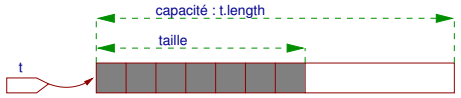
Algorithmique et Programmation

Jean-Christophe Engel

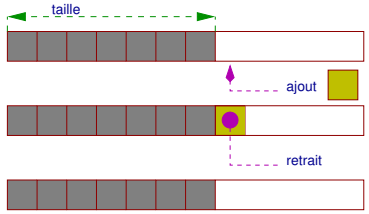
École supérieure d'ingénieurs de Rennes
Université de Rennes 1

Plan

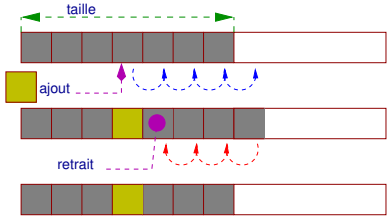
Structure intégrée : le tableau



- ajout/retrait en fin :



- ajout/retrait autre position :



Structure intégrée : le tableau

points positifs

- occupation mémoire minimale : O(N)
- accès direct à un élément en temps constant : O(1)
- ajout/retrait en fin en temps constant : O(1)

points négatifs

- ajout/retrait ailleurs coûteux : O(N)
- opérations « basiques » : consultation/modification
- pas de généricité
- capacité fixe

Conclusion

besoin de structure de données plus souple, munie d'opérations plus riches

Structure de données : la file

Caractéristiques

- opérations simples adaptées à la gestion d'une file d'attente

points positifs

- occupation mémoire minimale : O(N)
- ajout/retrait en temps constant : O(1)
- générique
- capacité non bornée

Intérêt

- programmation implémentation indépendante du problème
- fonctionnalités « riches »
- client : focalise sur problème du domaine d'application

Structures de données

Constat

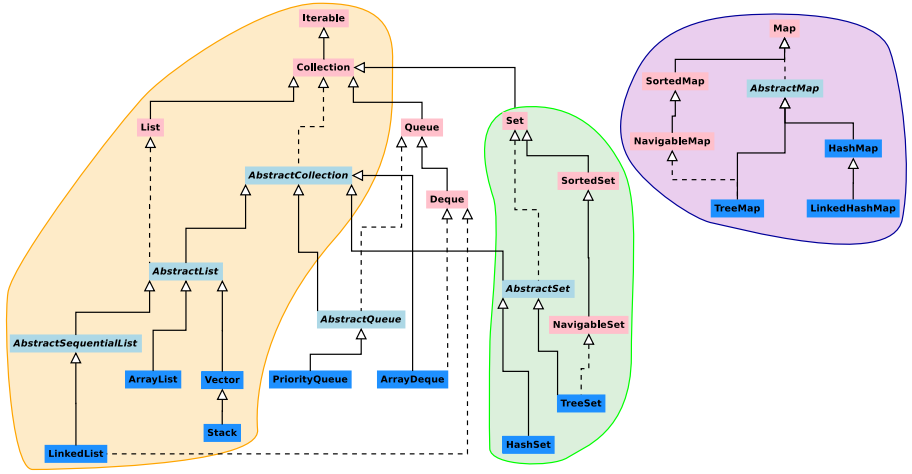
pas de structure de donnée universelle unique adaptée :

- à toutes les données
- à tous les problèmes

Bibliothèque de structures (java, C++, etc...)

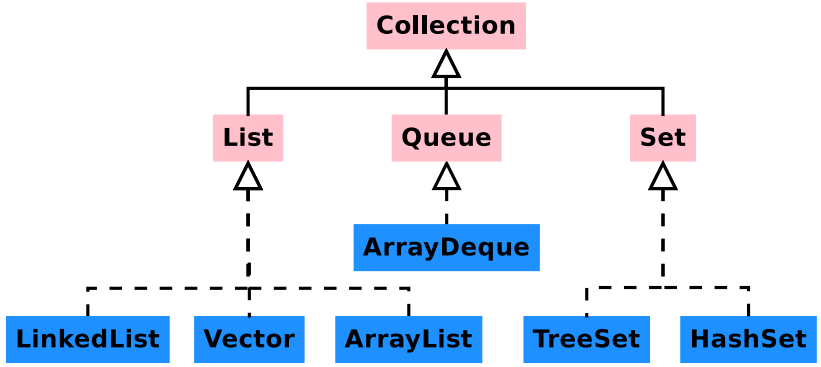
- nombreuses structures : génériques, capacité non bornée
- nombreuses opérations
- implémentations diverses
- avantages/inconvénients

Les structures de données de la bibliothèque Java



Les collections

- une interface pour chaque SDD
- plusieurs implémentations par interface



Les collections : opérations communes

1. Opérations individuelles
   - ajouter un élément : au début, en fin, position quelconque
   - retirer un élément : au début, en fin, position quelconque
   - rechercher une valeur
   - consulter la valeur d'un élément
2. Opérations de groupe
   - mêmes opérations sur plusieurs éléments
3. Opérations de parcours
   - initialiser un parcours
   - accéder à l'élément courant
   - passer à l'élément suivant
   - déterminer la fin de parcours

Parcours séquentiel de collection

Itérateur

- variable qui permet de programmer des itérations pour parcourir séquentiellement les éléments d'une collection.

- Opérations de parcours
  - initialiser un parcours
  - déterminer la fin de parcours
  - accéder à l'élément courant
  - passer à l'élément suivant

Tableau

- int icour = 0
- icour == t.length
- t[icour]
- icour += 1

Collection

- Iterator<E> itcour = collec.iterator()
- itcour.hasNext() : *vrai s'il reste un élément à parcourir*
- E valElem = itcour.next()

Parcours séquentiel de collection

- Initialiser un parcours (interfaces Collection<E>, List<E>, ...)

```
// returns an iterator over the elements in this collection.
Iterator<E> iterator();
```

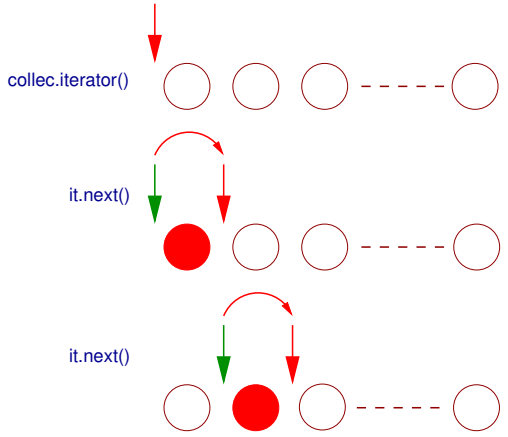
- Interface Iterator<E>

```
// spécification du type Iterator<E>
// E est le type des éléments de la collection à parcourir
public interface Iterator<E> {

  // Returns true if the iteration has more elements.
  boolean  hasNext();

  // Returns the next element in the iteration.
  E  next();

  // Removes from the underlying collection the last
  // element returned by this iterator.
  void  remove();
}
```


Parcours séquentiel de collection : principe

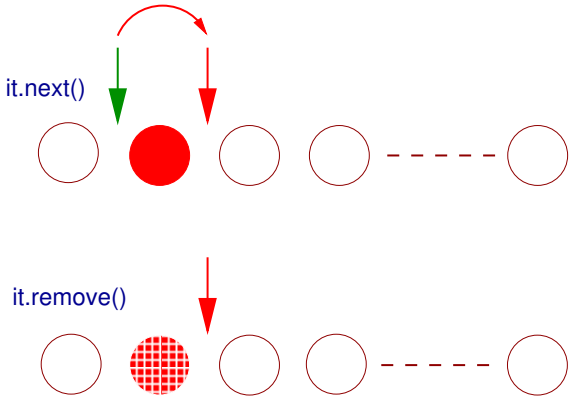
itérateur

- curseur qui désigne une position **entre** deux éléments.



- iterator() renvoie un itérateur positionné **avant** le premier élément
- hasNext() est vrai s'il reste un élément **après** le curseur
- next() renvoie l'élément **après** le curseur puis **avance le curseur**

Suppression lors du parcours



- Supprime l'élément renvoyé par le dernier next()
- Il faut obligatoirement (au moins) un next() avant chaque remove()
- Pas deux appels consécutifs de remove()
- Après remove, l'itérateur ne bouge pas

Parcours séquentiel de collection

```
public class  ChasseAuTresor {
static void deplacerRobots(Collection<SpecRobot> armee,
    int xCible, int yCible, int xBombe, int yBombe, int seuil) {
  // initialiser un parcours de la collection
  Iterator<SpecRobot> monIterateur = armee.iterator();
  // Appliquer un traitement à chaque robot
  while (monIterateur.hasNext()) {
    // parcours non terminé : traiter le robot courant
    SpecRobot courant = monIterateur.next();
    if (carreDistance(courant.getX(), courant.getY(),
          xBombe, yBombe) <= seuil) {
      monIterateur.remove(); // robot trop proche de la bombe
    } else if (courant.getDirection() != Direction.Ouest) {
      courant.tourner();
    } else if (carreDistance(courant.getX(), courant.getY(),
        xCible, yCible)) > 25) {
      courant.avancer();
    }
  } // while
} // deplacerRobots
```


Spécification de liste : interface List<E>

Opérations élémentaires

```
// Returns the number of elements in this list.
int            size();

// Returns true if this list contains no elements.
boolean        isEmpty();

// Returns true if this list contains the specified element.
boolean        contains(E element);

// Appends the specified element to the end of this list.
boolean        add(E element);          // always returns true

// Removes the first occurrence of the specified element
// from this list, if it is present.
boolean        remove(E element);     // returns true if present
```

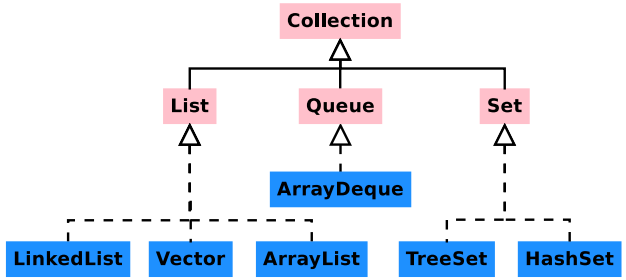

Spécification de liste : interface List<E>

Opérations « de masse »

```
// Returns true if this list contains all of the elements
// of the specified collection.
boolean        containsAll(Collection<E> c);

// Appends all of the elements in the specified collection
// to the end of this list, in the order that they are returned
// by the specified collection's iterator.
boolean        addAll(Collection<E> c);

// Removes from this list all of its elements that are
// contained in the specified collection.
boolean        removeAll(Collection<E> c);

// Retains only the elements in this list that are contained
// in the specified collection.
boolean        retainAll(Collection<E> c);

// Removes all of the elements from this list.
void           clear();
```

Implémentations de l'interface List<E>

- LinkedList<E> :
  - implémentation par chaînage ;
  - ajout/suppression performant ;
  - parcours séquentiel obligatoire ⇒ pas d'accès direct (indice)
- Vector<E>, ArrayList<E> :
  - tableau de capacité non bornée ;
  - parcours séquentiel possible, accès direct performant ;
  - ajout/suppression en début/fin performant, ailleurs : médiocre



Spécification de liste : interface List<E>

Opérations avec accès indexé

Attention : à réserver aux implémentations Vector et ArrayList
⟹ à éviter avec l'implémentation LinkedList

```
// Returns the element at the specified position in this list.
E               get(int index);

// Replaces the element at the specified position in this list
// with the specified element.
E               set(int index, E element);

// Returns the index of the first occurrence of the specified
// element in this list,
// or -1 if this list does not contain the element.
int             indexOf(E element);

// Returns the index of the last occurrence of the specified
// element in this list, or -1 if this list does not contain the elem
int             lastIndexOf(E element);
```


Spécification de liste : interface List<E>

Opérations avec accès indexé

Attention : à réserver aux implémentations Vector et ArrayList
⟹ à éviter avec l'implémentation LinkedList

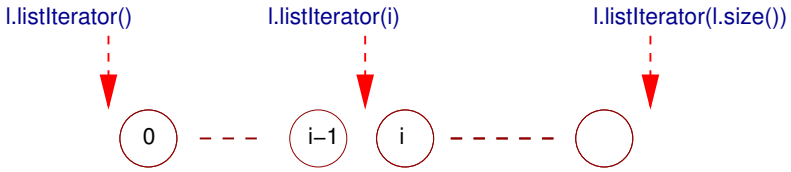
```
// Inserts the specified element at the specified position
//  in this list.
void            add(int index, E element);

// Inserts all of the elements in the specified collection
// into this list at the specified position.
boolean         addAll(int index, Collection<E> c);

// Removes the element at the specified position in this list.
E               remove(int index);
```


Quelques exemples

Itérateur de liste

- Dans l'interface List<E> on trouve deux nouvelles méthodes de création d'itérateur :

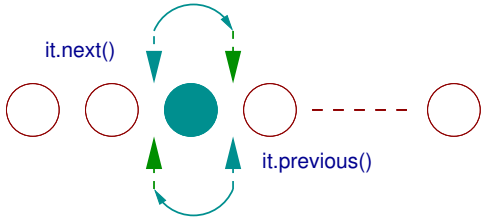
```
// Returns a list iterator over the elements in this list.
ListIterator<E>       listIterator();

// Returns a list iterator over the elements in this list,
// starting at the specified position in the list.
ListIterator<E>       listIterator(int index);
```



Itérateur de liste

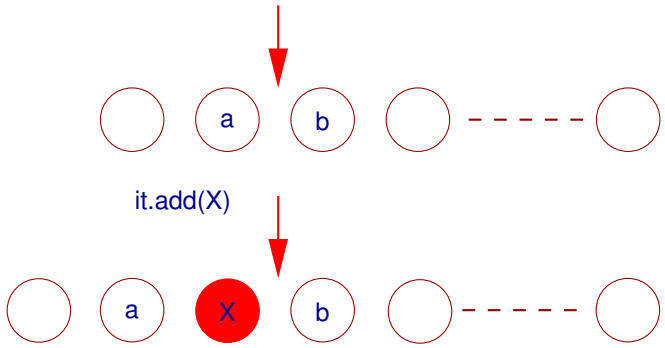
- dans l'interface ListIterator<E>, on dispose de 2 opérations de parcours supplémentaires

```
// Returns true if this list iterator has more elements
// when traversing the list in the reverse direction.
boolean  hasPrevious();

// Returns the previous element in the list
// and moves the cursor position backwards.
E  previous();
```



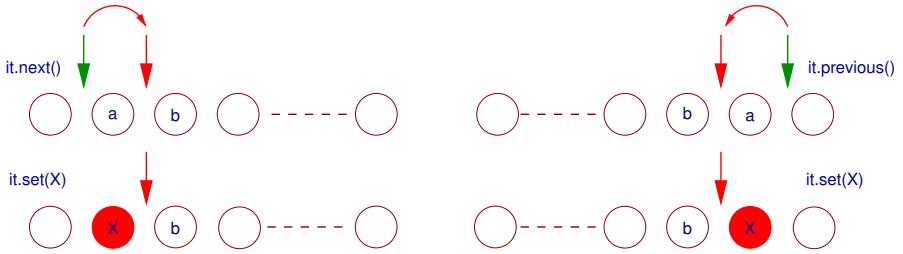
Itérateur de liste
opérations pendant le parcours

```
// Inserts the specified element into the list.
void  add(E e);
```



Itérateur de liste
opérations pendant le parcours

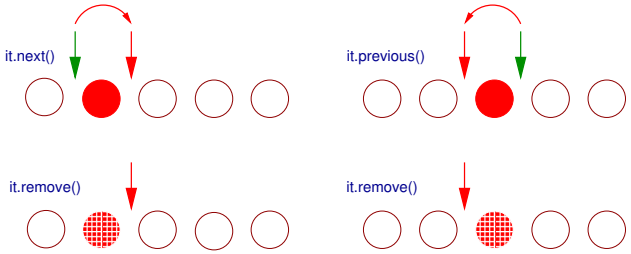
```
// Replaces the last element returned by next() or previous()
// with the specified element.
void  set(E e);
```



Itérateur de liste

opérations pendant le parcours

```
// Removes the last element returned by next() or previous().
void remove();
```

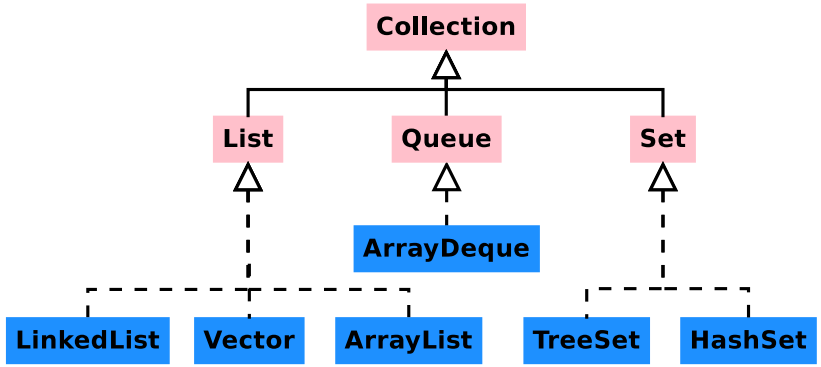


- Supprime l'élément renvoyé par le dernier next ou le dernier previous
- Au moins un next ou un previous avant chaque remove $\Longrightarrow$pas d'ajout juste avant remove
- Après remove, l'itérateur ne bouge pas

Listes : Algorithmes génériques

- trier
- mélanger aléatoirement
- inverser l'ordre des éléments d'une collection
- remplir une collection avec une valeur
- copier une collection dans une autre
- ajouter les éléments d'une collection à une autre

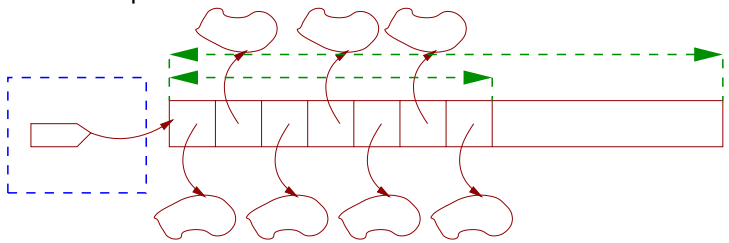
Implémentations de l'interface List<E>



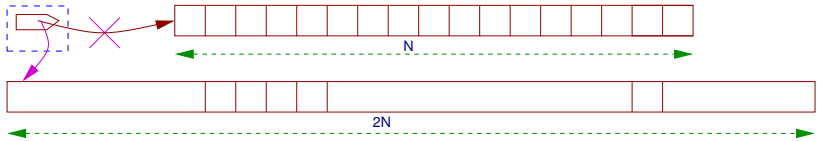
Implémentations de l'interface List<E>

Vector, ArrayList

- taille < capacité

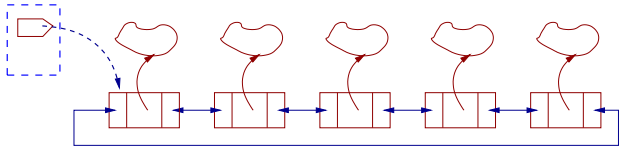


- taille = capacité $\Longrightarrow$ augmentation de la capacité

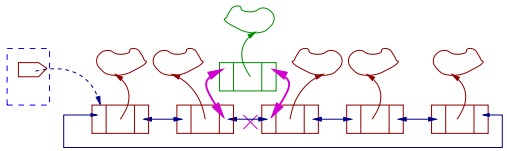


Implémentations de l'interface List<E>
LinkedList

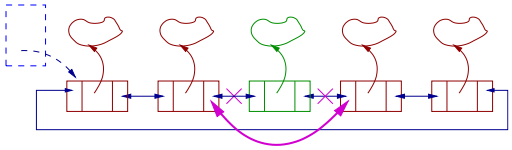
- chaînage



- ajout



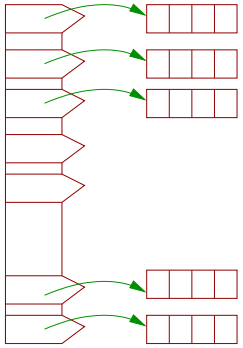
- retrait



Interface Set<E>

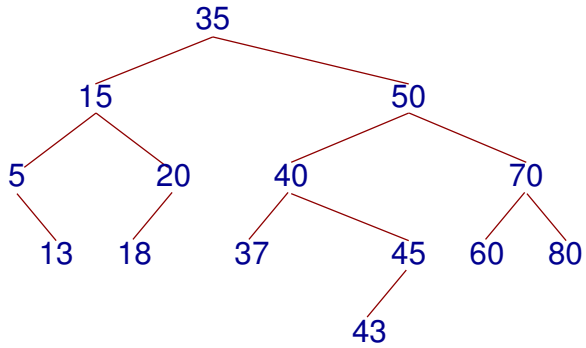
- Set<E> modélise un ensemble au sens mathématique
- Opérations unitaires similaires à celles de l'interface List<E>
  - add : renvoie vrai si l'ajout a eu lieu, faux sinon
  - remove : renvoie vrai si la suppression a eu lieu, faux sinon
- Opérations « de masse » similaires à celles de l'interface List<E>
- Accès aux éléments :
  - pas d'opération avec indice
  - parcours obligatoirement avec itérateur
  - pas d'itérateur bidirectionnel
- 2 implémentations principales
  - HashSet
  - TreeSet : les éléments doivent être comparables

Implémentations de l'interface Set<E>
HashSet



- calcul de l'indice à partir de la valeur de l'élément (*fonction de hachage* : int hashCode())
- très performant si taille tableau principal bien choisie et si la fonction de hachage répartit uniformément les indices
- aucun contrôle sur l'ordre des éléments dans l'ensemble

Implémentations de l'interface Set<E>
TreeSet



- représentation : arbre binaire de recherche équilibré (AVL ou rouge/noir)
- toutes opérations on $O(log_2(N))$
- rangement (donc parcours) des éléments en ordre croissant
- impose une relation d'ordre sur les éléments

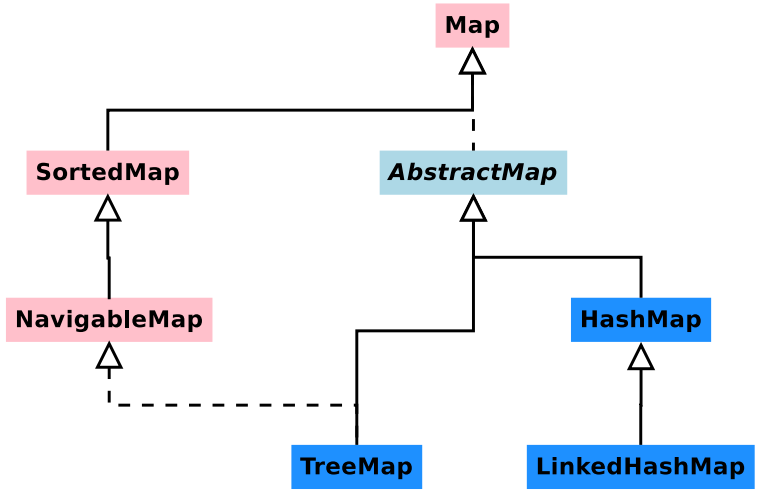
Interface Queue<E>

- Opérations spécifiques

```
// Inserts the specified element into this queue if it is
// possible to do so immediately without violating capacity
// restrictions.
// Returns true if the element was added to this queue, else false
boolean  offer(E e);

// Retrieves, but does not remove, the head of this queue,
// or returns null if this queue is empty.
E  peek();

// Retrieves and removes the head of this queue,
// or returns null if this queue is empty.
E  poll();
```

- Implémentations :
  - file d'attente générale : ArrayDeque<E>, LinkedList<E>
  - file de priorité : PriorityQueue<E>, éléments comparables, implémentation avec un *tas* (*heap*)

Interface Map<K, V>



Interface Map<K, V>

- Opérations spécifiques

V put(K key, V value) :
: Ajoute la paire <key, value> si clé absente et renvoie null ; remplace et renvoie l'ancienne valeur si clé présente

V get(K key) :
: Renvoie la valeur associée à la clé si présente ; renvoie null si clé absente

V remove(K key) :
: Supprime la paire <key, value> si clé présente et renvoie la valeur ; renvoie null si clé absente

boolean containsKey(K key) :
: Renvoie vrai si la clé est présente

boolean containsValue(V value) :
: Renvoie vrai si la valeur est présente

Interface Map<K, V>

- Opérations « classiques »

```
// Returns the number of key-value mappings in this map.
int  size();
// Returns true if this map contains no key-value mappings.
boolean  isEmpty();
// Removes all of the mappings from this map (optional operation).
void  clear();
```

- parcours avec itérateur : par l'intermédiaire de collections associées

Set<K> keySet() :
: Renvoie l'ensemble des clés

Collection<V> values() :
: Renvoie la collection des valeurs

On peut ensuite effectuer un parcours avec itérateur de l'ensemble des clés ou de la collection des valeurs

Interface Map<K, V>

- parcours avec itérateur : par l'intermédiaire de collections associées

  Set<Map.Entry<K,V>> entrySet() :

  Renvoie l'ensemble des paires <clé, valeur>

- interfaceMap.Entry<K, V>

```
// Returns the key corresponding to this entry.
K   getKey();

// Returns the value corresponding to this entry.
V   getValue();

// Replaces the value corresponding to this entry
// with the specified value.
V   setValue(V value);
```


Implémentations de l'interface Map<K,V>

HashMap, TreeMap

HashMap<K,V> :

même principe que HashSet ; les clés doivent disposer d'une fonction de hachage : int hashCode()

TreeMap<K,V> :

même principe que TreeSet ; les clés doivent être comparables