes
 - la **ré-allocation** du tableau se fait en un **temps constant**.
 - la **copie d'un nouvel élément** dans le tableau se fait en un **temps constant**.
- La **copie des éléments du tableau précédent** lors d'une ré-allocation nécessite la **copie des N éléments** du tableau précédent.
- Nous nous focalisons donc sur le nombre de copies en cas de ré-allocation.

Tableaux Dynamiques

- **Complexité de l'ajout**

- Supposons que l'on ajoute n éléments à une `ArrayList` initialement vide.
 - 1^{er} ajout : pas de copie
 - 2^{ème} ajout : 1 copie
 - 3^{ème} ajout : 2 copies
 - ...
 - $n^{\text{ème}}$ ajout : $n-1$ copies
- Notons $C(k)$ le nombre de copies effectuées par `add()` lors de l'ajout du $k^{\text{ème}}$ élément. Dans notre implémentation, on a

$$C(k) = k - 1$$

Tableaux Dynamiques

- **Complexité de l'ajout**

- Quel est le **nombre total de copies** effectuées pour ajouter les n premiers éléments ?

$$C_{total}(n) = \sum_{k=1}^n C(k) = \sum_{k=1}^n k-1 = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

- Quel est le **nombre moyen de copies** effectuées par ajout pour les n premiers éléments ?

$$\bar{C}(n) = \frac{C_{total}(n)}{n} = (n-1)/2$$

Exemple :

k	$C(k)$	$C_{total}(k)$	$\bar{C}(k)$
1	0	0	0
2	1	1	1/2
3	2	3	1
4	3	6	3/2
5	4	10	10/5

Tableaux Dynamiques

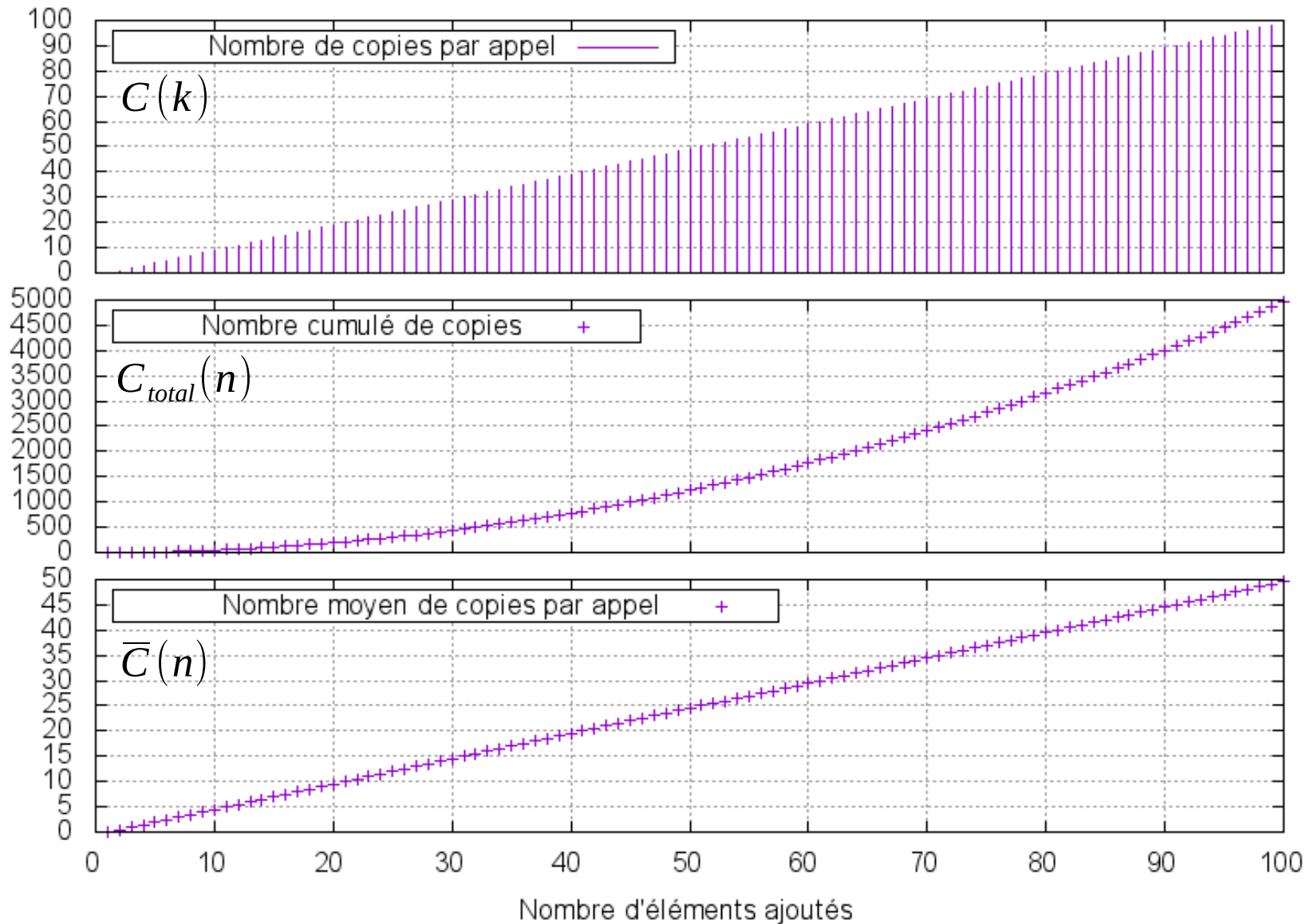


Table des Matières

1. Introduction
2. *Java Collections Framework*
- 3. Tableau dynamiques**
 1. Principe et implémentation
 - 2. Stratégies de croissance**
4. Listes chaînées
5. Tables de hachage

Tableaux Dynamiques

- **Complexité de add**

- La complexité temporelle de la méthode `add()` dans notre implémentation de `MyArrayList` **croît linéairement en fonction du nombre d'éléments** déjà dans le tableau !!!
- **Peut-on faire mieux ?**
 - En pratique, d'autres stratégies de croissance du tableau interne peuvent être utilisées.
 - **Stratégie additive** : ajouter à la taille du tableau une constante entière $a \geq 1$
 - **Stratégie multiplicative** : multiplier la taille du tableau par un facteur constant $b > 1$
- L'implémentation d'`ArrayList` de la librairie Java utilise une stratégie multiplicative.

Tableaux Dynamiques

- **Stratégie additive**

- Supposons que l'on ajoute n éléments à une `ArrayList` initialement vide avec une stratégie additive.
- Hypothèse : croissance de $a=10$ cellules
 - 1^{er} au 10^{ème} ajouts : pas de copie
 - 11^{ème} ajout : 10 copies
 - 12^{ème} au 20^{ème} ajouts : pas de copie
 - 21^{ème} ajout : 20 copies
 - 22^{ème} au 30^{ème} ajouts : pas de copie
 - ...

Tableaux Dynamiques

- **Stratégie additive**

- Avec cette stratégie, le nombre de copies pour le $k^{\text{ème}}$ ajout est égal à

$$C(k) = \begin{cases} k-1 & \text{si } \exists p \in \mathbb{N} : k=ap+1 \\ 0 & \text{sinon} \end{cases}$$

Exemple :
(a=10)

k	taille	$C(k)$	$C_{\text{total}}(k)$	$\bar{C}(k)$
1	$0 \rightarrow 10$	0	0	0
2	10	0	0	0
...				
11	$10 \rightarrow 20$	10	10	10/11
12	20	0	10	10/12
...				
21	$20 \rightarrow 30$	20	30	30/21
22	30	0	30	30/22
...				
31	$30 \rightarrow 40$	30	60	60/31
32	40	0	60	60/32

pires
cas

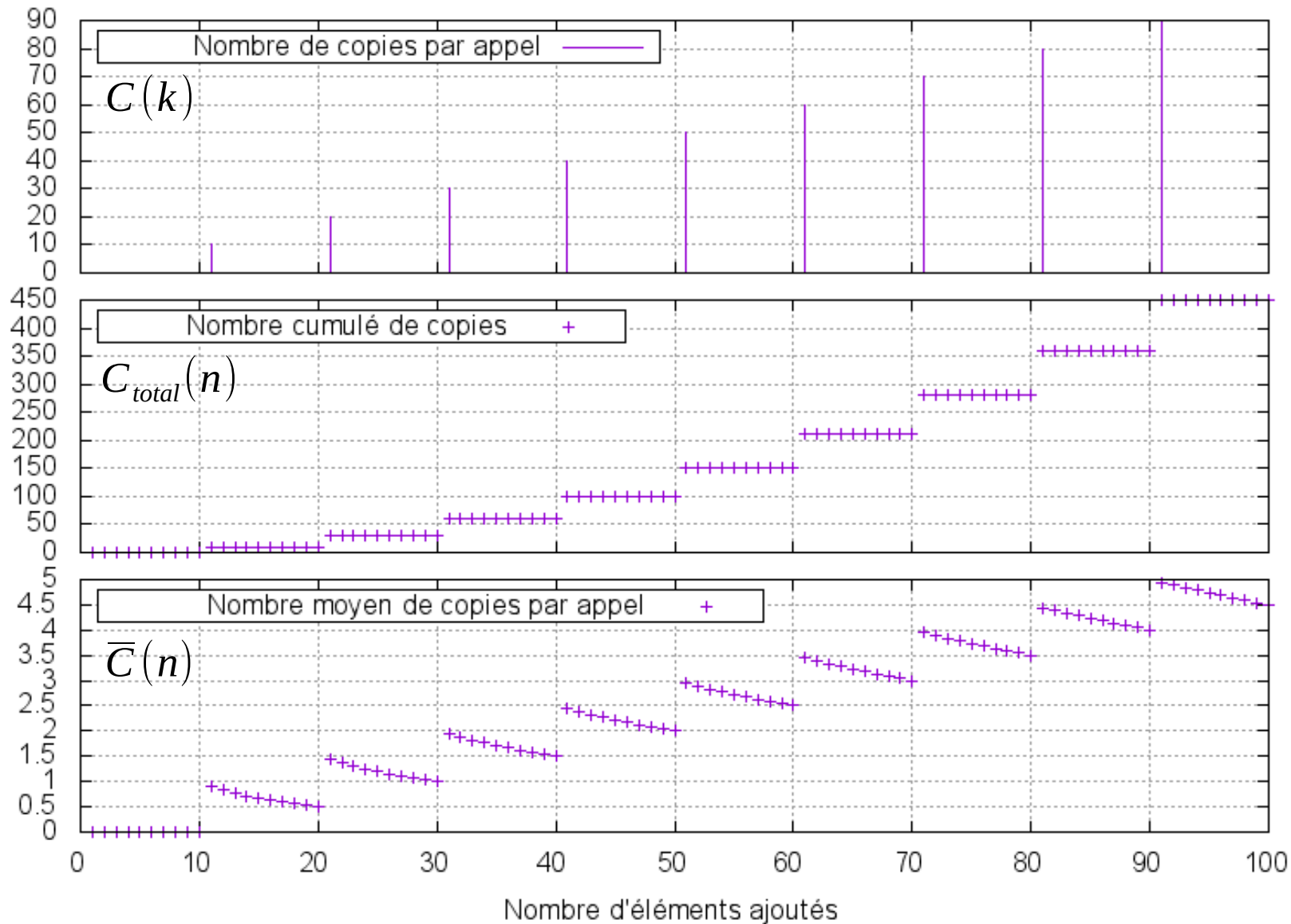


$k=a+1$

$k=2a+1$

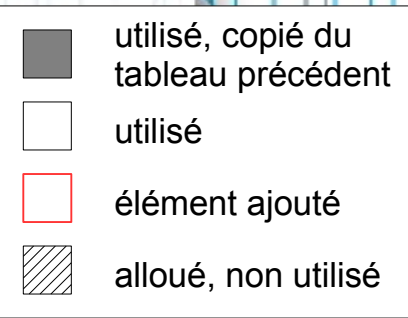
$k=3a+1$

Tableaux Dynamiques



Tableaux Dynamiques

- Stratégie multiplicative



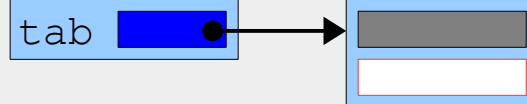
Situation initiale

tab null

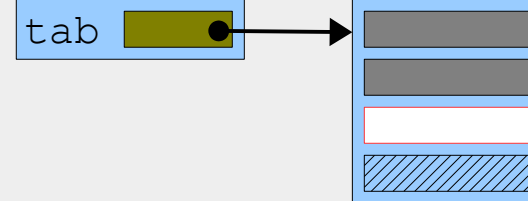
1^{er} ajout



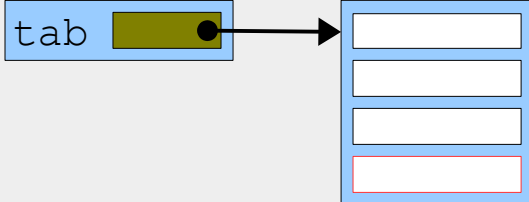
2^{ème} ajout



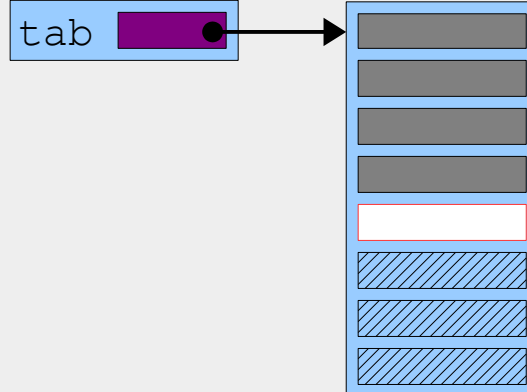
3^{ème} ajout



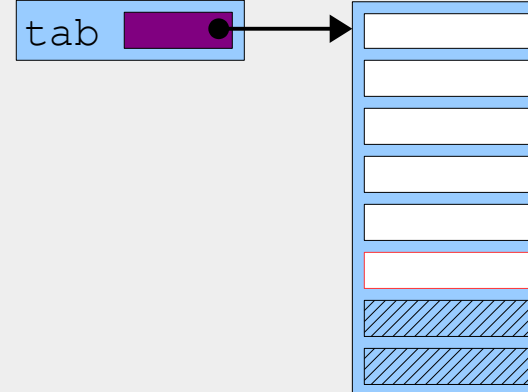
4^{ème} ajout



5^{ème} ajout



6^{ème} ajout



Tableaux Dynamiques

- **Stratégie multiplicative**

- Supposons que l'on ajoute n éléments à une `ArrayList` initialement vide avec une stratégie multiplicative.
- Hypothèse : facteur $b=2$
 - 1^{er} ajout : pas de copie
 - 2^{ème} ajout : 1 copie
 - 3^{ème} ajout : 2 copies
 - 4^{ème} ajout : pas de copie
 - 5^{ème} ajout : 4 copies
 - 6^{ème} au 8^{ème} ajouts : pas de copie
 - 9^{ème} ajout : 8 copies
 - 10^{ème} au 16^{ème} ajout : pas de copie
 - 17^{ème} ajout : 16 copies
 - ...

Tableaux Dynamiques

- **Stratégie multiplicative**

- Avec cette stratégie, le nombre de copies pour le $k^{\text{ème}}$ ajout est égal à

$$C(k) = \begin{cases} k-1 & \text{si } \exists p \in \mathbb{N} : k = b^p + 1 \\ 0 & \text{sinon} \end{cases}$$

Exemple :
($b=2$)

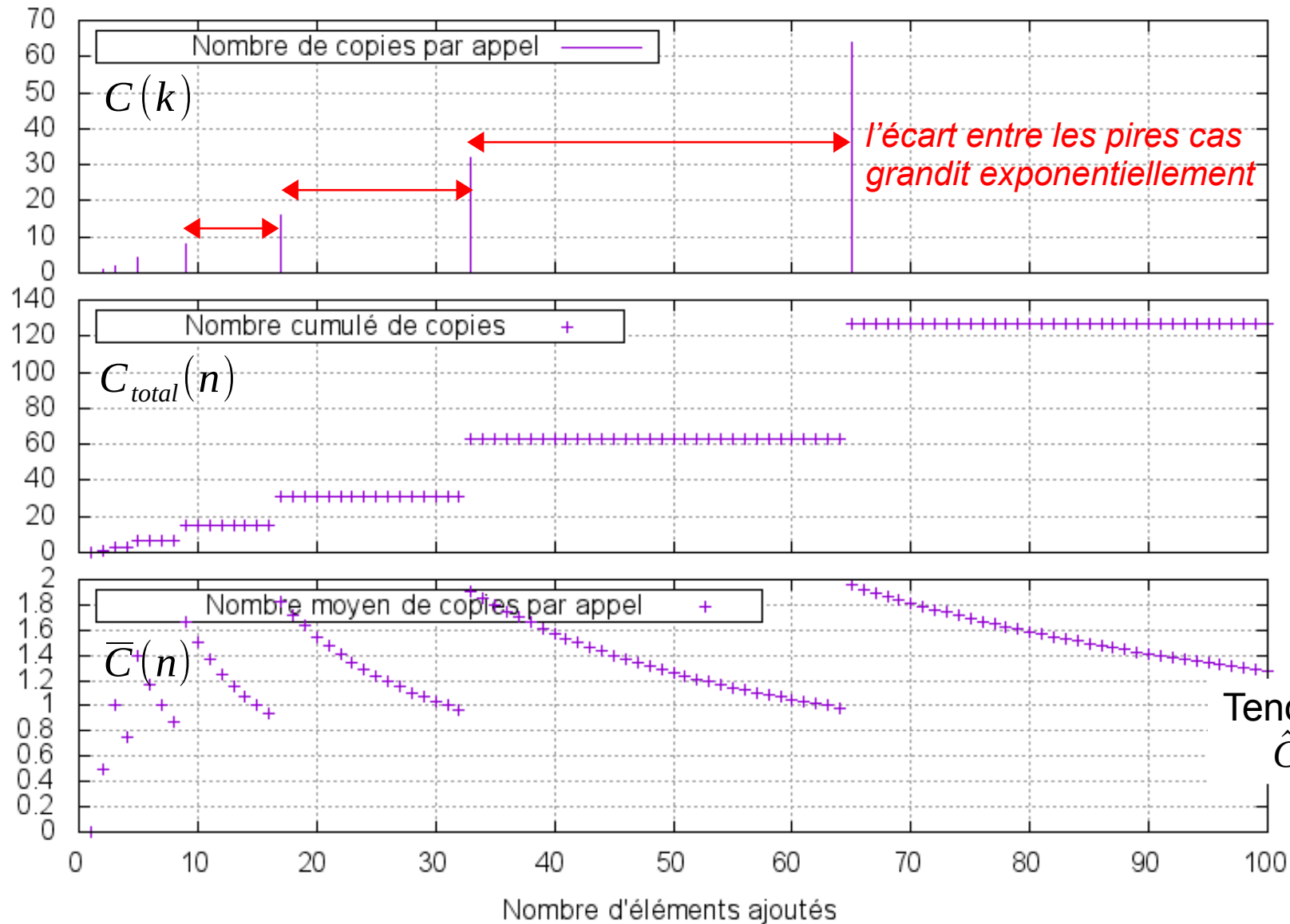
k	taille	$C(k)$	$C_{\text{total}}(k)$	$\bar{C}(k)$	
1	$0 \rightarrow 1$	0	0	0	
2	$1 \rightarrow 2$	1	1	$1/2$	$k=2^0+1$
3	$2 \rightarrow 4$	2	3	1	$k=2^1+1$
4	4	0	3	$3/4$	
5	$4 \rightarrow 8$	4	7	$7/5$	$k=2^2+1$
6	8	0	7	$7/6$	
7	8	0	7	$7/7$	
8	8	0	7	$7/8$	
9	$8 \rightarrow 16$	8	15	$15/9$	$k=2^3+1$
10	16	0	15	$15/10$	

pires
cas



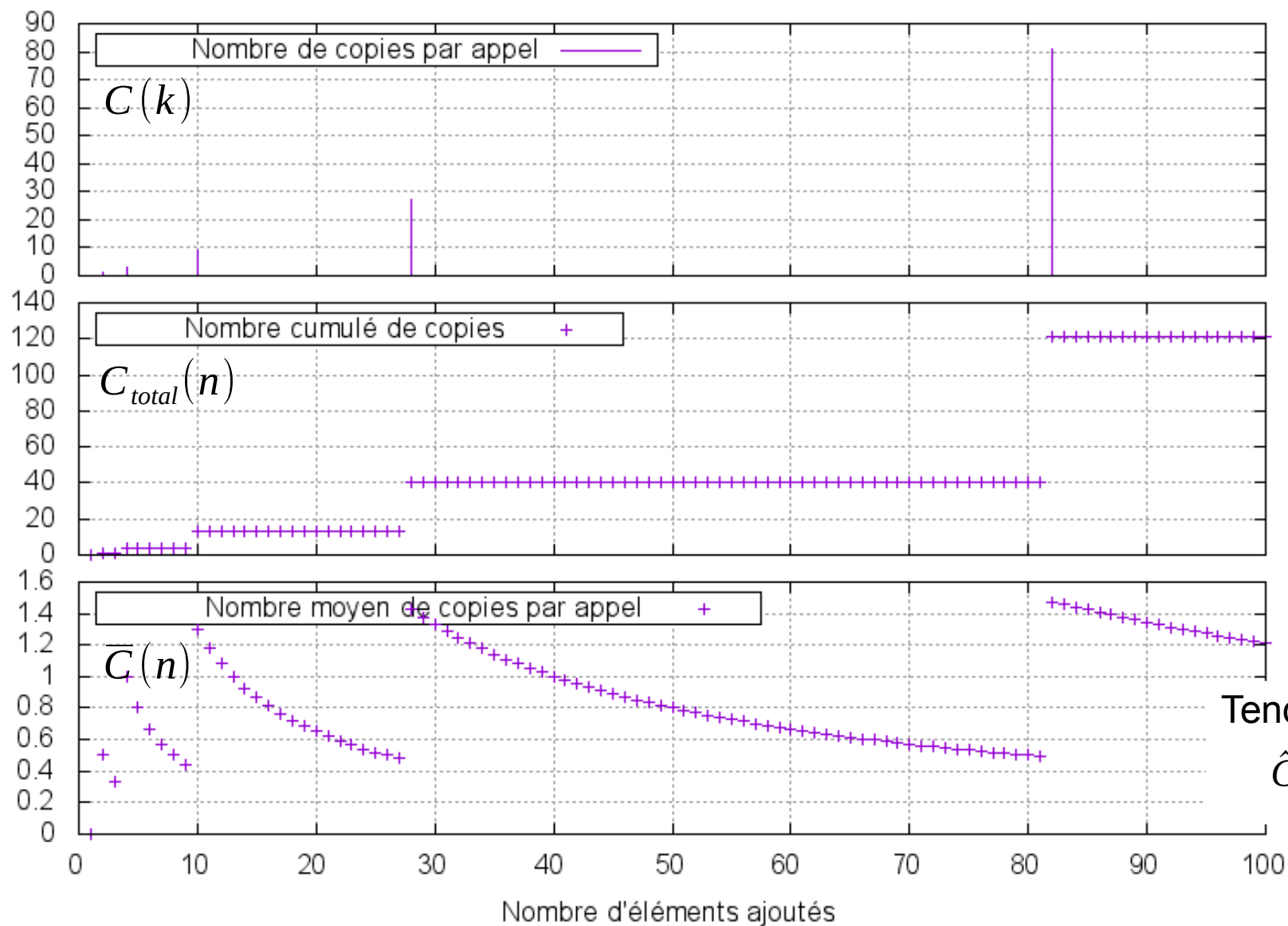
cas
 $b = 2$

Tableaux Dynamiques



cas
 $b = 3$

Tableaux Dynamiques



Tend vers
 $\hat{C} = \frac{3}{2}$

Tableaux Dynamiques

- **Stratégie multiplicative**

- Les **pires cas** arrivent lorsque $k=b^p+1$. Avec p entier.

- Effectuons un changement de variable de k à p

$$C(k)=k-1 \Rightarrow C(b^p+1) = b^p$$

- Le **nombre total de copies** effectuées lors d'une séquence d'ajouts devient

$$C_{total}(b^p+1) = \frac{1}{b-1}(b^{p+1}-1)$$

- Le **nombre moyen de copies** effectuées par ajout lors d'une séquence d'ajouts devient

$$\bar{C}(b^p+1) = \frac{C_{total}(b^p+1)}{b^p+1} = \frac{1}{b-1} \frac{b^{p+1}-1}{b^p+1}$$

Tableaux Dynamiques

- **Stratégie multiplicative**

- Le **nombre moyen de copies** a une limite finie lorsqu'une stratégie de croissance multiplicative est employée.

$$\hat{C} = \lim_{p \rightarrow +\infty} \overline{C}(b^p + 1) = \lim_{p \rightarrow +\infty} \frac{1}{b-1} \frac{b^{p+1} - 1}{b^p + 1} = \frac{b}{b-1}$$

- Par exemple
 - avec $b=2$, le nombre moyen de copies vaut 2
 - avec $b=3$, il vaut $3/2$
 - avec $b=3/2$, il vaut 3
- Note : avec une stratégie additive, cette limite est infinie.

Tableaux Dynamiques

- **Stratégie multiplicative**

- La complexité temporelle classique s'intéresse au pire des cas pour un **ajout individuel**.
 - La complexité est en $O(n)$ quelque soit la stratégie (additive ou multiplicative).
- Cependant, si on considère des **séquences de n ajouts**, certains ajouts sont coûteux et d'autres non. Le coût est amorti sur plusieurs ajouts.
 - La complexité pour une séquence de n ajouts est en $O(n)$ avec une **stratégie multiplicative**.
 - Elle est en $O(n^2)$ avec une **stratégie additive**.
- La complexité moyenne (amortie) d'un ajout individuel est en $O(1)$ avec une stratégie multiplicative.

Tableaux Dynamiques

- **Taux d'occupation**

- La stratégie multiplicative permet une complexité amortie constante. Cependant, **quel facteur multiplicatif b utiliser ?**
- Un second point important à considérer est le **taux d'occupation** du tableau interne.
 - La taille de celui-ci est toujours au moins égale au nombre d'éléments à stocker.
 - Lorsque la taille du tableau est supérieure au nombre d'éléments à stocker, un espace mémoire supplémentaire a été alloué mais n'est pas utilisé. Il s'agit d'une forme de gaspillage.
- Le **taux d'occupation** du tableau est mesuré comme le nombre d'éléments qu'il contient divisé par sa taille.

Tableaux Dynamiques

- **Taux d'occupation**

- Observons le **taux d'utilisation moyen** d'un tableau dynamique avec différentes stratégies de croissance

- Additive $a=1$: toujours 100 %
- Additive $a=10$: tend vers 100 % (avec assez d'éléments)
- Multiplicative $b=2$: **75 %** (pire = ~50%)
- Multiplicative $b=3$: **66 %** (pire = ~33%)
- Multiplicative $b=3/2$: **83 %** (pire = ~ 66%)

- Le choix du facteur b de la stratégie multiplicative est donc un **compromis** entre rapidité et espace mémoire consommé.

- $b=2 \rightarrow$ nombre moyen de copies = 2
taux moyen d'occupation = 75 %
- $b=3/2 \rightarrow$ nombre moyen de copies = 3
taux moyen d'occupation = 83 %

pire

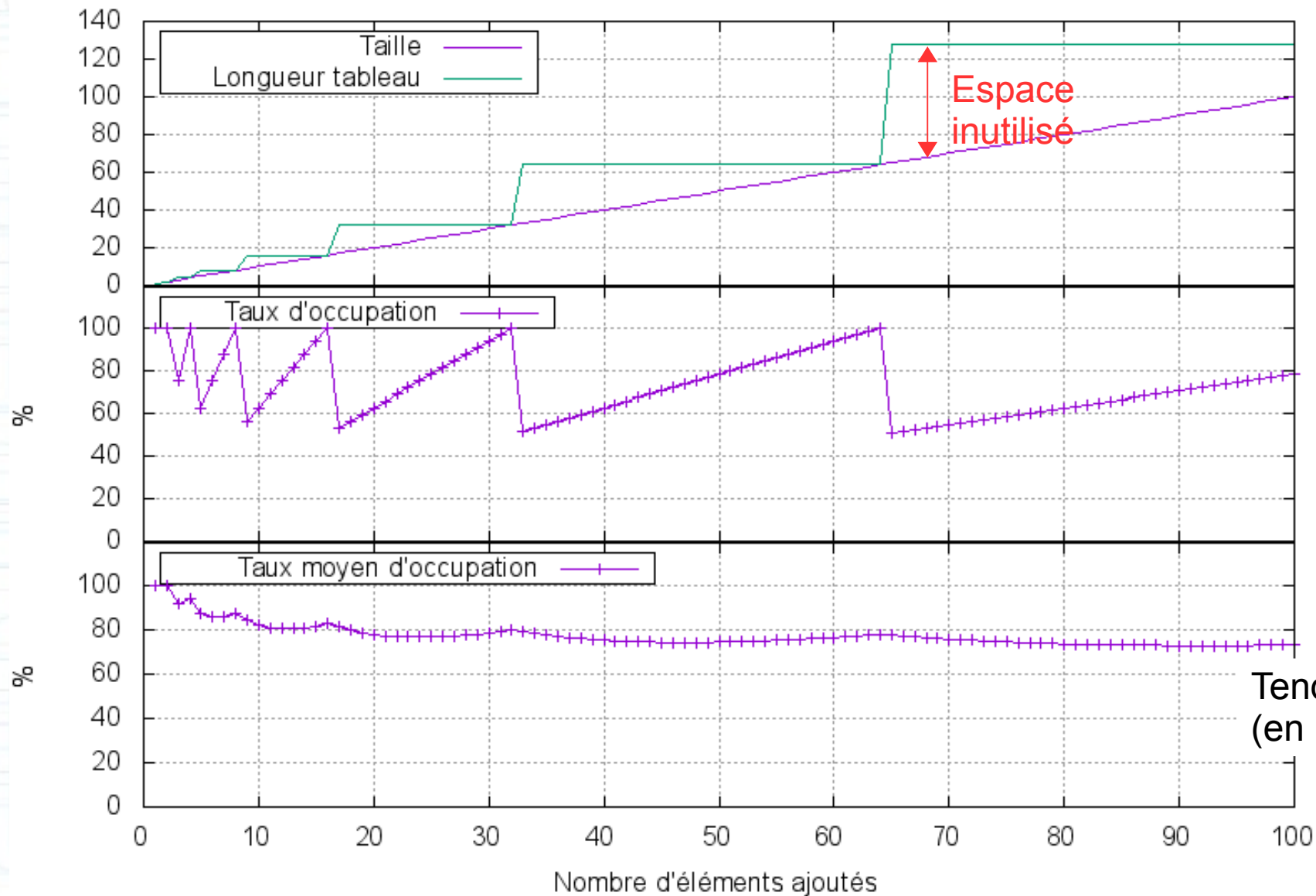
$$\frac{ap+1}{a(p+1)}$$

$$\frac{b^p+1}{b^{p+1}}$$

le JCF utilise $b=3$
pour *ArrayList*

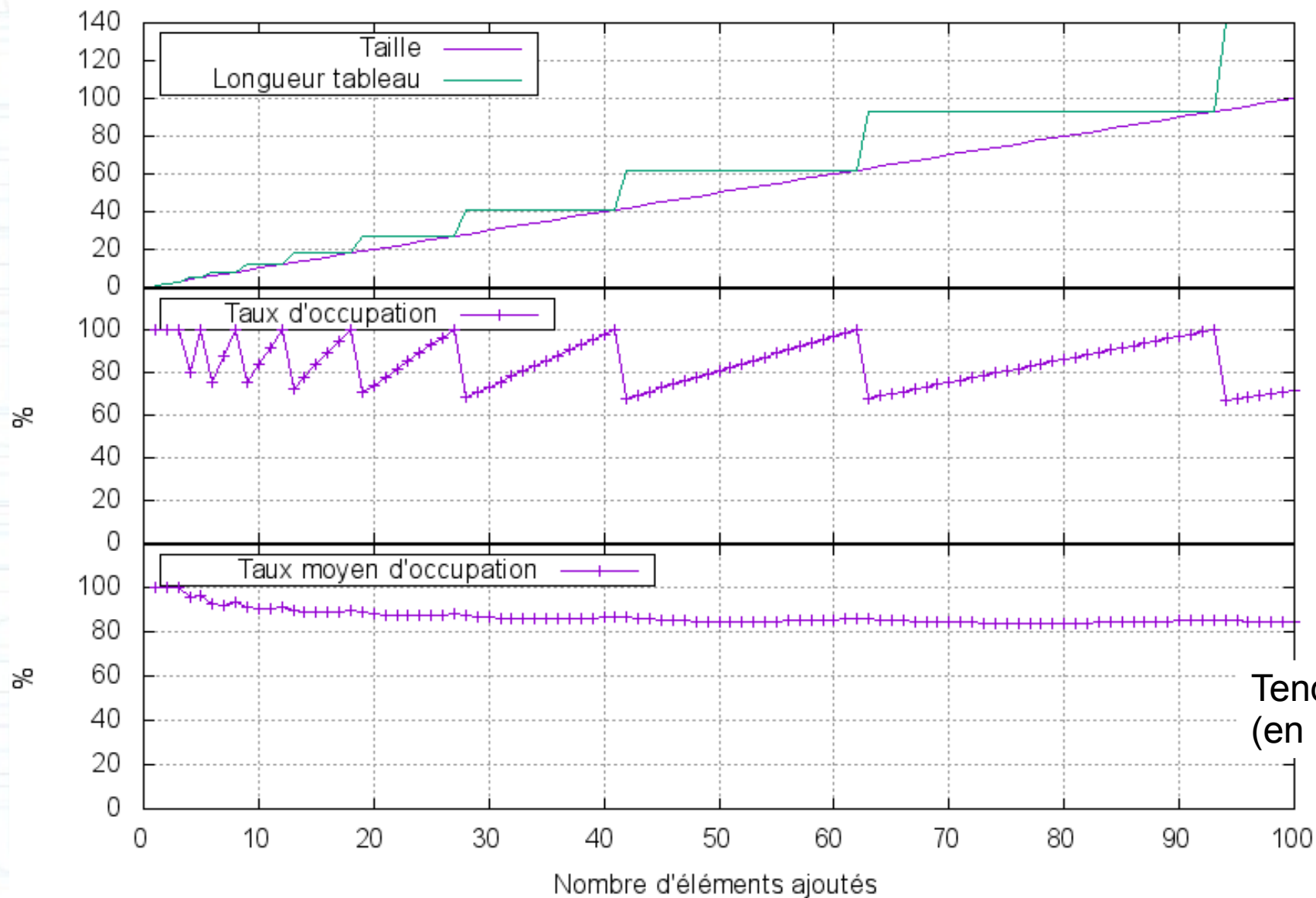
cas
 $b = 2$

Tableaux Dynamiques



cas
 $b = 3/2$

Tableaux Dynamiques



Tableaux Dynamiques

- **Opérations d'insertion / suppression**
 - D'autres opérations fournies par la classe `ArrayList` sont moins efficaces: il s'agit des opérations **d'insertion et de suppression**. Le coût de ces opérations vient de la nécessité de déplacer un grand nombre d'éléments du tableau interne à `ArrayList`.

Tableaux Dynamiques

- **Insertion**

- La méthode `add(int index, E o)` de la classe `ArrayList` permet l'insertion d'une référence à une position spécifique de la liste.
 - La méthode `add(E o)` que nous avons étudiée précédemment est un cas particulier d'insertion en fin de liste : `add(size(), o)`

- Exemple

```
List<Carre> carres= new ArrayList<Carre>();  
carres.add(new Carre(5, 7, -60, 5));  
carres.add(new Carre(4, 1, 0, 2));  
carres.add(1, new Carre(2, 2, 2, 2));  
for (int i= 0; i < carres.size(); i++)  
    System.out.println(carres.get(i));
```

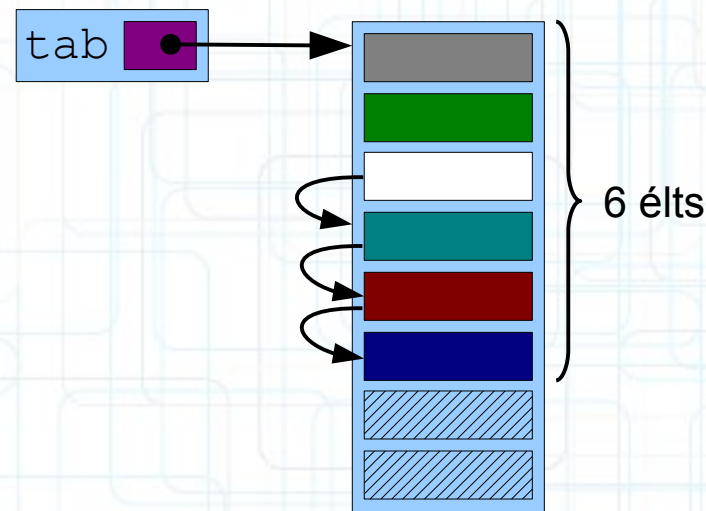
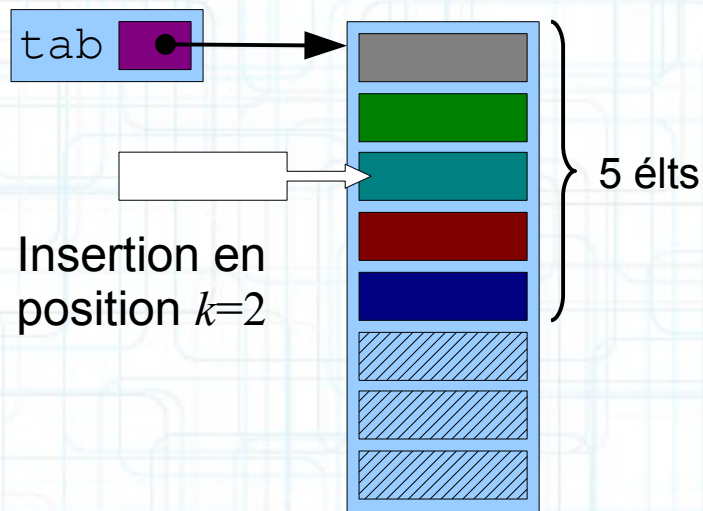
```
bash-3.2$ java ArrayListEx  
Carre(x=5.0;y=7.0)  
Carre(x=2.0;y=2.0)  
Carre(x=4.0;y=1.0)
```

Elément inséré à l'index 1

Tableaux Dynamiques

- **Complexité de l'insertion**

- L'insertion dans un tableau de taille N en position k nécessite le déplacement de $N-k$ éléments.

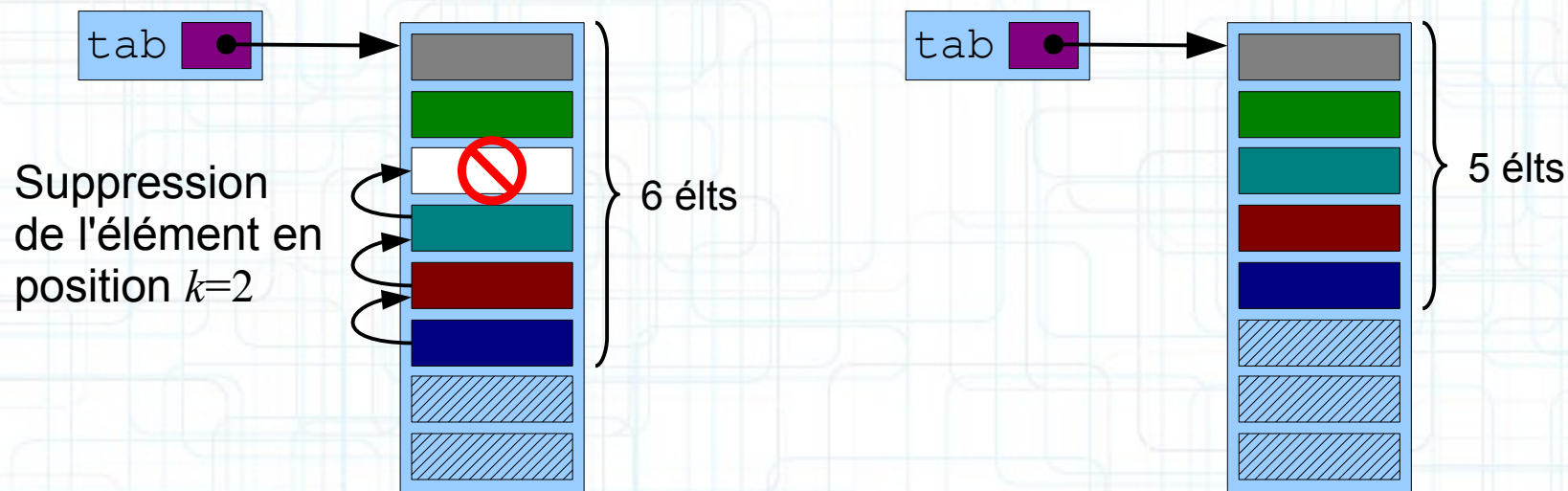


- La complexité temporelle de l'insertion (dans le pire des cas) est donc $O(N)$.
- Note : on ne considère pas le coût de l'expansion éventuelle du tableau. Est-ce un problème ?

Tableaux Dynamiques

- **Complexité de la suppression**

- La suppression d'un tableau de taille N en position k nécessite le déplacement de $N-k-1$ éléments.



- La complexité temporelle de la suppression (dans le pire des cas) est donc $O(N)$

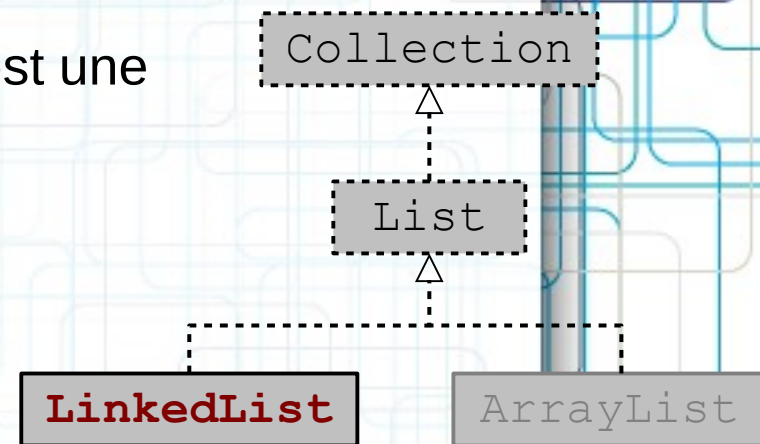
Table des Matières

1. Introduction
2. *Java Collections Framework*
3. Tableau dynamiques
4. **Listes chaînées**
5. Tables de hachage

Listes Chaînées

- Introduction

- Une **liste chaînée** est une structure de données qui met en oeuvre l'ADT Liste, i.e. une séquence d'éléments.
- Contrairement aux tableaux dynamiques, elle permet l'insertion et la suppression efficaces d'éléments en milieu de liste.
- En revanche, elle est moins efficace que les tableaux dynamiques pour accéder à un élément situé à une position arbitraire.
- La classe `LinkedList` fournie par le JCF est une implémentation de liste chaînée.



Listes Chaînées

- Exemple

```
import java.util.LinkedList;
import java.util.List;

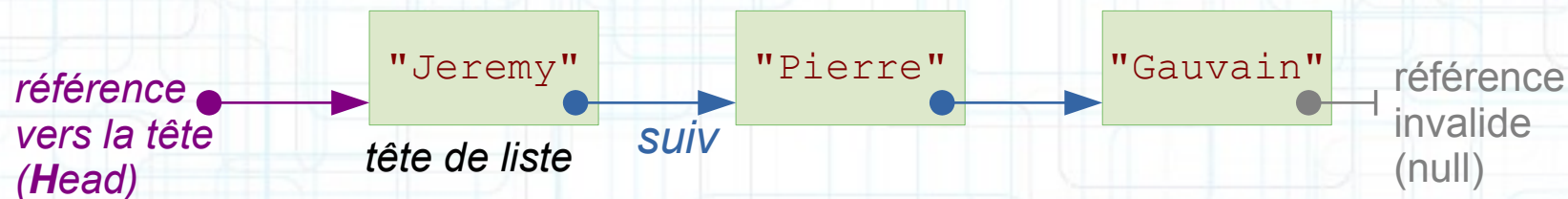
List<Carre> carres= new LinkedList<Carre>();
carres.add(new Carre(5, 7, -60, 5));
carres.add(new Carre(4, 1, 0, 2));
System.out.println("Taille = "+carres.size());

for (int i= 0; i < carres.size(); i++)
    carres.get(i).deplacer(Math.random(),
                           Math.random());
carres.set(0, new Carre(2, 2, 2, 2));
```

Listes Chaînées

- **Principe**

- Une liste chaînée est une séquence de **nœuds**.
- Chaque nœud est un *objet* qui stocke
 - un **élément** ou la *référence vers cet élément*
 - un **lien** (référence) vers le nœud *suivant*

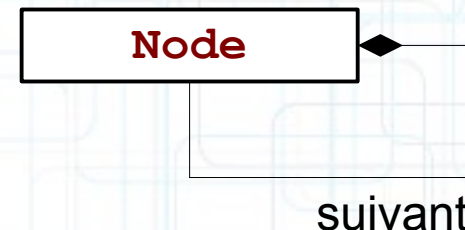


- Le premier nœud est appelé la *tête de la liste (head)*. Conserver la **référence vers la tête** suffit à accéder à l'entièreté de la liste.
- Une telle liste est appelée *liste simplement chaînée*.

Listes Chaînées

- **Structure récursive / auto-référentielle**

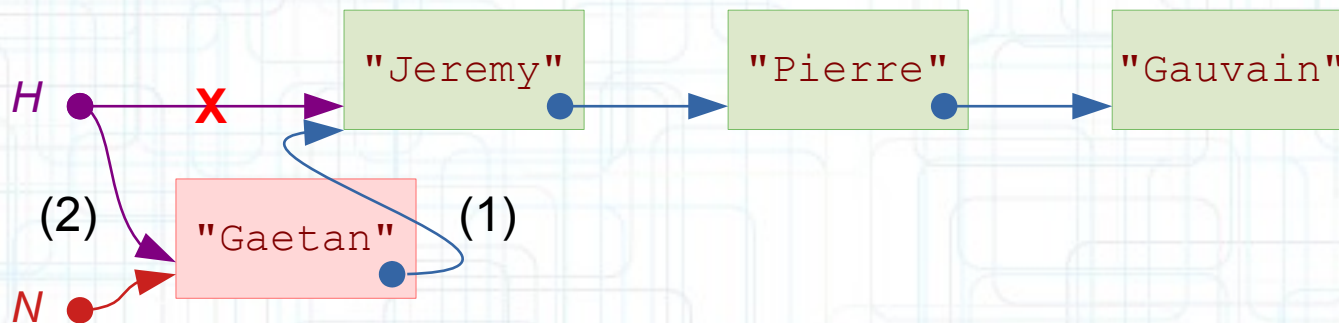
- Une liste chaînée peut être définie par une *relation de récurrence*. Une liste l est
 - soit une liste vide $[]$
 - soit (n, r) où n est le nœud de tête suivi d'une liste r .
- Exemple : la liste 1, 2, 3 pourrait être écrite $(1, (2, (3, [])))$.
- Cela se reflète également dans les relations entre classes. Supposons que la classe `Node` soit utilisée pour représenter un nœud d'une liste chaînée. La classe `Node` contient une référence vers un objet de type `Node`.



Listes Chaînées

- **Principe**

- Ajout d'un élément en tête de liste

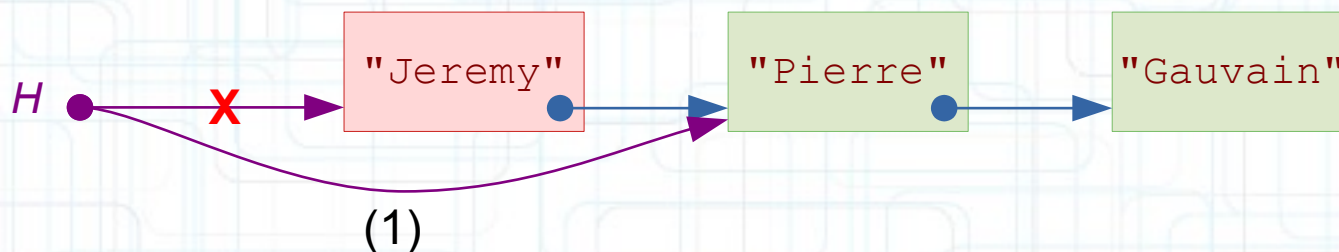


soit N nouveau nœud

(1) $N.\text{suiv} = H$

(2) $H = N$

- Suppression de l'élément en tête de liste



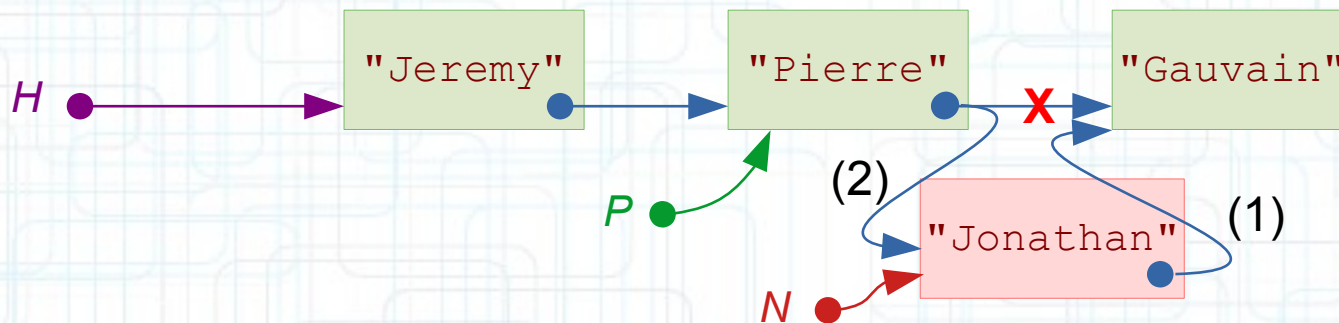
(1) $H = H.\text{suiv}$

- Ces deux opérations sont réalisées en temps constant.

Listes Chaînées

- **Principe**

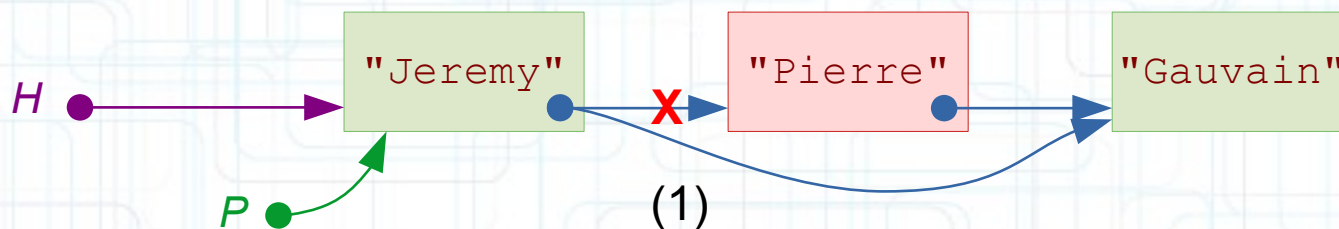
- Ajout d'un élément en « milieu » de liste



soit N nouveau nœud
 P prédécesseur

- (1) $N.\text{suiv} = P.\text{suiv}$
- (2) $P.\text{suiv} = N$

- Suppression d'un élément en « milieu » de liste



soit P prédécesseur

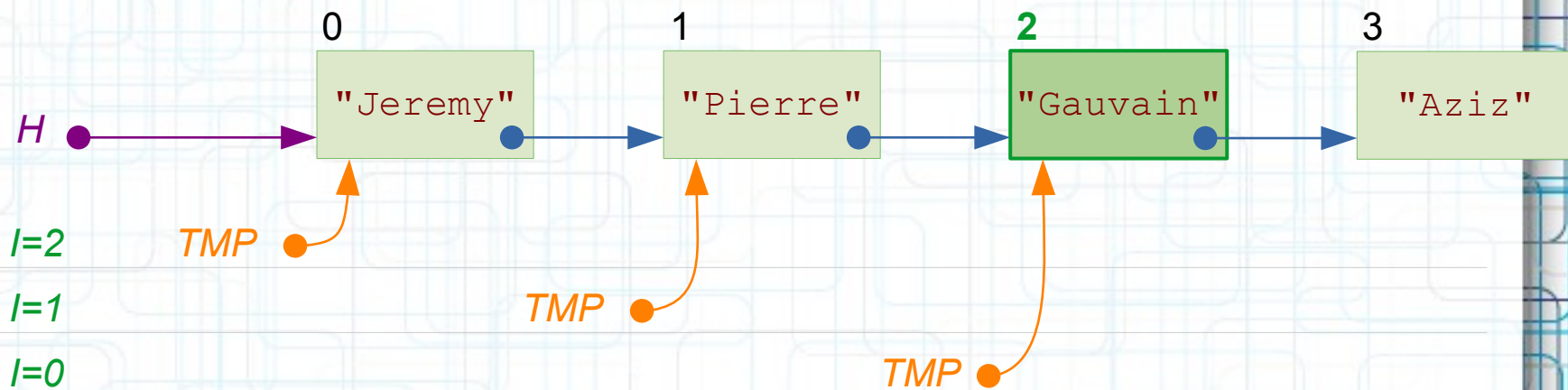
- (1) $P.\text{suiv} = P.\text{suiv}.\text{suiv}$

- Ces deux opérations sont réalisées en temps constant... à condition de connaître le prédécesseur P !

Listes Chaînées

- **Principe**

- Accès à un élément à une position arbitraire
- Soit i l'index de cet élément, $0 \leq i < N$



soit TMP , variable temporaire

$TMP = H$

tant que ($i > 0$) **faire**

$TMP = TMP.suiv$

$i = i - 1$

retourner $TMP.element$

Listes Chaînées

- **Principe**

- Rechercher la position d'un élément
- Soit **E** l'élément à rechercher
- Si l'élément n'est pas trouvé, la position retournée est -1

soit **TMP**, variable temporaire
I, index

TMP = **H**

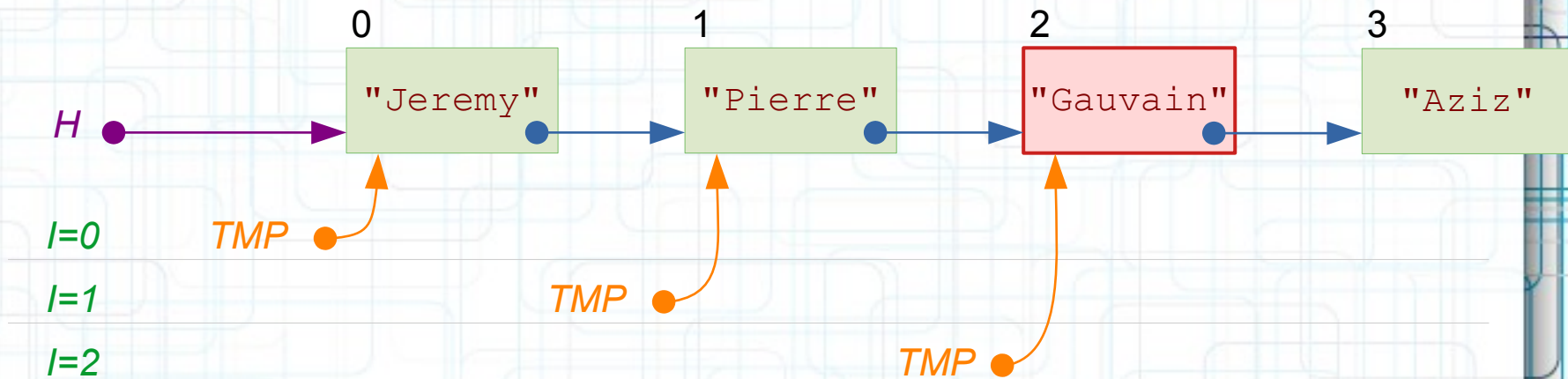
I = 0

tant que (**TMP** != null) **faire**
 si (**E** = **TMP**.element) **alors**
 retourner I

TMP = **TMP**.suiv

I = **I** + 1

retourner -1



Listes Chaînées

- **Principe**

- Ajouter/supprimer un élément en fin de liste ?
- Ajouter/supprimer un élément à une position arbitraire ?
- Déterminer la longueur de la liste ?
- Améliorations possibles
 - Maintenir la **longueur de la liste dans une variable** permet de diminuer le temps d'obtention de la longueur
 - Maintenir une **référence vers la fin** de la liste permet d'ajouter efficacement à la fin
 - Liste **doublement chaînée** : permet aussi de supprimer efficacement à la fin ; permet de diminuer le temps moyen d'accès par deux (en commençant par la tête ou la fin selon les cas)

Listes Chaînées

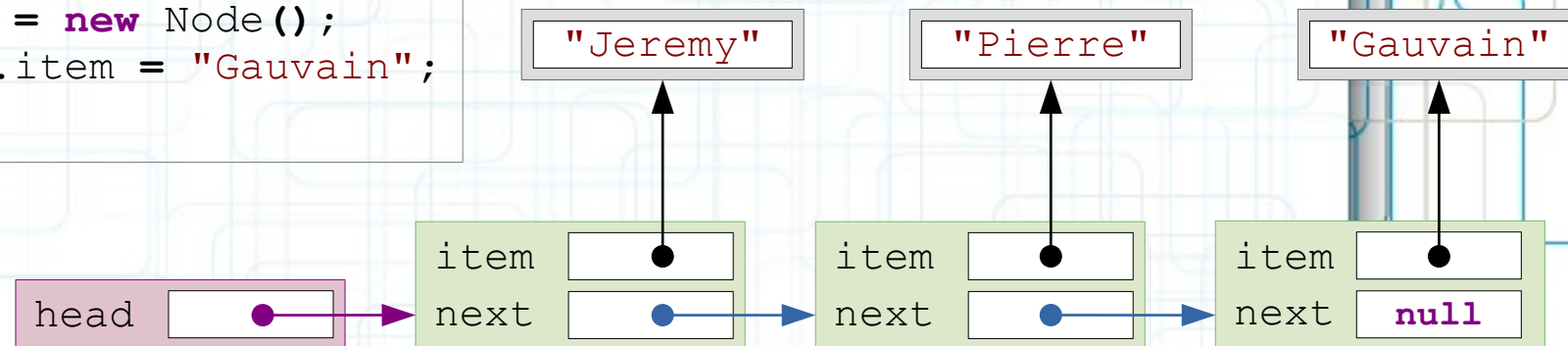
- **Implémentation**

- L'élément de base d'une liste, un nœud, est représenté à l'aide de la classe Node, composée de deux variables

- `item`, référence le **contenu**
- `next`, est le **lien vers le nœud suivant**

```
public class Node {  
    public Object item;  
    public Node next;  
}
```

```
public static void main(...) {  
    Node head = new Node();  
    head.item = "Jeremy";  
    head.next = new Node();  
    head.next.item = "Pierre";  
    head.next.next = new Node();  
    head.next.next.item = "Gauvain";  
}
```



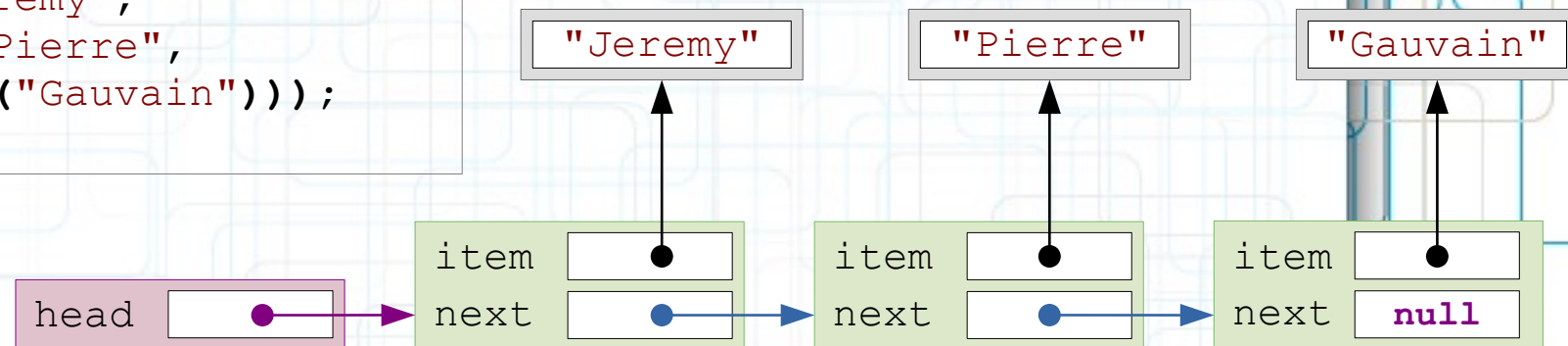
Listes Chaînées

- Implémentation - variante

- L'ajout de deux *constructeurs* à la classe Node permet de rendre la création de nœuds et de listes de nœuds très compacte.

```
public class Node {  
    public Object item;  
    public Node next;  
    public Node(Object item, Node next) {  
        this.item = item;  
        this.next = next;  
    }  
    public Node(Object item) {  
        this.item = item;  
    }  
}
```

```
public static void main(...) {  
    Node head =  
        new Node("Jeremy",  
            new Node("Pierre",  
                new Node("Gauvain")));  
}
```



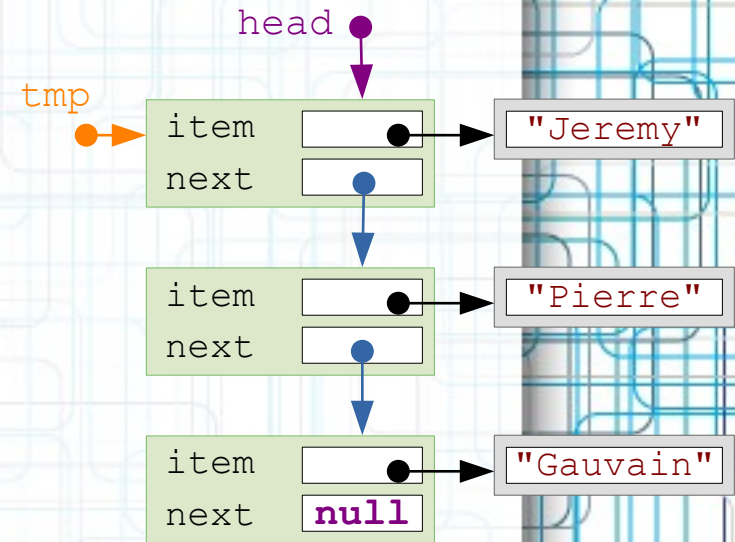
Listes Chaînées

- Implémentation

```
public static void main(...) {  
    Node head = new Node();  
    head.item = "Jeremy";  
    head.next = new Node();  
    head.next.item = "Pierre";  
    head.next.next = new Node();  
    head.next.next.item = "Gauvain";  
  
    Node tmp = head;  
    while (tmp != null) {  
        System.out.println((String) tmp.item);  
        tmp = tmp.next;  
    }  
}
```

➔ *Attention, perdre la tête (de liste)
est une erreur fréquente...*

```
while (head != null)  
    head = head.next;
```



Listes Chaînées

- **Encapsulation**

- La classe `LinkedList` permet de gérer l'ensemble de la liste
 - garder la référence vers la **tête**
 - maintenir la **taille** de la liste
 - rendre les services de l'ADT liste : ajout, suppression, accès aléatoire, etc.
- Cette approche permet d'assurer une **bonne encapsulation**.

```
public class LinkedList {  
    private class Node {  
        public Object item;  
        public Node next;  
    }  
    private Node head;  
    private int size;  
    (...)  
}
```

+ instances
de Node pas
accessibles
de l'extérieur

```
public static void main(...) {  
    LinkedList l = new LinkedList();  
    l.add("Jeremy");  
    l.add("Gauvain");  
    l.add(1, "Pierre");  
  
    for (int i = 0; i < l.size(); i++)  
        System.out.println(l.get(i));  
}
```


Listes Chaînées

- Implémentation – Insertion en tête

```
public class LinkedList {  
    (...)  
    public void add(Object item) {  
        Node n = new Node();  
        n.item = item;  
        n.next = head;  
        head = n;  
        size++;  
    }  
    (...)  
}
```

Toutes les méthodes de manipulation de liste qui suivent sont situées dans la classe `LinkedList`. Cependant, pour des raisons de place, cela ne sera pas indiqué explicitement.

Listes Chaînées

- Implémentation – accès aléatoire

recherche
du nœud i

```
public Object get(int i) {  
    if ((i < 0) || (i >= size))  
        throw new IndexOutOfBoundsException();  
  
    Node tmp = head;  
    while (i-- > 0)  
        tmp = tmp.next;  
  
    return tmp.item;  
}
```

```
public void set(int i, Object item) {  
    if ((i < 0) || (i >= size))  
        throw new IndexOutOfBoundsException();  
  
    Node tmp = head;  
    while (i-- > 0)  
        tmp = tmp.next;  
  
    tmp.item = item;  
}
```

Listes Chaînées

- Insertion à une position arbitraire

```
public void add(int i, Object item) {  
    if ((i < 0) || (i > size))  
        throw new indexOutOfBoundsException();  
  
    if (i == 0) {  
        Node n = new Node();  
        n.item = item;  
        n.next = head;  
        head = n;  
        size++;  
        return;  
    }
```

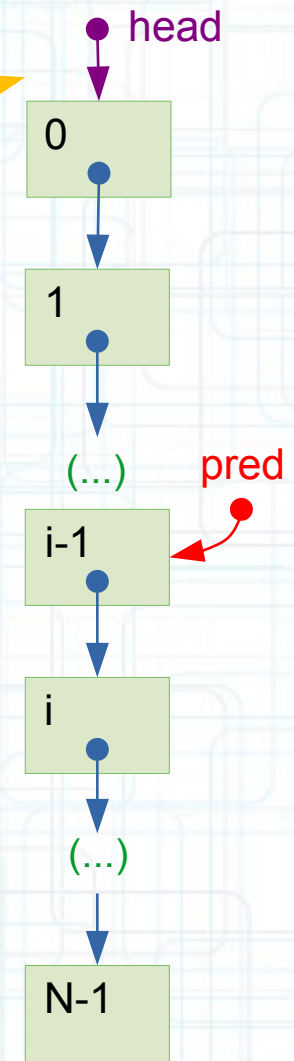
recherche du
prédécesseur
du nœud i

```
    Node pred = head;  
    while (i-- > 1)  
        pred = pred.next;
```

```
    Node n = new Node();  
    n.item = item;  
    n.next = pred.next;  
    pred.next = n;  
    size++;  
}
```

insertion
en tête
 $i = 0$

insertion
dans
la suite
 $i \in \{1, \dots, N\}$



Listes Chaînées

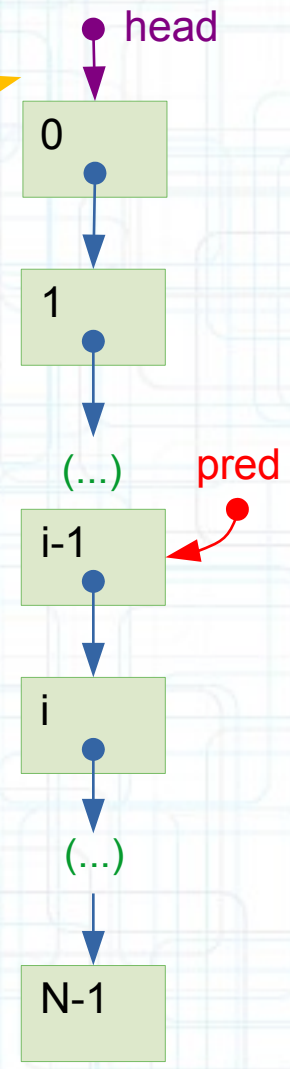
- **Suppression à une position arbitraire**

```
public void remove(int i) {  
    if ((i < 0) || (i >= size))  
        throw new indexOutOfBoundsException();  
  
    if (i == 0) {  
        head = head.next;  
        size--;  
        return;  
    }  
  
    Node pred = head;  
    while (i-- > 1)  
        pred = pred.next;  
  
    pred.next = pred.next.next;  
    size--;  
}
```

recherche du
prédécesseur
du nœud i

suppression
de la tête
i = 0

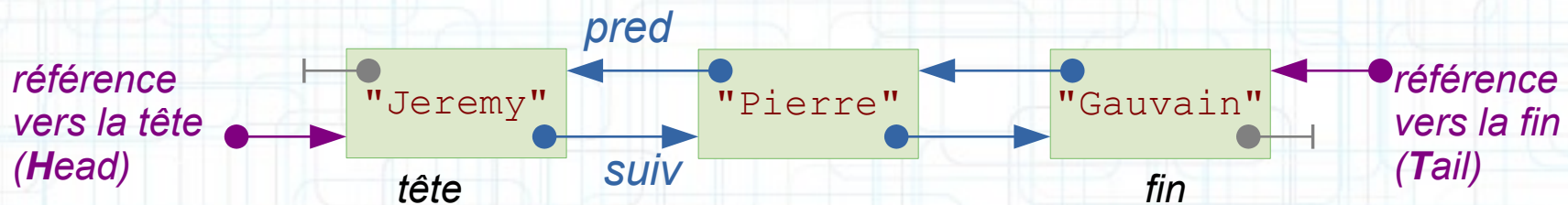
suppression
d'un élément
dans la suite
 $i \in \{1, \dots, N-1\}$



Listes Chaînées

- **Liste doublement chaînée**

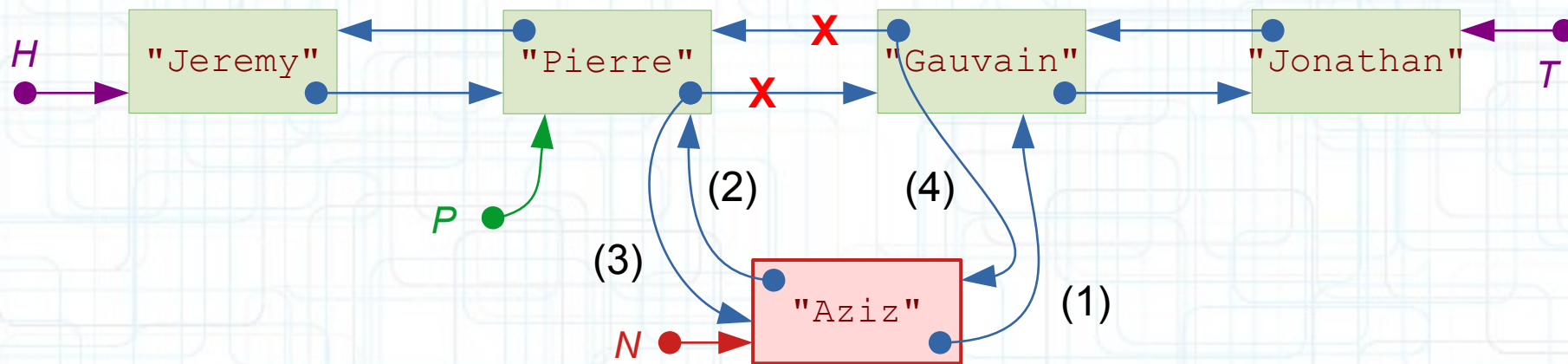
- Une **liste doublement chaînée** est composée de nœuds qui possèdent deux liens : l'un vers leur successeur (noté *suiv*), l'autre vers leur prédécesseur (noté *pred*).



- Les avantages d'une liste doublement chaînée sont
 - permet l'insertion et la suppression efficaces en fin de liste
 - permet de se déplacer d'avant en arrière dans la liste
 - divise par deux le temps d'accès dans le pire des cas (en commençant par la tête ou la fin, la moitié de la liste est à parcourir)

Listes Chaînées

- Liste doublement chaînée - Insertion



soit N nouveau nœud
 P prédécesseur

quid en début de liste ?

- (1) $N.\text{suiv} = P.\text{suiv}$
- (2) $N.\text{pred} = P$
- (3) $P.\text{suiv} = N$
- (4) $N.\text{suiv}.\text{pred} = N$

quid en fin de liste ?

Listes Chaînées

- **ArrayList VS LinkedList**

- ArrayList et LinkedList sont deux implémentations de l'ADT Liste. Il est important de connaître leurs différences.

	ArrayList	LinkedList
ajout/suppression en fin	$O(1)$ en moyenne	$O(1)$
ajout/suppression en tête	$O(N)$	$O(1)$
insertion / suppression	$O(N)$	$O(N)$ $O(1)$ si préd./succ. connu ⁽¹⁾
taille	$O(1)$	$O(1)$
parcours séquentiel	$O(N)$	$O(N)$
accès aléatoire	$O(1)$	$O(N)$
surcoût mémoire par élément	0 (attention au taux d'occupation)	2 références (chacune de 32/64-bits)

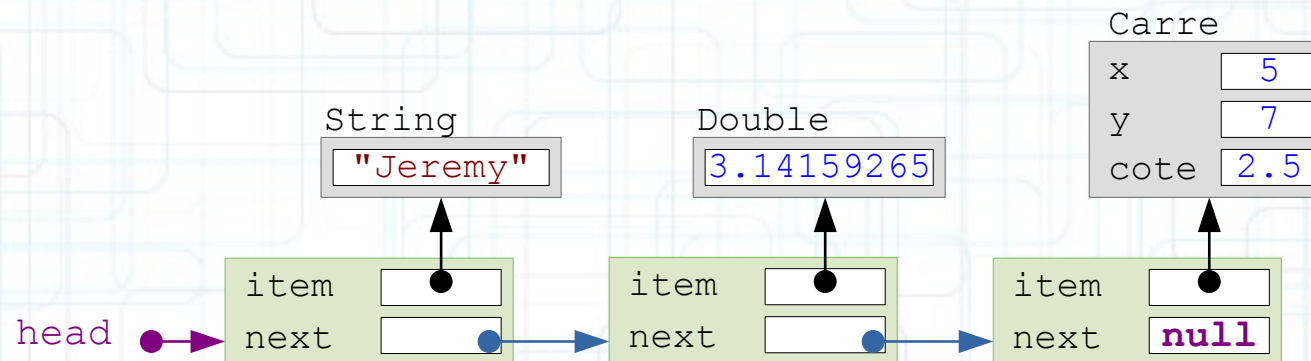
Listes Chaînées

- **Polymorphisme**

- Dans l'implémentation proposée, les *éléments* sont conservés dans les nœuds à l'aide de références de type `Object`.

```
public class Node {  
    public Object item;  
    public Node next;  
}
```

- Il s'agit de références *polymorphiques*, ce qui veut dire qu'elles peuvent référencer des instances de n'importe quelle classe en relation *is-a* avec `Object` \Rightarrow toutes les instances



Listes Chaînées

- **Polymorphisme**

- Une autre approche se base sur le *polymorphisme paramétrique*. Dans l'exemple ci-contre, les éléments doivent être en relation **is-a** avec `E`, le paramètre de type.

- Attention, cependant lorsque `Node` devient une classe interne de `LinkedList`, le paramètre de type `E` doit seulement être défini sur la classe externe⁽¹⁾.

```
public class Node<E> {  
    public E    item;  
    public Node next;  
}
```

```
public class LinkedList<E> {  
    private class Node {  
        public E    item;  
        public Node next;  
    }  
  
    private Node head;  
    private int  size;  
  
    public void add(E item);  
    public E get(int i);  
  
    (...)  
}
```

⁽¹⁾ la raison sera expliquée dans le chapitre portant sur les « generics », le polymorphisme paramétrique de Java..

Table des Matières

1. Introduction
2. *Java Collections Framework*
3. Tableau dynamiques
4. Listes chaînées
- 5. Tables de hachage**

Table de hachage

- Introduction

- Une **table de hachage** est une implémentation de l'ADT Association (*map*). Pour rappel, une association maintient des paires clés / valeurs (k, v). Elle fournit deux opérations
 - $\text{put}(k, v)$
 - $\text{get}(k) \rightarrow v$
- Une table de hachage utilise typiquement
 - un tableau de N cellules pour stocker les associations.
 - une fonction de hachage pour déterminer dans quelle cellule du tableau une association doit être stockée.
 - des listes de collisions

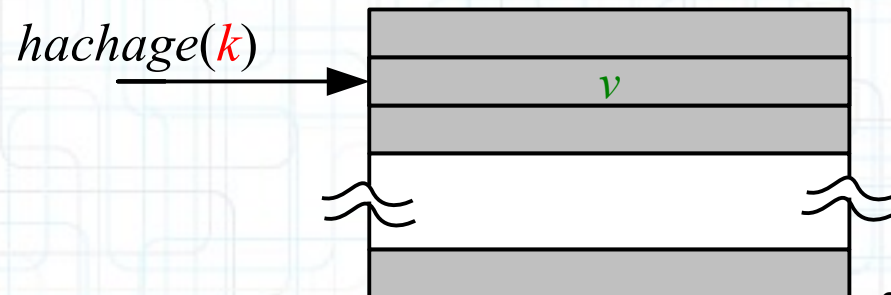


Table de hachage

- **Fonction de hachage**

- Une **fonction de hachage** prend une clé k (de type `Object`) en argument et retourne un entier $h(k)$.
 - $h: \text{Object} \rightarrow \text{int} : k \rightarrow h(k)$
- Un object k doit toujours avoir la même image $h(k)$!!
- $h(k)$ doit être « facile à calculer » \Rightarrow put et get sont en $O(1)$
- Comme $|\text{Im}(h)| > N$, l'index du tableau est calculé $\text{mod } N$

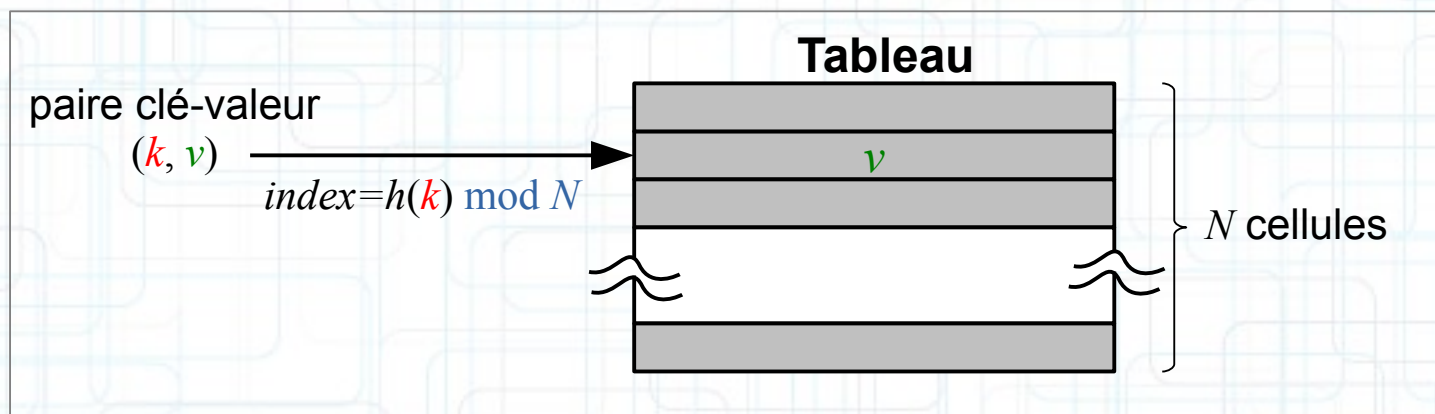


Table de hachage

- Première approche...

```
public class HashMap {  
  
    public static final int TAB_SIZE = 1024;  
  
    private Object [] tab = new Object[TAB_SIZE];  
  
    public void put(Object k, Object v) {  
        tab[h(k) % TAB_SIZE] = v;  
    }  
  
    public Object get(Object k) {  
        return tab[h(k) % TAB_SIZE];  
    }  
}
```

► Mais d'où vient la fonction h ?

Table de hachage

- **Fonction de hachage en Java**

- En Java, la fonction de hachage est définie dans la classe `Object` sous forme de la méthode
 - `public int native hashCode()`
- Cette méthode retourne un entier différent pour chaque objet, typiquement sur base de l'adresse en mémoire de l'objet.
- Cependant, cette méthode **peut être redéfinie** dans les sous-classes de façon à retourner une valeur identique pour des objets de contenus égaux.

Note : le mot clé `native` signifie que la méthode n'est pas implémentée en Java mais directement par la JVM (en langage C).

Table de hachage

- Fonction de hachage en Java

```
public class SomeObject {  
    public final int x, y;  
    public SomeObject(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public boolean equals(Object o) {  
        return (x == o.x) && (y == o.y);  
    }  
}
```

```
Object o1 = new SomeObject(4, 5);  
Object o2 = new SomeObject(4, 5);  
System.out.println(o1 == o2);  
System.out.println(o1.equals(o2));  
System.out.println(o1.hashCode());  
System.out.println(o2.hashCode());  
Map m = new HashMap();  
m.put(o1, "toto");  
System.out.println(m.get(o2));
```

hashCode
hérité
de Object

→ **false** (références différentes)

→ **true** (contenus identiques)

→ 1020391880

→ 977993101

**hash codes
différents !**

→ **null** (clé inconnue)

Table de hachage

- Redéfinition de hashCode

```
String s1 = "plop";
String s2 = s1.substring(0, 3) + 'p';
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
Map m = new HashMap();
m.put(s1, 42);
System.out.println(m.get(s2));
```

hashCode
redéfinie
dans String

→ **false** (références différentes)

→ **true** (contenus identiques)

→ 3443933

→ 3443933

**hash codes
identiques !**

→ 42 (clé connue)

- le hachage d'une chaîne de N caractères s est calculée comme suit⁽¹⁾

$$h(s) = \sum_{i=0}^{N-1} \left(31^{N-i-1} \times s.charAt(i) \right)$$

- Exemples

• "a" → 97

• "b" → 98

• "plop" → 3443933 (112 x 31³ + 108 x 31² + 111 x 31 + 112)

• "aa" → 3104 (97 x 31 + 97)

• "bB" → 3104 (98 x 31 + 66)

(1) voir http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4045622

Table de hachage

- **Redéfinition de hashCode**

- **Integer** : le hachage d'un entier est l'entier lui-même.
- **String** : le hachage d'une chaîne de N caractères s est calculée comme suit⁽¹⁾

$$h(s) = \sum_{i=0}^{N-1} \left(31^{N-i-1} \times s.charAt(i) \right)$$

- Object composé : le hachage d'un objet composé de N objets v_0 à v_{N-1} peut être calculé à partir du hachage des object v_i comme suit

$$h(s) = \sum_{i=0}^{N-1} \left(31^i \times h(v_i) \right)$$

- cette approche peut être déléguée à la méthode `System.util.Objects.hash(Object... values)`

(1) voir http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4045622

Table de hachage

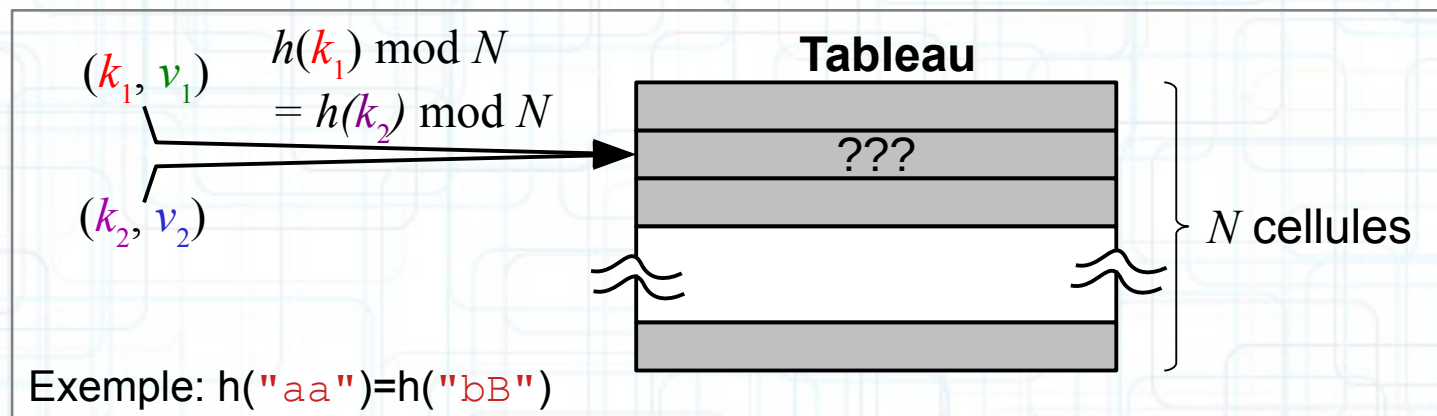
- **Redéfinition de hashCode**

- Il est nécessaire de respecter les règles suivantes lorsque l'on redéfinit la méthode `hashCode`
 - **Hachage en accord avec l'égalité**
 - si `a.equals(b)`
alors `a.hashCode() == b.hashCode()`
 - la réciproque ne sera pas toujours vraie
 - **Fonction de hachage déterministe** : pour une même valeur d'instance, toujours retourner le même hachage
 - **Attention aux objets mutables** : ne peuvent être utilisés comme clé si le hachage peut changer
 - **Fonction de hachage rapide à calculer** (éventuellement mettre la valeur calculée en "cache")

Table de hachage

- **Collisions**

- La fonction de hachage n'est généralement **pas injective**.
- Il est donc possible que deux clés différentes k_1 et k_2 donnent
 - la même image : $h(k_1) = h(k_2)$
 - la même position dans le tableau : $h(k_1) \bmod N = h(k_2) \bmod N$.
- Cette situation est appelée **collision**.

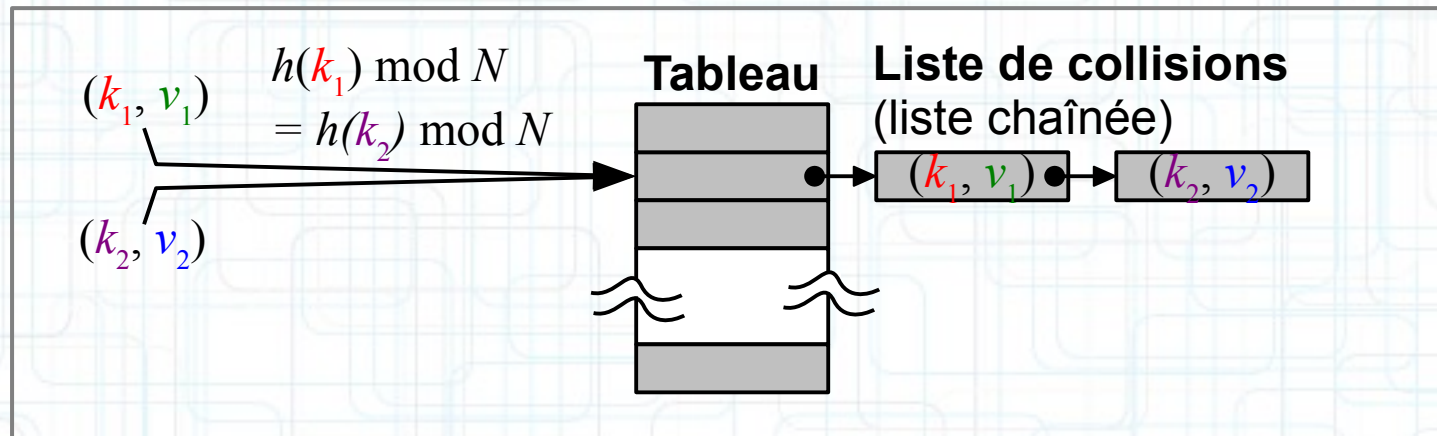


Rappel : une fonction f est *injective* si $\forall x, y, f(x) = f(y) \Rightarrow x = y$

Table de hachage

- **Liste de collisions**

- Les collisions sont gérées en stockant les associations dont la clé donne la même position dans une **liste de collisions**.



- La performance de l'accès à un élément de la table de hachage dépend alors de la longueur des listes de collisions. Etant donnés M éléments et un tableau de N cellules,
 - idéal : éléments répartis uniformément. Accès en $O(M / N)$.
 - pire des cas : tous les éléments sont dans la même liste. Accès en $O(M)$.

Table de hachage

- Liste de collisions

```
public class HashMap {
```

```
    private class HashEntry {  
        public Object k, v;  
        public HashEntry(Object k, Object v) {  
            this.k = k;  
            this.v = v;  
        }  
    }  
}
```

→ Classe interne représentant un couple clé-valeur (k , v)

```
public static final int TAB_SIZE = 1024;
```

```
@SuppressWarnings("undefined")
```

```
private List<HashEntry> [] tab; =  
    new LinkedList<>[TAB_SIZE];
```

```
/* suite au slide suivant... */
```

→ Nécessaire car création de tableau « générique »

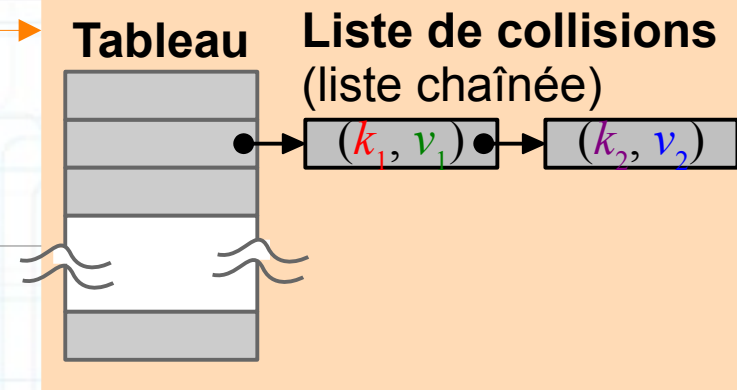


Table de hachage

- Liste de collisions

```
public void put(Object k, Object v) {  
    int i = k.hashCode() % TAB_SIZE;  
    if (tab[i] == null)  
        tab[i] = new LinkedList<>();  
    for (HashEntry he: tab[i])  
        if (he.k.equals(k)) {  
            he.v = v;  
            return;  
        }  
    tab[i].add(new HashEntry(k, v));  
}
```

Création de la liste
lors de l'ajout du 1^{er} élément

Remplacement si clé déjà
existante

Insertion nouvelle paire
dans la liste

```
public Object get(Object k) {  
    int i = k.hashCode() % TAB_SIZE;  
    if (tab[i] == null)  
        return null;  
    for (HashEntry he: tab[i])  
        if (he.k.equals(k))  
            return he.v;  
    return null;  
}
```

liste vide

Recherche clé dans la liste

clé inconnue

Table de hachage

- Liste de collisions

- Quelques précautions sont nécessaires lors du calcul de l'index du tableau avec $h(k) \bmod N$.
- L'utilisation de l'opérateur `%` n'est pas correct car en Java le reste retourné peut être négatif lorsque le `hashCode` est négatif.
- Un moyen simple efficace consiste à prendre N égal à une puissance de 2 et calculer `k.hashCode() & (N - 1)`
- Exemples avec $N=1024$ (2^{10})

- $h(\text{"plop"}) = 3443933$

- $h(\text{"carambar"}) = -10275565$

1101001000110011011101	$h(k)$
$\&$ 1111111111	$(N-1)$
0011011101	221

11111111011000110011010100010011	$h(k)$
$\&$ 1111111111	$(N-1)$
0100010011	275

Table de hachage

- **Les collisions sont-elles fréquentes ?**
 - Soit N le nombre de cellules du tableau
 M le nombre d'éléments à stocker
 - Supposons que pour chaque élément à stocker, la cellule du tableau soit choisie par la fonction de hachage **de façon aléatoire**, avec la même probabilité pour chaque cellule : $1 / N$
 - Quelle est la probabilité $P_{\text{collision}}$ d'avoir au moins une collision ?
 - Pour des raisons techniques, il est plus facile de déterminer la probabilité $P_{\text{no collision}}$ de n'avoir aucune collision. On peut ensuite déterminer $P_{\text{collision}} = 1 - P_{\text{no collision}}$

$N=4$

1/4

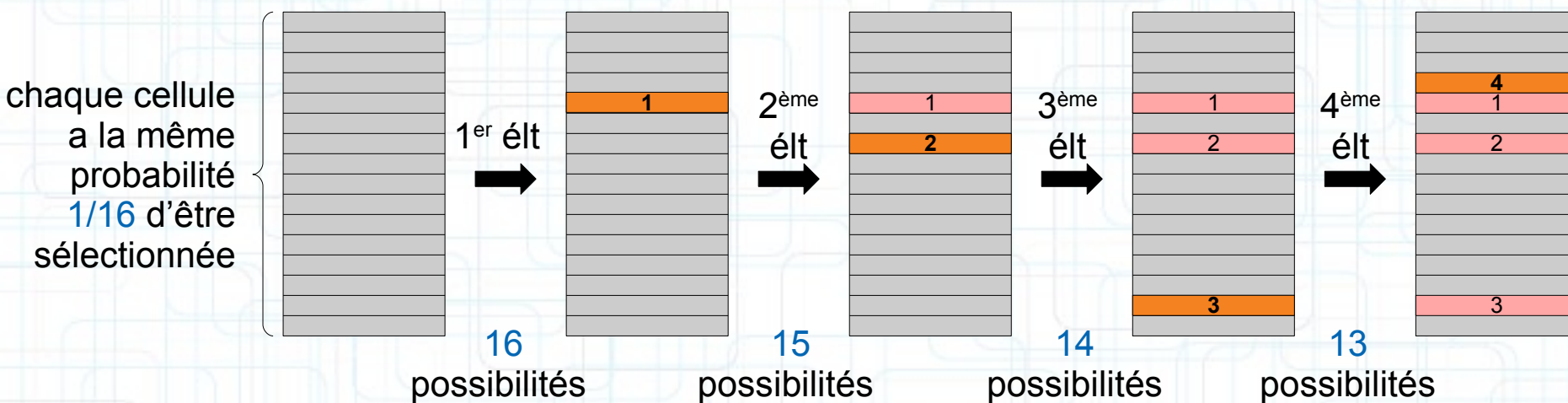
1/4

1/4

1/4

Table de hachage

- Les collisions sont-elles fréquentes ?
 - Exemple avec un tableau de $N=16$ cellules dans lequel placer les $M=4$ éléments suivants : $\{ 1, 2, 3, 4 \}$.
 - Quelle est la probabilité de **ne pas avoir de collision** ?



$$P_{\text{no collision}}(N, M) = \frac{16}{16} \times \frac{15}{16} \times \frac{14}{16} \times \frac{13}{16} \approx 0,666$$

$$P_{\text{collision}}(N, M) = 1 - P_{\text{no collision}}(N, M) \approx 0,333$$

Table de hachage

- **Les collisions sont-elles fréquentes ?**

- De manière générale,

- si $M > N$, $P_{\text{collision}} = 1$

- sinon,

$$\begin{aligned} P_{\text{collision}}(N, M) &= 1 - \left(\frac{N}{N} \times \frac{N-1}{N} \times \dots \times \frac{N-M+1}{N} \right) \\ &= 1 - \frac{N!}{(N-M)! N^M} \end{aligned}$$

- On peut conclure que, même avec un petit nombre d'éléments à stocker en regard de la taille du tableau, **la probabilité d'avoir deux éléments ou plus dans la même cellule peut être élevée.**

Table de hachage

- **Les collisions sont-elles fréquentes ?**
 - Autre application : probabilité qu'au moins 2 personnes dans un groupe de 25 aient le même jour d'anniversaire ?

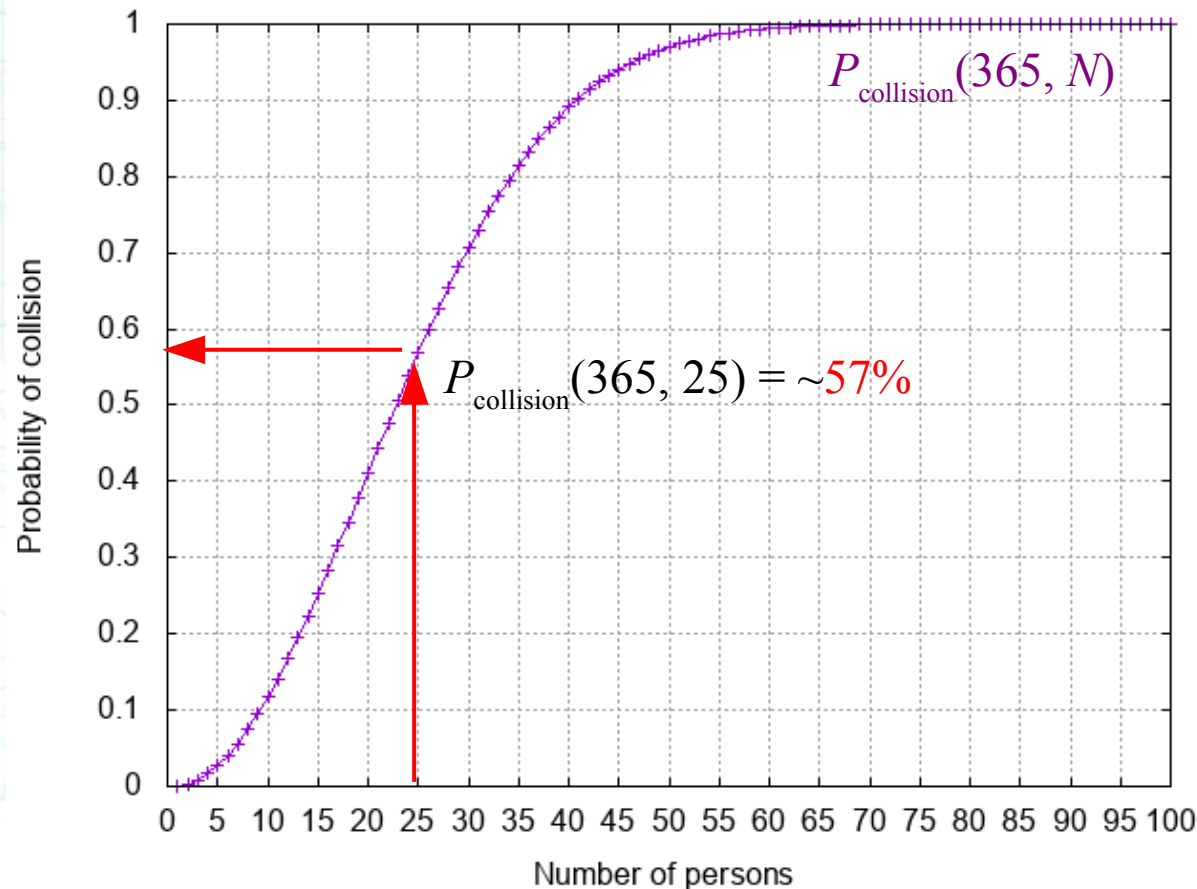


Table de hachage

- **Expérience**

- Cette expérience a pour objectif de déterminer la qualité du hachage de chaînes de caractères (`String`) en Java, c.-à-d. comment un ensemble de chaînes de caractères est réparti sur les cellules d'un tableau ?
- Deux ensembles de données sont utilisés
 - EN : Une liste de $M=194433$ mots anglais
 - FR : Une liste de $M=208916$ mots français
- Des tableaux de tailles différentes sont utilisés
 - $N = 131072$ (2^{17})
 - $N = 262144$ (2^{18})
 - $N = 524288$ (2^{19})

Nombre de
cellules
contenant x
entrées

Table de hachage

Nombre moyen
d'entrées par
cellule (N/M).

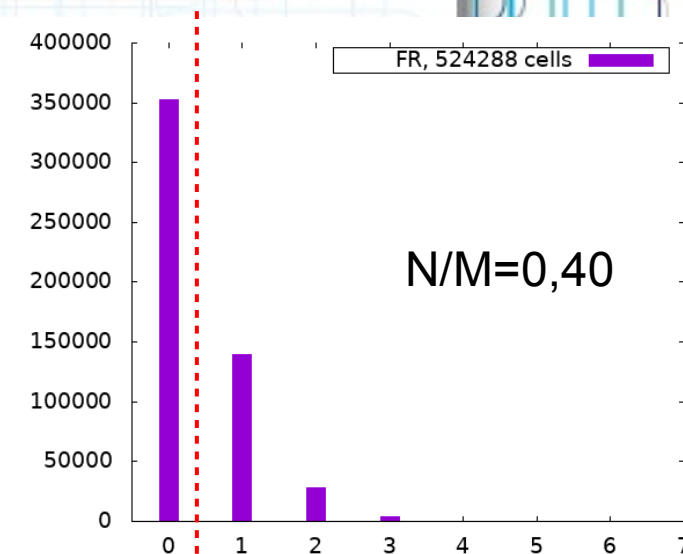
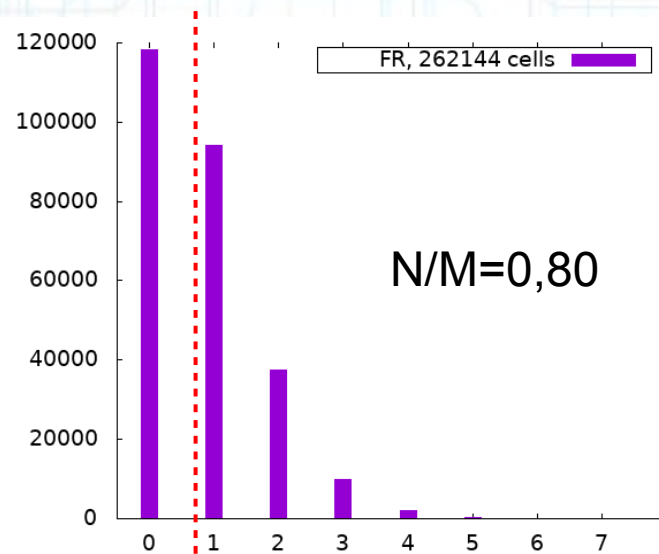
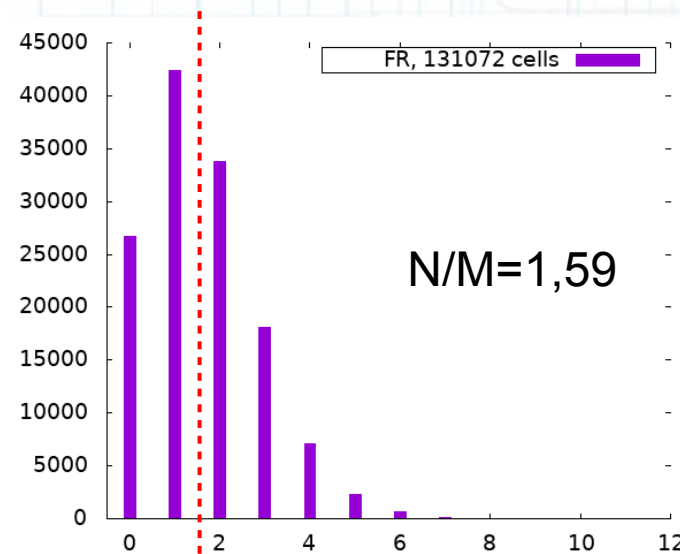
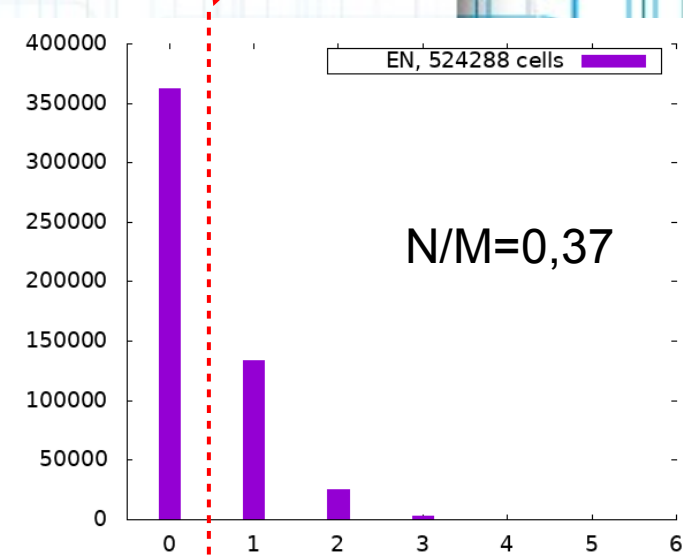
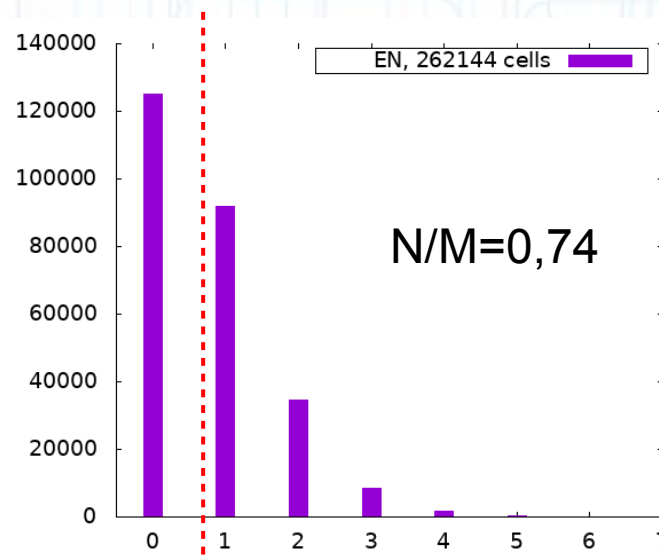
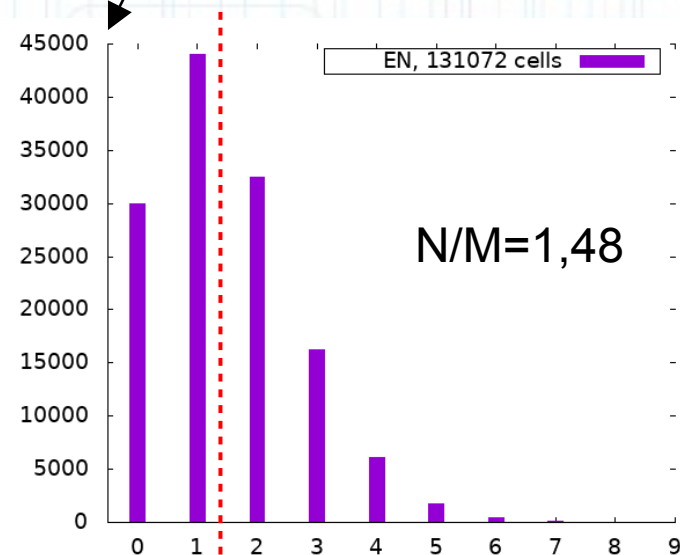


Table de hachage

- **Ré-allocation automatique**

- Lorsque le nombre d'éléments dans une table de hachage croît, la **probabilité de collision augmente**.
- Pour cette raison, les tables de hachage de Java **redimensionnent** automatiquement le tableau sous-jacent lorsque leur taux d'occupation est trop élevé.
- A cette fin elles sont munies d'un paramètre *load factor* (l). Sa valeur par défaut est 75%.
- Soit N la taille du tableau, M le nombre d'éléments.
- Le taux d'occupation est M / N
- Le tableau est ré-alloué (typ. doublé) lorsque $M / N > l$