Structures de données linéaires

- tableaux,
- listes à accès direct,
- listes chaînées,
- table de hachage
- …

Tableaux

- Un exemple pour un tableau à 2 dimensions

```
String tab[][]={{"a", "e", "i", "o", "u"}, {"1", "2", "3", "4"}};

int i = 0, j = 0;

for(String sousTab[] : tab) {

  j = 0;

  for(String str : sousTab) {

    System.out.println("Valeur du tableau à l'indice ["+i+"]["+j+"]: " + tab[i][j]);

    j++;

  }

  i++;

}
```

```
Valeur du tableau à l'indice [0][0] : a
Valeur du tableau à l'indice [0][1] : e
Valeur du tableau à l'indice [0][2] : i
Valeur du tableau à l'indice [0][3] : o
Valeur du tableau à l'indice [0][4] : u
Valeur du tableau à l'indice [1][0] : 1
Valeur du tableau à l'indice [1][1] : 2
Valeur du tableau à l'indice [1][2] : 3
Valeur du tableau à l'indice [1][3] : 4
```

Exemple

Tableaux (suite)

- A peu près la même base qu'en C++ (Au moins en apparence)

- Déclaration

  typeDesElementsDuTableau[] nomDuTableau ;

  Taille du tableau non précisée à la déclaration

```
int[] tabEntiers ;
```

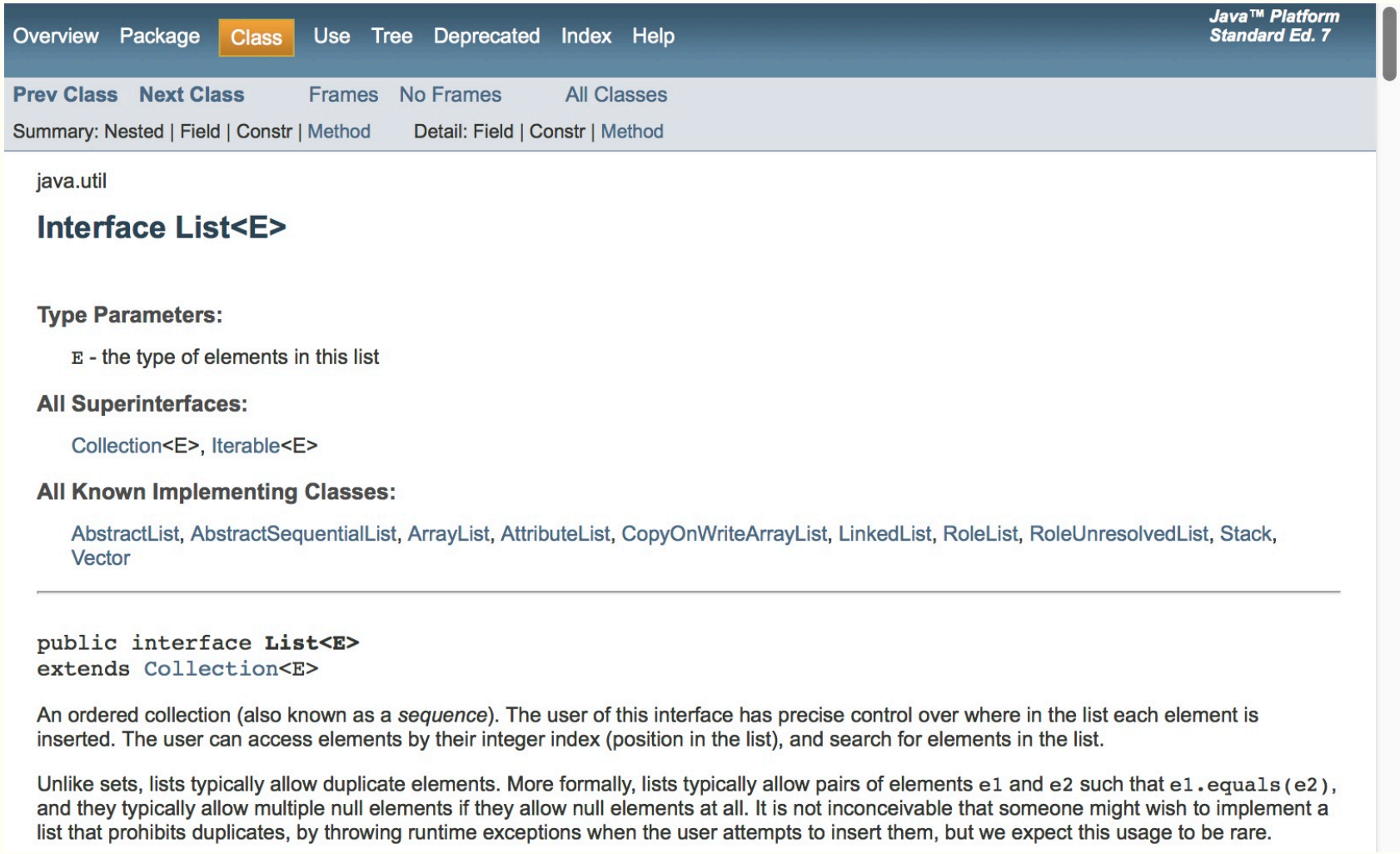
Exemple

- Contrairement à C++, les tableaux sont des objets
  Ils doivent être créés avec new

  nomDuTableau = new TypeDesElementsDuTableau[taille] ;

```
tabEntiers = new int[40] ;
// création effective du tableau précédent
```

Les listes



Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface List<E>

Type Parameters:

`E` - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

---

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

Listes (suite)

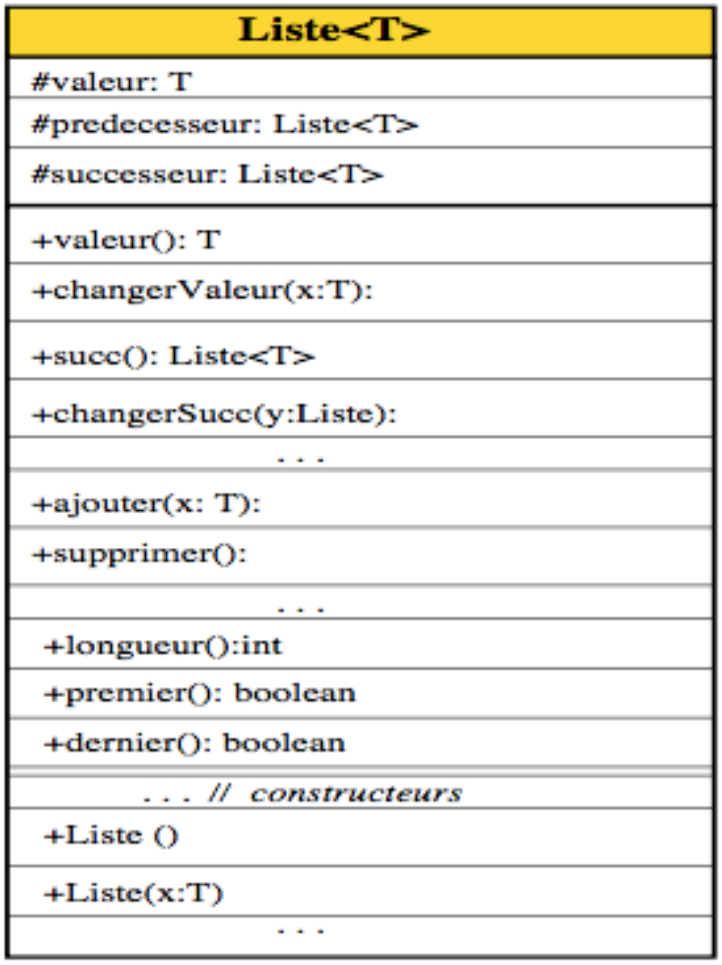
Les listes (`ArrayList` et `LinkedList`) :

Définition [liste]. *Ensemble fini d'éléments liés par une* « relation de **séquentialité** » *qui a les propriétés suivantes :*

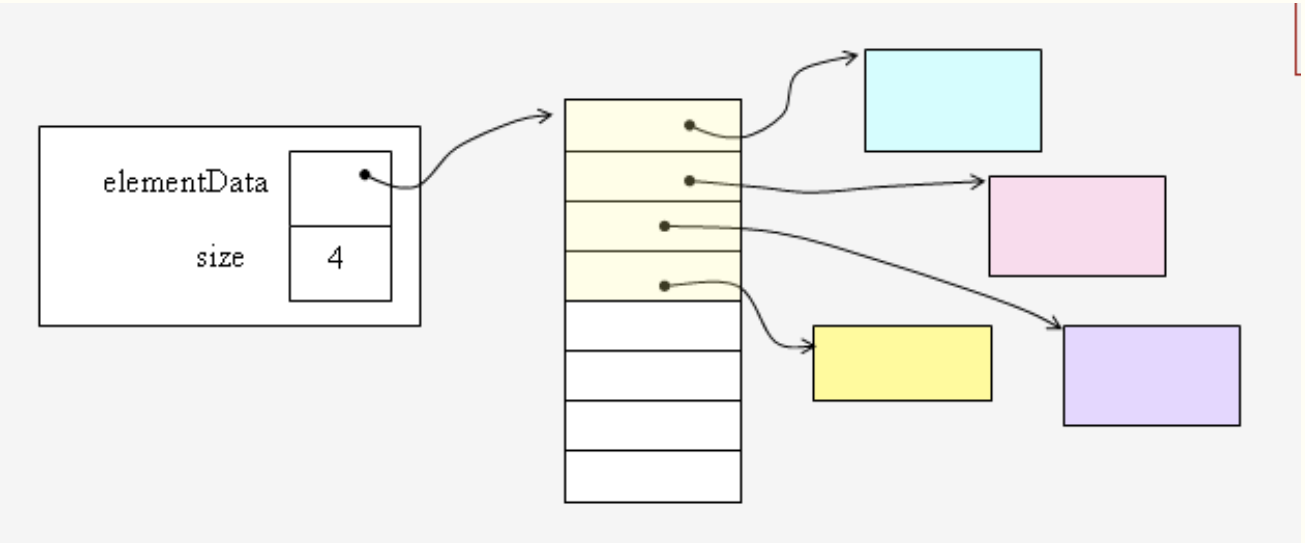
◁ *sauf aux extrémités, tout élément a un* prédécesseur *et un* successeur *;*

◁ *le premier élément (*la tête*) n'a pas de prédécesseur ;*

◁ *le dernier élément n'a pas de successeur.*

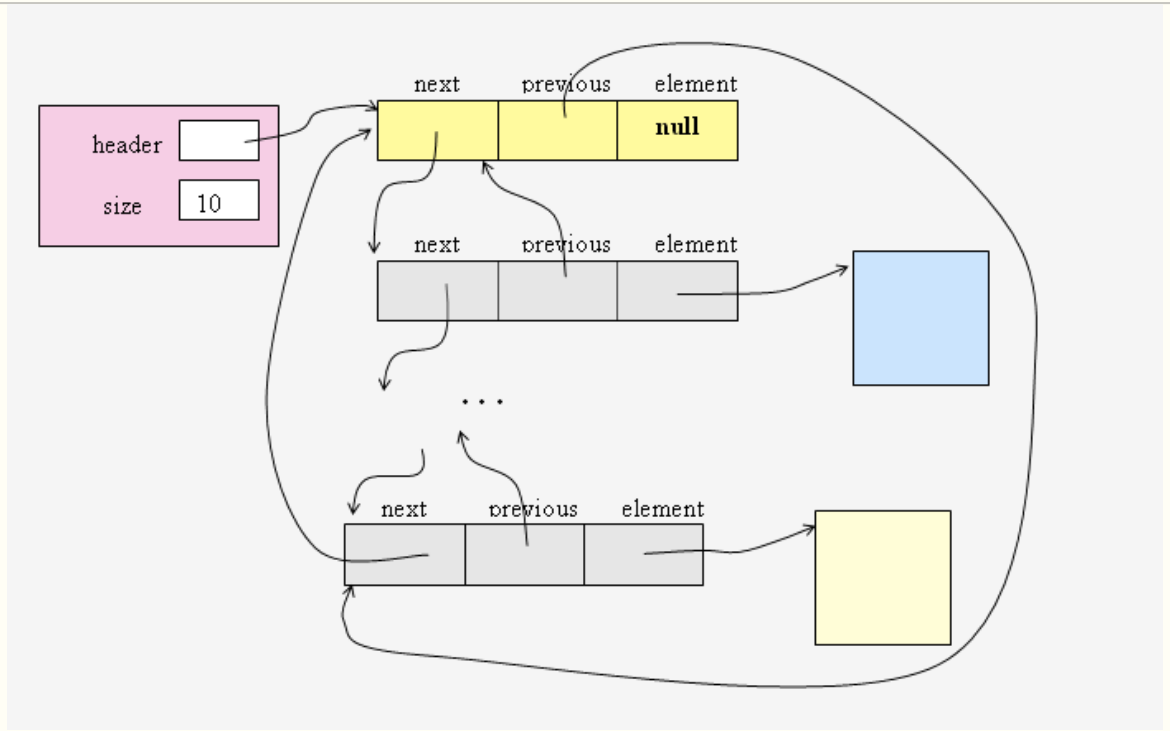


Listes à accès direct



- Accès immédiats (lectures) par un *indice*
- Insertion et suppression couteuses (écritures)
- C'est le cas des tableaux, vecteurs, …

Listes chainées



- Accès éventuellement longs
- Insertions et suppressions très rapides (lorsque c'est en tête)

Une implémentation des listes chaînées en Java

```
public class Liste<T> {
protected T valeur;
protected Liste<T> succ;
protected Liste<T> pred;

  public T valeur(){
    return valeur;
  }
  public void changerValeur(T x){
    valeur = x;
  }
  public Liste<T> succ(){
    return succ;
  }
  public void changerSucc(Liste<T> y){
    succ = y;
  }
  public void changerPred(Liste<T> y){
    pred = y;
  }
```

```
public Liste(){ // Constructeur de liste vide
pred = null; succ = null; }

public Liste(T x){// Constructeur de liste a un element
    valeur = x;
    pred = null;
    succ = null;
  }
  public void ajouter(T x){// insertion en 2e position
    Liste<T> y = new Liste(x);
    if (succ!=null) {succ.changerPred(y);}
    y.changerPred(this);
    y.changerSucc(succ);
    succ = y;
  }

  public void supprimer(){ // a faire (exercice) }
```

Exemple

Listes (suite): en Java

```
import java.util.*;

public class TestListe {

   public static void testListe(List<Integer> l) {
      //ajouter un nombre au hasard en fin de la liste
      for (int i=0 ; i<10000 ; i++) {
         int a = (int) Math.floor(Math.random()*1000);
         l.add(a);
      }

      System.out.println("taille de la liste : " +l.size());
      System.out.println("element d'indice 24 : " +l.get(24));

      //modifier le 24e en lui donnant la valeur 0
      l.set(23,0);

      // affichage
      for (Integer x : l) {
         //pour tous les entiers x de l
         System.out.print(x+" ");
      }
   }
```

Exemple

```
   public static void main(String[] arg){
     List<Integer> l1,l2;
     l1 = new ArrayList<Integer>();
     l2 = new LinkedList<Integer>();

     testListe(l1);
     testListe(l2);
   }

}
```

Comparaison d'utilisation des listes

Explications (pour les temps de suppression (types `List`)) :

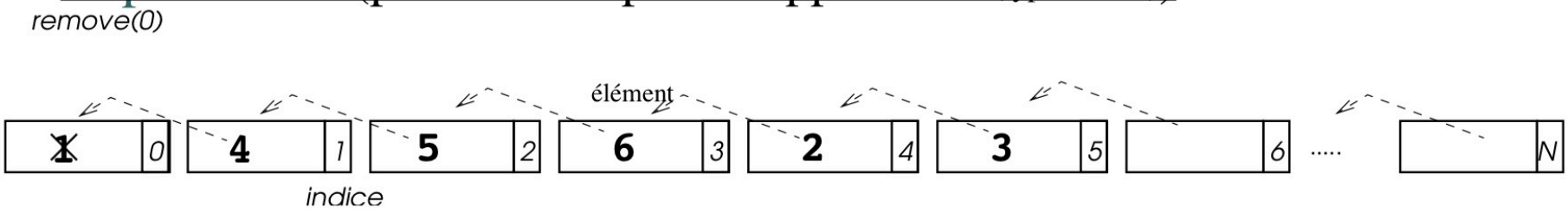


Figure 3 : Suppression dans un tableau

$\rightsquigarrow$ *N opérations d'affectations de la valeur d'indice* $i+1$ *dans la case d'indice* i

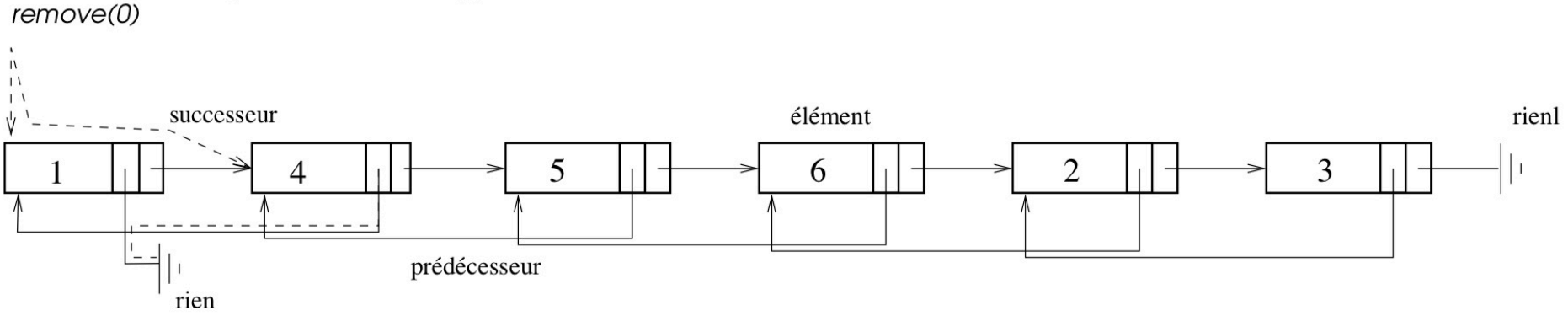


Figure 4 : Suppression dans une liste chaînée

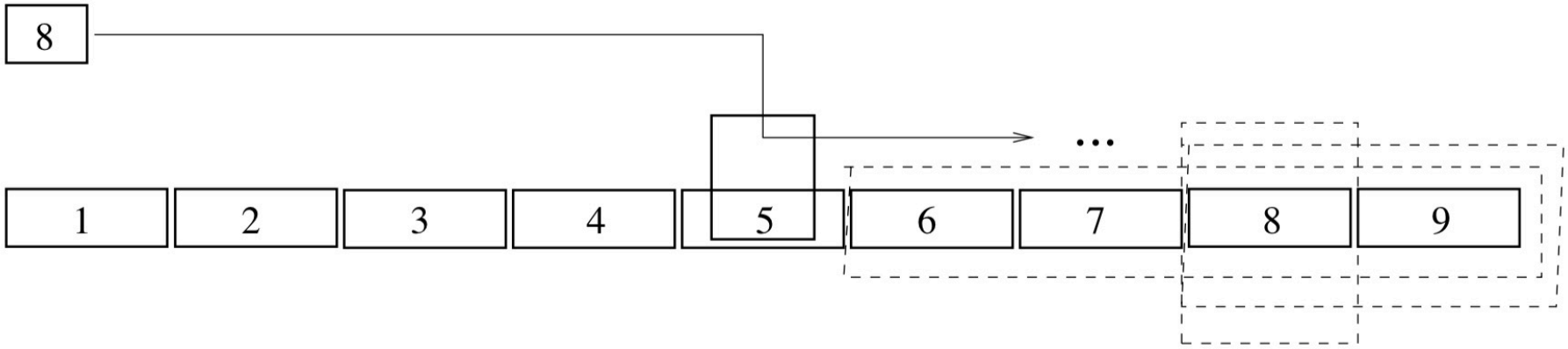
$\rightsquigarrow$ *2 opérations de modifications de chaînage sur les éléments*

La recherche dichotomique dans les listes ordonnées

Définition **(liste ordonnée.)** *Les élémenst* $\texttt{a}_\texttt{i}$ *de la liste* `l` *sont tels que :* $\forall \texttt{i}\ (\texttt{0} \leq \texttt{i} < \texttt{l.length} - \texttt{1}), a_i \leq a_{i+1}$. □

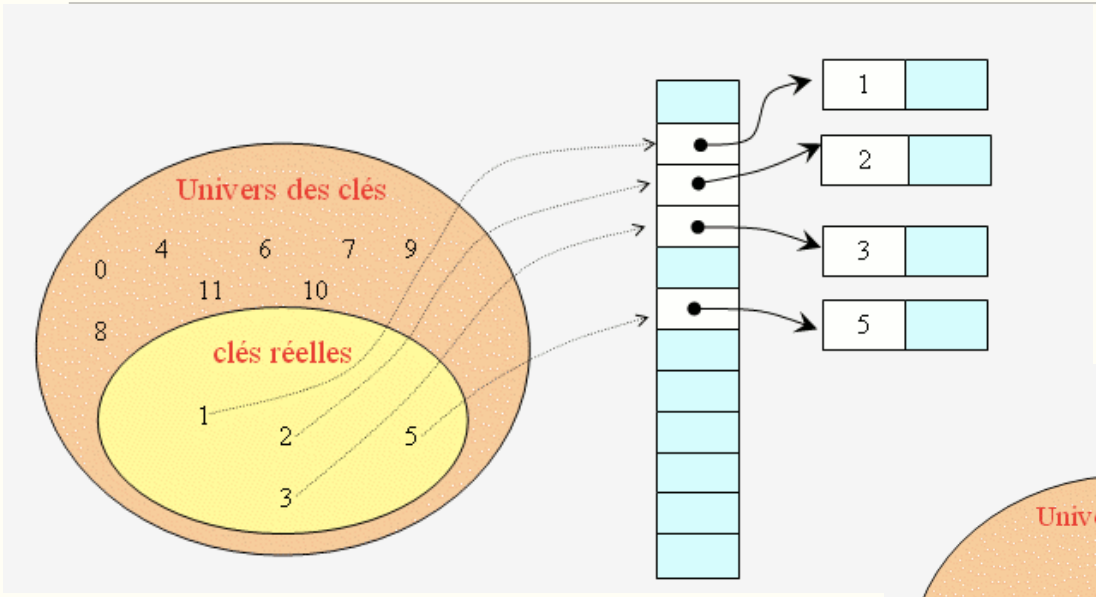
Motivations :

- la recherche d'un élément dans une liste ordonnée est plus rapide.

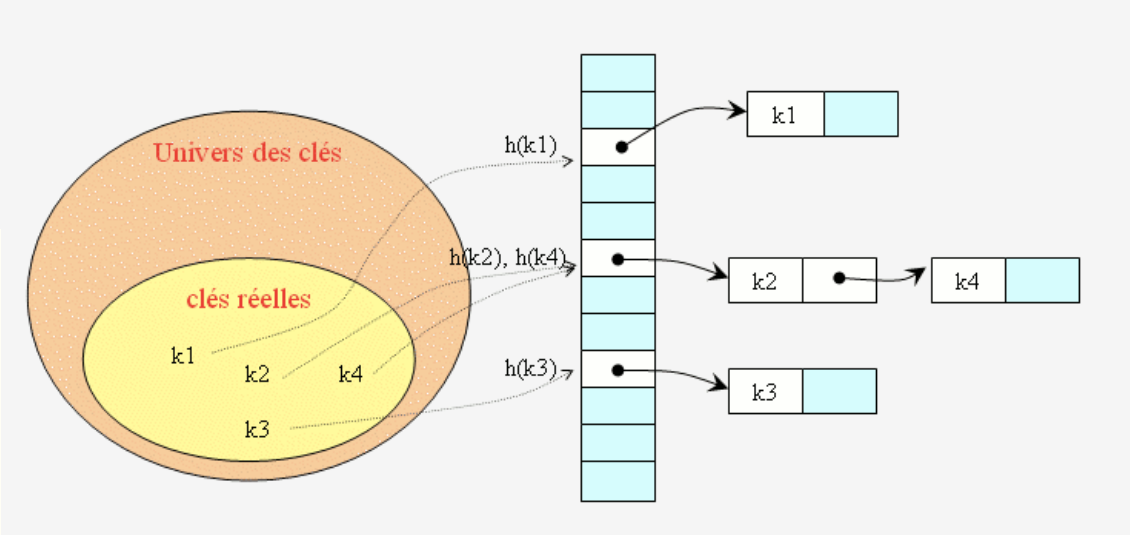


Recherche "dichotomique" dans une liste ordonnée.

Tables de hachage



- Accès aux éléments par une structure intermédiaire et non par indice



- Accès "relativement" rapide
- insertion et suppression "relativement" rapide

Tables de hachage (suite)

Objectifs :

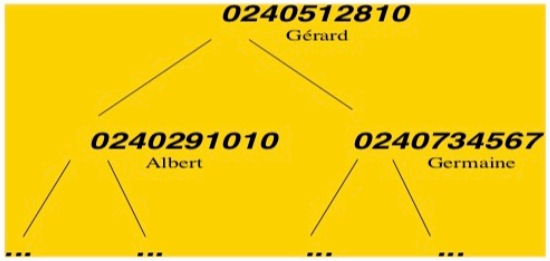
- Optimiser les temps d'accès

  exemple : recherche d'un nom d'abonné (parmi n) à partir d'un numéro de téléphone (parmi $R = 10^{10} - 1$)

  solutions :

  –par un arbre ordonné équilibré (cf. infra) : espace mémoire $\mathcal{O}(n)$ et temps d'accès $\mathcal{O}(log\ n)$

  –par un tableau : temps d'accès $\mathcal{O}(1)$ mais espace mémoire $\mathcal{O}(R)$



Tables de hachage (suite)

Principe :

- table avec m lignes accessibles par une « *clé* » (par exemple le modulo (base m))
- mais risque de conflits $\rightsquigarrow$ une liste chaînée pour chaque ligne
- $\rightsquigarrow$ espace mémoire $\mathcal{O}(n+m)$ et temps d'accès $\mathcal{O}(n/m)$ (donc réglable en fonction de m)

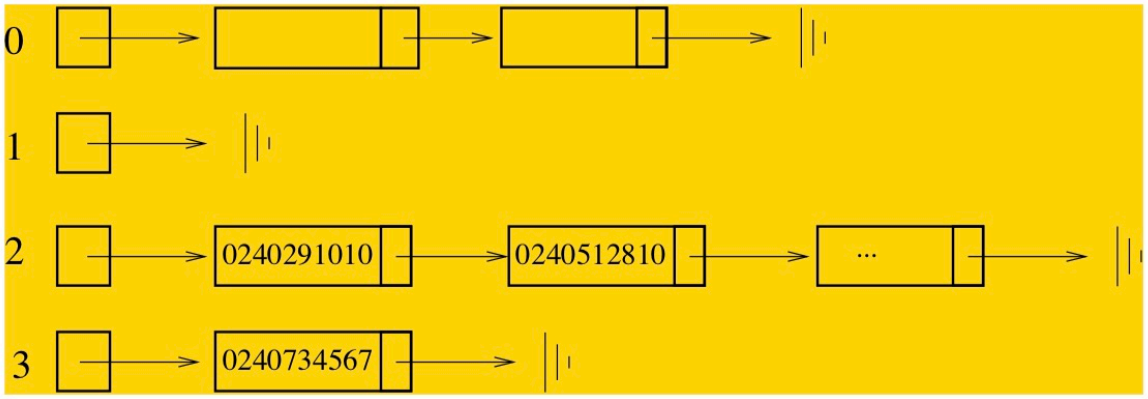


Figure 5 : Table de hachage (exemple précédent avec un modulo 4)

Tables de hachage (suite): en Java

- Map<K,V> (en français une association) entre une clé de type K et une valeur de type V

Java s'occupe de tout !
Exemple :

```
Map<Integer,String> m ;
```

Exemple

- Différences entre HashMap et TreeMap :

utilisation strictement identique
TreeMap peut s'utiliser lorsqu'on a des relations d'ordre sur la clé 🔑
=> insertions beaucoup plus longues, recherche beaucoup plus rapides

- Si m est une table de hachage

`m.put(clé,valeur)` : associe une valeur à une clé
`m.containsKey(clé)` : renvoie vrai si la clé est présente dans la table de hachage
`m.get(clé)` : renvoie la valeur associée à clé si elle est présente (toujours utiliser containsKey() juste avant)

Note :
La valeur associée à la clé est forcément unique… mais cette valeur unique peut être… une liste !
Exemple : `Map<String,LinkedList<Integer>> m1;`

Tables de hachage (suite): en Java

Exemple

```
import java.util.HashMap;

public class TestHash {

   public static void main(String[] args) {
      HashMap<String,String> annuaire = new HashMap<String,String>();

      // ajout des valeurs
      annuaire.put("Alfred","2399020806");
      annuaire.put("Daniel", "2186000000");

      // obtention d'un numéro
      if (annuaire.containsKey("Danielle")) {
            String num = annuaire.get("Danielle");
            System.out.println(« Danielle : "+num);
      }
      else {
         System.out.println("pas trouve");
      }

   }
}
```

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
V	**compute**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V	**computeIfAbsent**(**K** key, **Function**<? super **K**,? extends **V**> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
V	**computeIfPresent**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) Returns true if this map maps one or more keys to the specified value.
Set<**Map.Entry**<**K**,**V**>>	**entrySet**() Returns a **Set** view of the mappings contained in this map.
void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.