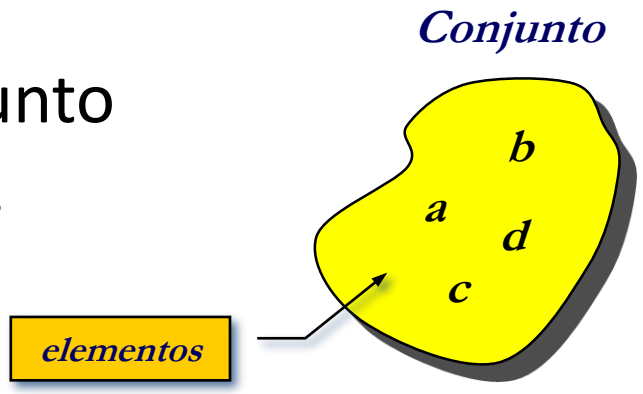
Otras estructuras no lineales: Conjuntos y Diccionarios.

Tema IV

Conjuntos

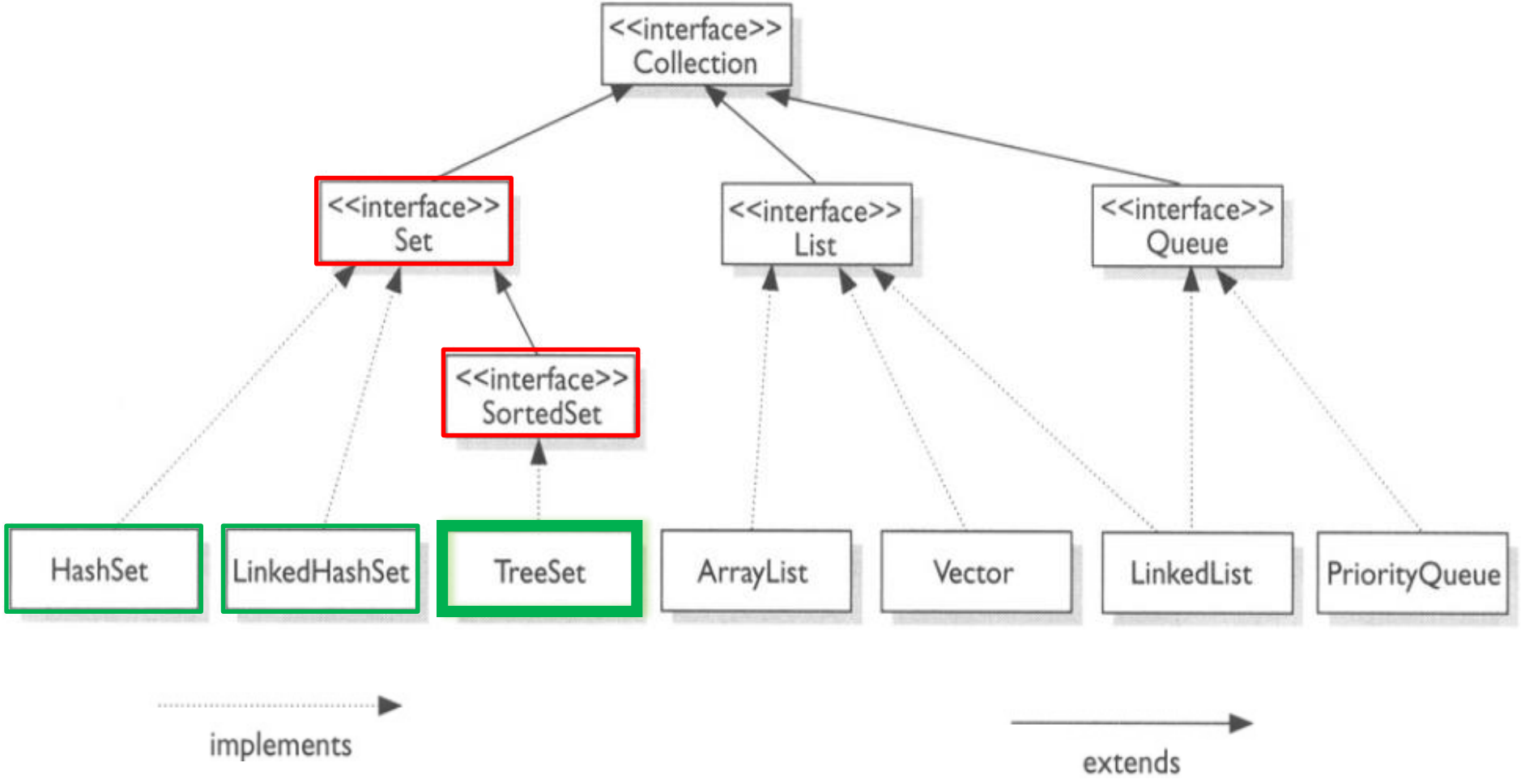
- Un **conjunto** es una colección **no ordenada** de elementos **sin repetir.**
- Operaciones de conjuntos:
  - Adición y borrado de elementos, test de inclusión de un elemento.
  - Intersección, unión, subconjunto y diferencia entre conjuntos.



Variantes de Conjuntos

- Conjuntos ordenados
  - Las eficiencia de las operaciones mejora si los elementos se mantienen en orden
- Multiconjuntos (bag)
  - A veces interesa tener elementos repetidos

Conjuntos en JAVA



Interfaz *set<E>* (I)

```
public interface Set<E> extends Collection<E> {      // Iterable<E>
    int size();
    boolean isEmpty();

    /* Añade al conjunto el elemento especificado si no está presente.
       Retorna falso si no se puede insertar el elemento*/
    boolean add(E e);                  // opcional
    boolean remove(Object o);          // opcional
    boolean contains(Object o);        ∈

    Iterator<E> iterator();
(…)
```

Interfaz *set<E>* (II)

(…)

```
/* Retorna cierto si el objeto "o" es un conjunto y tiene los mismos
   elementos que el conjunto receptor  */
boolean equals(Object o);

/*Retorna el valor del código hash para el conjunto:
  la suma de los códigos hash de los elementos del conjunto,
  siendo 0 el código hash de null.  */
int hashCode();
```

(…)

Interfaz *set<E>* (III)

```
(…)
    // Operaciones entre conjuntos
    boolean containsAll(Collection<?> c);                    → subconjunto
    boolean addAll(Collection<? extends E> c); //opcional    → unión
    boolean removeAll(Collection<?> c);        //opcional    → diferencia
    boolean retainAll(Collection<?> c);        //opcional    → intersección
    void clear();                              //opcional
}
```

Ej: S1={1,3,9,5} S2={7,1,4,3,2}

S1.addAll(S2)	**S1=S1∪S2**	**S1={1,3,9,5,7,4,2}**
S1.retainAll(S2)	**S1=S1∩S2**	**S1={1,3}**
S1.removeAll(S2)	**S1=S1−S2**	**S1={9,5}**
S1.containsAll(S2)	**S2⊂S1**	**S1=false**

AbstractCollection<E> -AbstractSet<E>

~~http://www.docjar.org/~~ (para ver las implementaciones)

https://developer.classpath.org/doc/java/util/

Subconjunto

```
public boolean containsAll(Collection<?> c)
{
    for (Object e : c)
        if (!contains(e))
            return false;
    return true;
}
```

Unión

```
public boolean addAll(Collection<?> c){
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

AbstractCollection<E> -AbstractSet<E>

https://developer.classpath.org/doc/java/util/

```
public boolean containsAll(Collection<?> c){
    Iterator<?> itr = c.iterator();
    int pos = c.size();
    while (--pos >= 0)
        if (!contains(itr.next()))
                return false;
    return true;
}
```

```
public boolean addAll(Collection<? extends E> c){
    Iterator<? extends E> itr = c.iterator();
    boolean modified = false;
    int pos = c.size();
    while (--pos >= 0)
        modified |= add(itr.next());
    return modified;
}
```

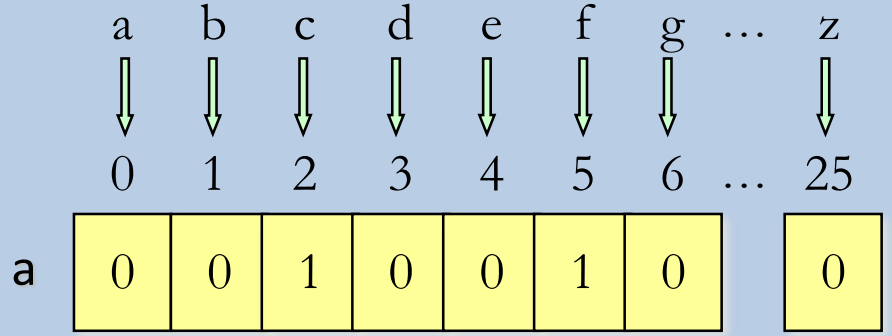
Representaciones de Conjuntos

- Conjuntos basados en vectores de bits
- Conjuntos basados en listas
- Conjuntos basados en tablas hash
- Conjuntos basados en árboles (**ordenados**)
- …

Conjuntos con vectores de bits

- No se guardan los elementos en el conjunto sino una "indicación" de si cada elemento pertenece al conjunto a no (booleano).
- Se debe poder convertir elemento a posición de vector y viceversa.
- Permite hacer operaciones a nivel de bit (and, or, xor,…).

Ej: Representación de un conjunto C de caracteres en el rango 'a'..'z'



'a' ∉ C ya que a[0]=0 (false)

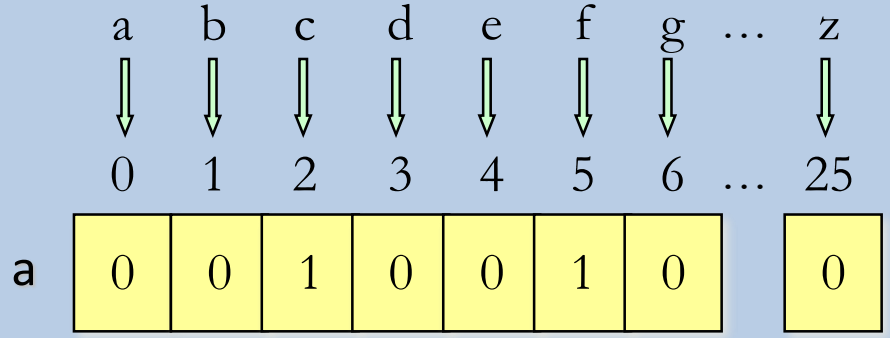
'f' ∈ C ya que a[2]=1 (true)

BitSet

Conjuntos con vectores de bits (II)

- La inserción, borrado y test de pertenencia son de tiempo constante.
- Las operaciones entre conjuntos son de O(n) *

Ej: Representación de un conjunto S de caracteres en el rango 'a'..'z'



'a' ∉ S ya que a[0]=0 (false)

'c' ∈ S ya que a[2]=1 (true)

Conjuntos con vectores de bits (III)

- Ventajas:
  - No se guardan los elementos propiamente dichos, así que se puede ahorrar mucho espacio
  - Coste temporal de las operaciones contante en la mayoría de los casos
- Inconvenientes:
  - El tamaño siempre es el tamaño del número total de elementos que podría tener el conjuntos, aunque el conjunto esté vació.
  - Por tanto, sólo tiene sentido si este tamaño máximo es conocido y no es muy elevado.
  - Debe poderse establecer una equivalencia los elementos del conjunto y la posición en el vector de bits y viceversa.

Conjuntos con listas

- Los elementos del conjunto se guardan en algún tipo de listas **por composición (no es adecuada la herencia)**
  - ArrayList<E>
  - LinkedList<E>
- Inserción, borrado, test de pertenencia: **O(n)**
  - Alguna de estas operaciones podría mejorarse si hay orden
- Unión, intersección, subconjunto y diferencia entre conjuntos: **$O(n^2)$**
  - Con **listas ordenadas** se puede mejorar el coste de las operaciones entre conjuntos → **algoritmo de mezcla**
  - El coste pasaría a ser **O(n)**

Conjuntos con listas *ArrayList*

- Dado que se puede acceder a los elementos de cualquier posición en O(1), se puede sobrescribir el método **contains** usando **búsqueda binaria.**
- El coste pasa de **O(n)** a **O(log n)** http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

Versión iterativa

```
función búsqueda_binaria (Colección c, Elemento e): retorna booleano
 /* c debe ser ordenada e indexada (índices de 0 a tamaño_de_c-1)  */
   ini = 0
   fin = tamaño_de_c-1
   mientras ini <= fin hacer
      medio = ((fin - ini) / 2) + ini      // División entera
      si  elemento_de_la_posición_medio== e  entonces
            retornar Verdadero    //o retornar medio
      si no
         si    e < elemento_de_la_posición_medio entonces
               fin = medio - 1
         si no
               ini = medio + 1
   fin (mientras)
   retornar Falso    //o retornar -1
fin (función)
```

Versión recursiva

```
función búsqueda_binaria (Colección c, Entero ini, Entero, fin
                          Elemento e): retorna booleano
 /* c debe ser ordenada e indexada (índices de ini a fin)  */
   resultado = Falso  // o también -1
   si ini <= fin entonces
      medio = ((fin- ini) / 2) + ini      // División entera
      si  elemento_de_la_posición_medio== e  entonces
            resultado = Verdadero    //o medio
      si no
         si    e < elemento_de_la_posición_medio entonces
               resultado = búsqueda_binaria (c, ini, medio – 1, e)
         si no
               resultado = búsqueda_binaria (c, medio+1, fin, e)
   fin (si)
   retornar resultado
fin (función)
```

Mezcla de colecciones ordenadas

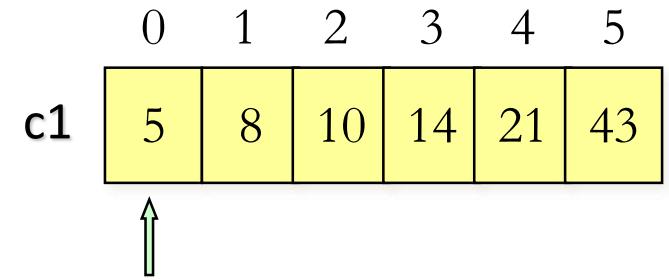
- La idea consiste en recorrer **simultánemante** ambas colecciones
- Con este algoritmo se puede hacer la unión, la intersección,…

```
acción mezcla (Colección c1, Colección c2)   //o función
 /* c 1 y c2 debe estar ordenadas */
   situarse sobre el primer elemento de c1 y c2
   mientras haya_elementos_en_ambas_colecciones hacer
      elem_c1=obtener elemento de c1
      elem_c2=obtener elemento de c2
      si  elem_c1 == elem_c2  entonces
            tratar elementos
            avanzar en ambas colecciones
      si no
         si    elem_c1< elem_c2  entonces
               tratar elem_c1
               avanzar en c1
         si no
               tratar elem_c2
               avanzar en_c2
   fin (mientras)
 …
```

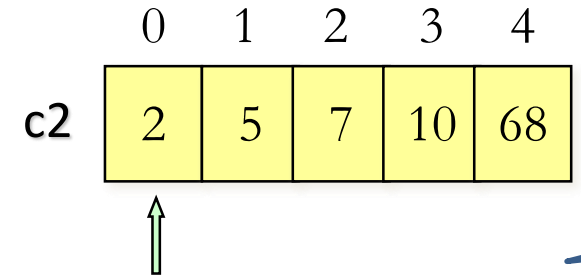
O no se hace ninguno o sólo uno de ellos

```
…
   //Tratamientos por si se acaba una colección antes
   mientras haya_elementos_en_c1 hacer
      elem_c1=obtener elemento de c1
      tratar elem_c1
      avanzar en c1
   fin (mientras)
   mientras haya_elementos_en_c2 hacer
      elem_c2=obtener elemento de c2
      tratar elem_c2
      avanzar en c2
   fin (mientras)
fin (acción)
```

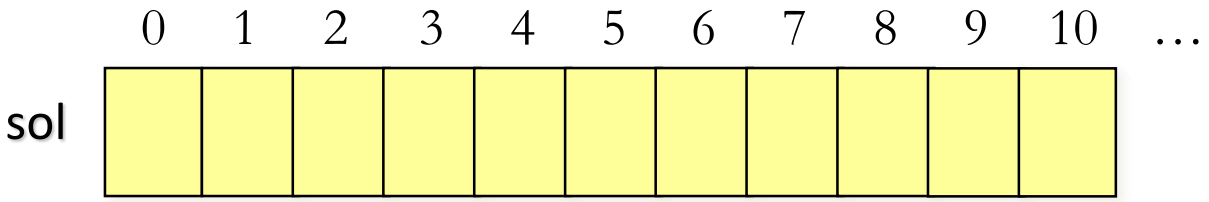
Mezcla de colecciones ordenadas



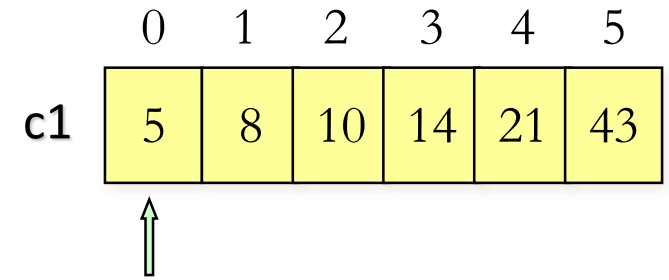
Ejemplo para la operación de *unión*



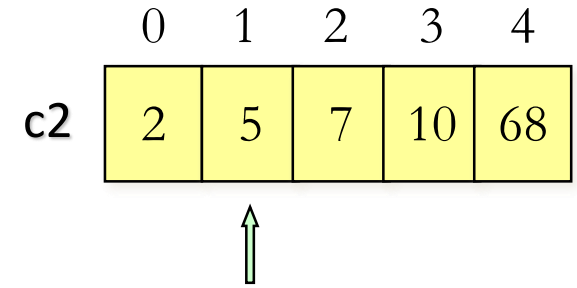
Se avanza en la colección del menor



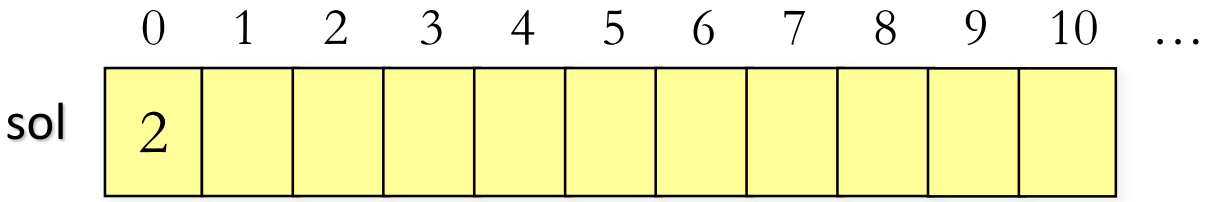
Mezcla de colecciones ordenadas



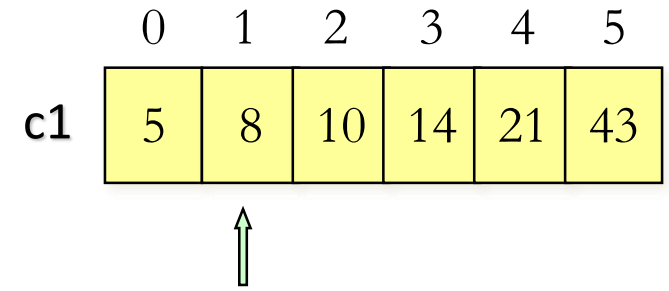
Ejemplo para la operación de *unión*



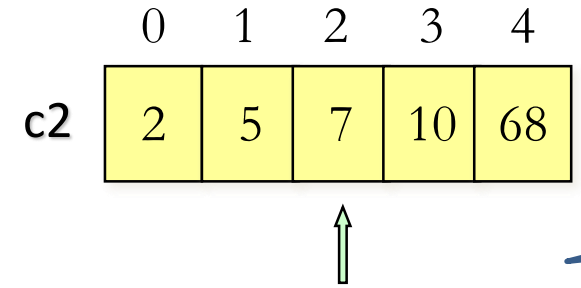
Se avanza en ambas colecciones

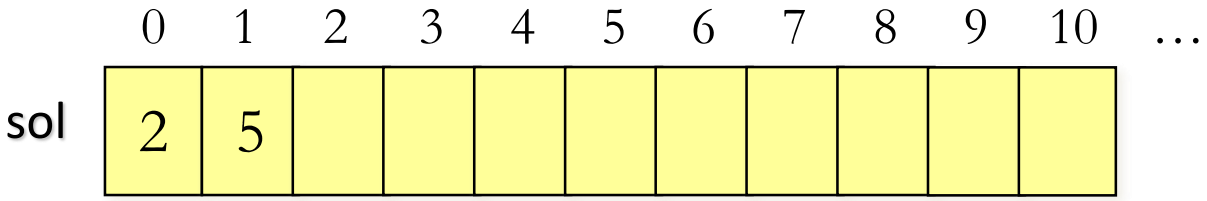


Mezcla de colecciones ordenadas

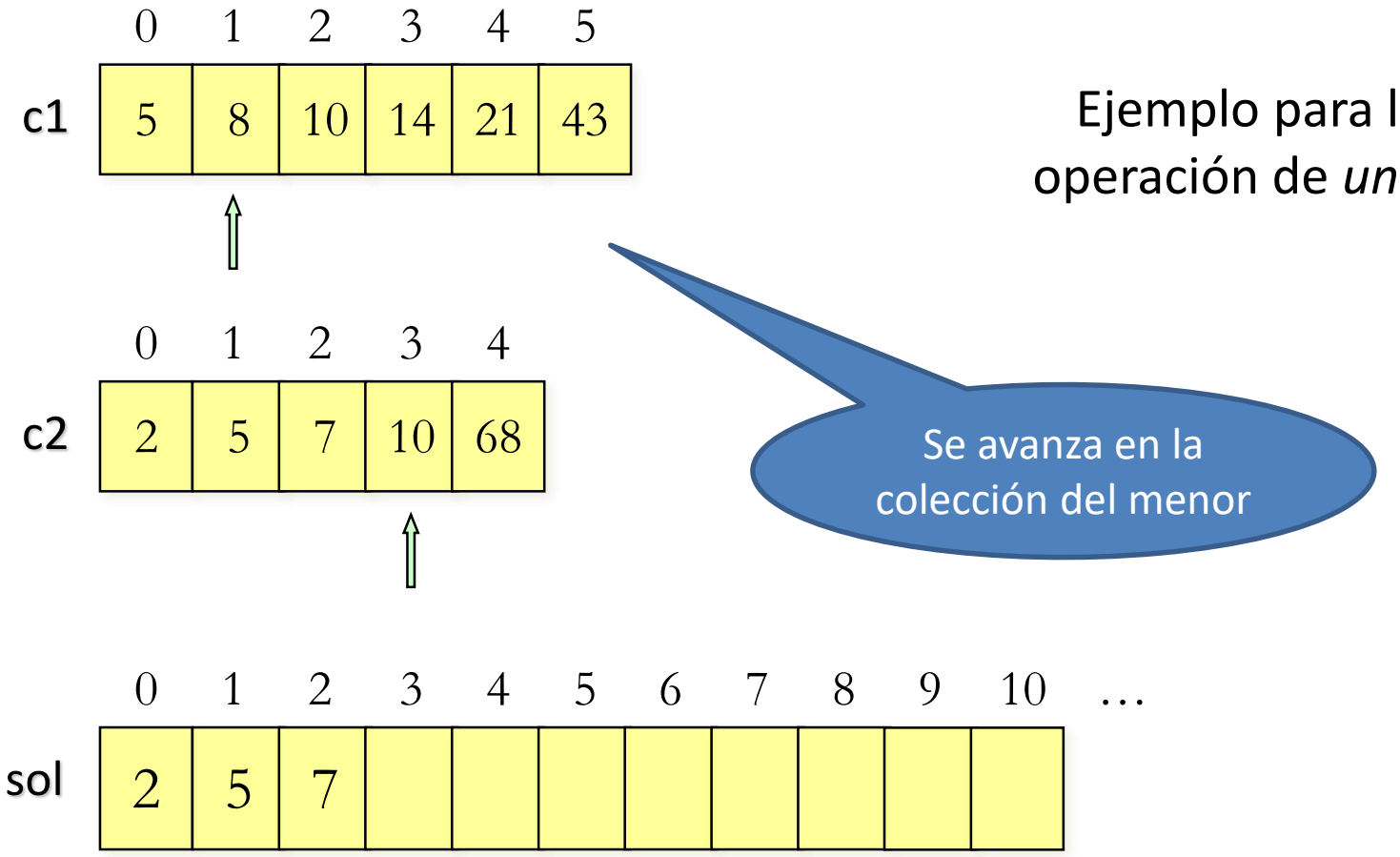


Ejemplo para la operación de *unión*



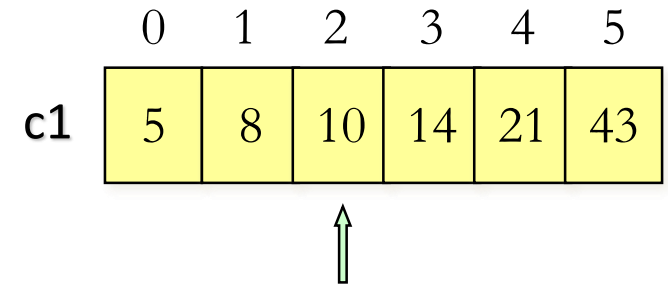


Mezcla de colecciones ordenadas

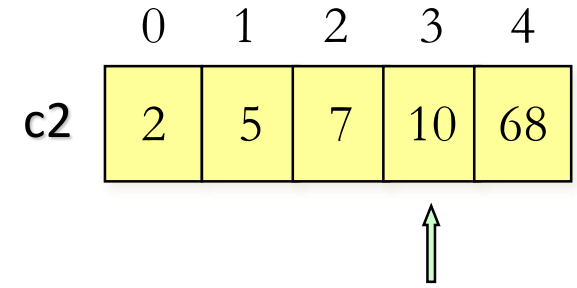


Ejemplo para la operación de *unión*

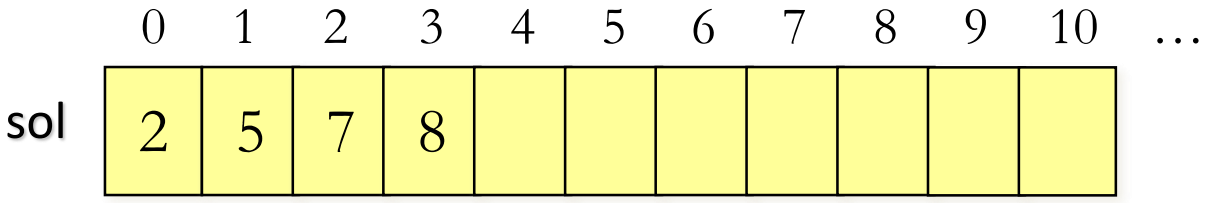
Mezcla de colecciones ordenadas



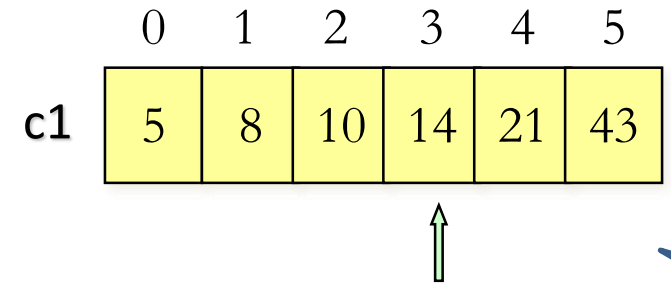
Ejemplo para la operación de *unión*

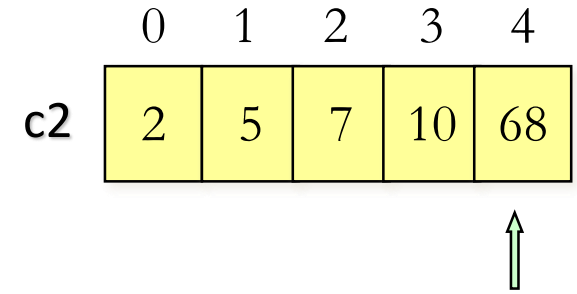


Se avanza en ambas colecciones



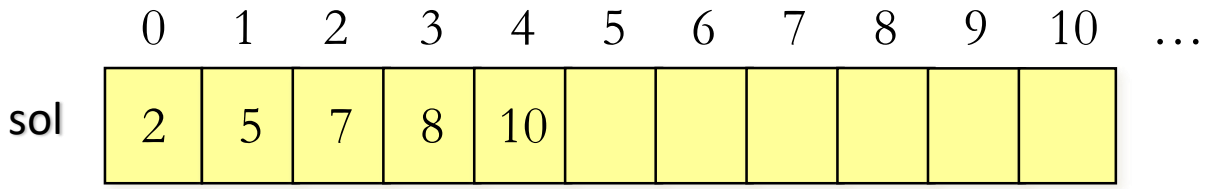
Mezcla de colecciones ordenadas



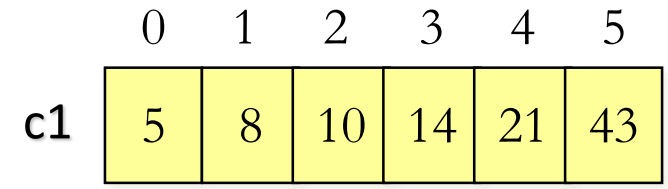


Ejemplo para la operación de *unión*

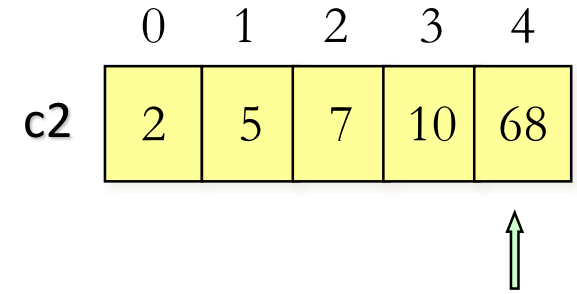
Se avanza en la colección del menor (3 veces en el ejemplo)



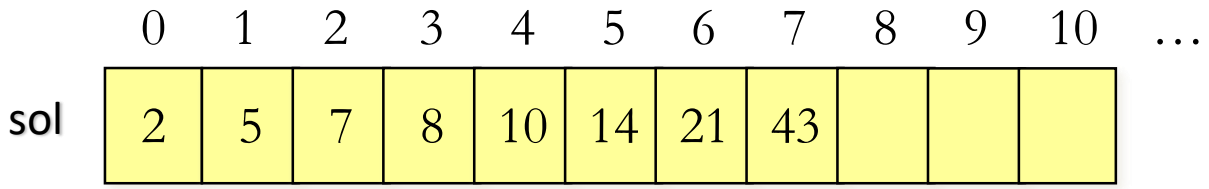
Mezcla de colecciones ordenadas



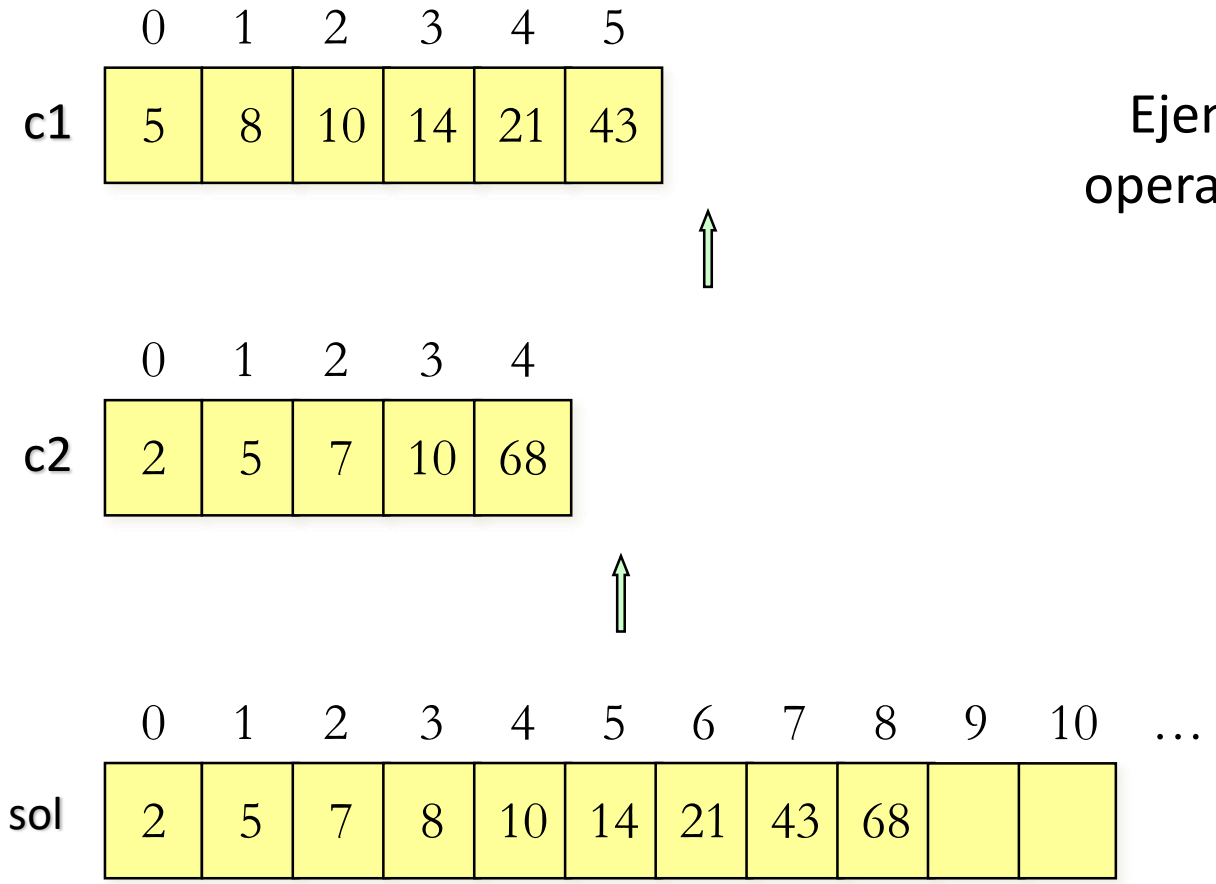
Ejemplo para la operación de *unión*



Cuando se acaba una colección, se tratan todos los elementos que quedan de la otra



Mezcla de colecciones ordenadas



Ejemplo para la operación de *unión*

Unión de listas con y sin repetidos

```
// Unión de dos listas SIN ORDENAR y CON elementos repetidos  -> O(n)
public static <E> LinkedList<E> mergeRepes (LinkedList<E> l1,LinkedList<E> l2 ) {
    LinkedList<E>   result=new LinkedList< > (l1);
    ListIterator<E> itr2 = l2.listIterator();
    while (itr2.hasNext())
            result.addLast(itr2.next());              // O(1)
    return result;
}

// Unión de dos listas SIN ORDENAR y SIN elementos repetidos   -> O(n²)
public static <E> LinkedList<E> mergeSinrepes(LinkedList<E> l1,LinkedList<E> l2 ) {
    LinkedList<E>   result=new LinkedList< >(l1);
    ListIterator<E> itr2 = l2.listIterator();
    while (itr2.hasNext()){
            E  val=itr2.next();
            if (!result.contains(val))                // O(n)
                result.addLast(val);   }              // O(1)
   return result;
}
```

Unión eficiente de listas ordenadas O(n)

```
public static <E> LinkedList<E> mergeOK(LinkedList<E> l1,LinkedList<E> l2 ) {
    ListIterator<E>  itr1 = l1.listIterator();
    ListIterator<E>  itr2 = l2.listIterator();
    LinkedList<E>    result=new LinkedList< >();
    while (itr1.hasNext()  && itr2.hasNext())   {   //Bucle principal: ambas listas tienen elementos
       E  val1=itr1.next();     itr1.previous();
       E  val2=itr2.next();     itr2.previous();
       if (val1<val2){
                result.addLast(val1);
                itr1.next();  }                  // se avanza en la lista 1
       else if (val2<val1){
                result.addLast(val2);
                itr2.next();  }                  // se avanza en la lista 2
       else {                                    //val1==val2
                result.addLast(val1);            // no meto repetidos (pero se pueden meter si interesa)
                itr1.next();                     // se avanza en ambas listas
                itr2.next();  }
   }  //fin del bucle principal
```

Sólo si el iterador de las colecciones tiene “previous”

OJO: (<,>,==) Las comparaciones no pueden ser así!!!!

Unión eficiente de listas ordenadas (cont)

```
    …
  }  //fin del bucle principal

 // Miro a ver si una lista es mayor que la otra
  while ( itr1.hasNext() )                  // La lista 1 tiene mas elementos
      result.addLast(itr1.next());

  while ( itr2.hasNext() )                  // La lista 2 tiene mas elementos
     result.addLast(itr2.next());

  return result;
}
```

Ej: lista1=(0, 2, 4, 8) lista2={3, 6, 8, 12, 15}

merge(lista1,lista2) → **(0, 2, 3, 4, 6, 8, 12, 15)**

Unión eficiente de colecciones ordenadas$_1$ O(add)

```
public static <E> Collection<E> mergeOK(Collection<E> c1, Collection<E> c2 ) {
      PeekIterator<E>  itr1 = new PeekingIterator(c1);
      PeekIterator<E>  itr2 = new PeekingIterator(c2);
      Collection<E> result=new LinkedList<E>();  // O cualquier colección
      while (itr1.hasNext()  && itr2.hasNext())   {   //Bucle principal: ambas listas tienen elementos
       E  val1=itr1.peek() ;   E  val2=itr2.peek();
       if (val1<val2){
                result.add(val1);
                itr1.next();  }                  // se avanza en la colección 1
       else if (val2<val1){
                result.add(val2);
                itr2.next();  }                  // se avanza en la colección 2
       else {                                    //val1==val2
                result.add(val1);                // no meto repetidos (pero se pueden meter si interesa)
                itr1.next();                     // se avanza en ambas colecciones
                itr2.next();  }
   }  //fin del bucle principal
```

Si el iterador **NO** tiene "previous" puedo usar PeekIterator

OJO: (<,>,==) Las comparaciones no pueden ser así!!!!

Unión eficiente de colecciones ordenadas$_1$ (cont)

```
    …
  } //fin del bucle principal

  // Miro a ver si una lista es mayor que la otra
  while ( itr1.hasNext() )       // La colección 1 tiene mas elementos
    result.add(itr1.next());

  while ( itr2.hasNext() )       // La colección 2 tiene mas elementos
    result.add(itr2.next());

 return result;
}
```

Forma de hacer las comparaciones en JAVA (orden natural)

addAll se dede redefinir usando "merge" así

```
int comp=((Comparable<? super E>) val1).compareTo(val2);
if (comp<0)                 //val1<val2)
                …
else if (comp>0)     //val1>val2)
                …
else                        //comp==0 ; val1==val2
                …
```

Unión eficiente de colecciones ordenadas$_2$ O(add)

```
public static <E> Collection<E> mergeOK(Collection<E> c1,Collection<E> c2 ) {
    Iterator<E>  itr1 = c1.iterator();
    Iterator<E>  itr2 = c2.iterator();
    Collection<E> result=new LinkedList<E>(); //cualquier colección
    E val1=(itr1.hasNext())? itr1.next():null;
    E val2=(itr2.hasNext())? itr2.next():null;
    //Bucle principal: ambas colecciones tienen elementos
    while (val1!=null  && val2!=null) {              //Bucle principal: ambas listas tienen elementos
      if (val1<val2){
          result.add(val1);
          val1=(itr1.hasNext())? itr1.next():null;} // se avanza en c1
      else if (val2<val1){
          result.add(val2);
          val2=(itr2.hasNext())? itr2.next():null;} // se avanza en c2
      else {                                          //val1==val2
          result.add(val1);              // no meto repetidos (pero se pueden meter si interesa)
          val1=(itr1.hasNext())? itr1.next():null;     // se avanza en ambas colecciones
          val2=(itr2.hasNext())? itr2.next():null; }
    } //fin del bucle principal  (faltan los otros dos bucles)
```

Si el iterador **NO** tiene "previous" puedo usar **null**

OJO: (<,>,==) Las comparaciones no pueden ser así!!!!

PeekIterator

```
public interface PeekIterator<E> extends Iterator<E> {
	/**
	 * Permite consultar el siguiente elemento a iterar.
	 * @return el valor del siguiente elemento de la iteración  (el que retornaría la operación next()
	 * @throws NoSuchElementException si la iteración no tiene más elementos.
	 */
	E peek();
}

public class PeekingIterator<E> implements PeekIterator<E> {
	private Iterator<E> itr;		// el iterador de un objeto iterable
	private E current;			// el elemento que retornaría la operación next()
	private boolean setCurrent;	// cierto si current contiene el siguiente elemento

	public PeekingIterator(final Iterable<E> obj) {
		itr = obj.iterator();
		setCurrent = itr.hasNext();
		if (setCurrent)
			current = itr.next();
	}
```

PeekIterator (cont)

```
public E peek() {
        if (!hasNext())
            throw new NoSuchElementException();
         return current;
}


@Override
public boolean hasNext() {
        return setCurrent;
}
```

```
@Override
public E next() {
        if (!hasNext())
                throw new NoSuchElementException();
        E temp = current;
        setCurrent = false;
        if (itr.hasNext()) {
                current = itr.next();
                setCurrent = true;
        }
        return temp;
}
```

Conjuntos con tablas hash

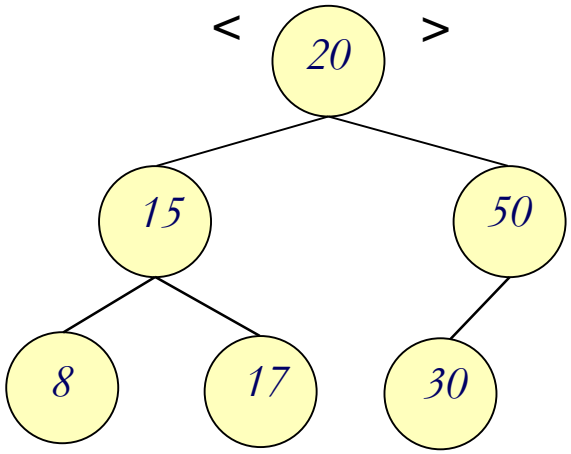
- Los elementos del conjunto se guardan en una **tabla hash**
- Este tipo de dato se estudiará en el tema siguiente
- El coste de las operaciones del conjunto depende del coste de las operaciones de la tabla hash
  - Inserción, borrado, test de pertenencia: **O(1) (en media)**
  - Unión, intersección, subconjunto y diferencia entre conjuntos: **O(n) (en media)**
- Java proporciona las clases
  - HashSet<E> y LinkedHashSet<E> que implementan la interfaz Set<E>

Conjuntos con árboles (abb+)

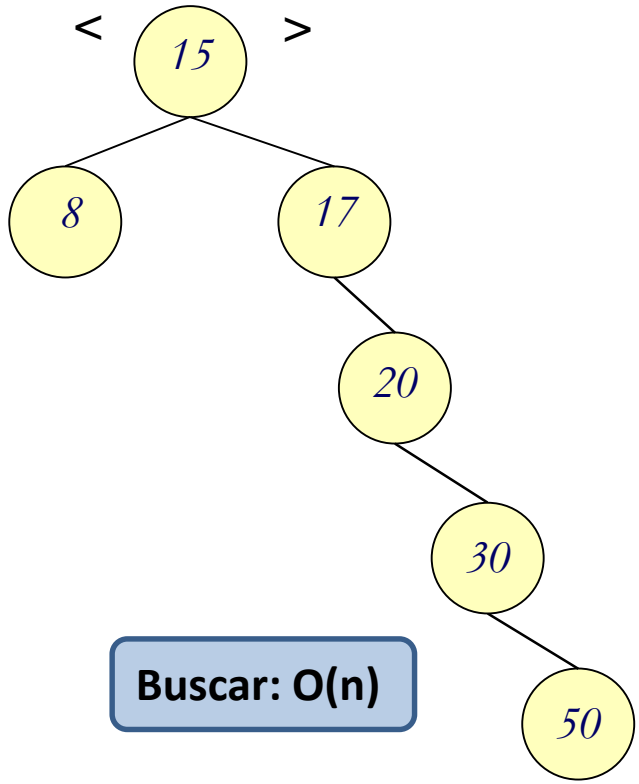
- Los elementos del conjunto se guardan en algún tipo de árbol de búsqueda binario **equilibrado** (**por composición**)
- El coste de las operaciones del conjunto depende del coste de las operaciones del árbol → **coste logarítmico**
  - Inserción, borrado, test de pertenencia: **O(log n)**
  - Unión, intersección, subconjunto y diferencia entre conjuntos: **O(n log n)**
- Java proporciona la clase
  - TreeSet<E> que implementa la interfaz sortedSet<E>

Árboles binarios de búsqueda **equilibrados**

- Son un tipo de árboles binarios de búsqueda con algún mecanismo adicional para **"equilibrar"** los elementos
  - AVL, Rojo/Negro, AA, etc



Buscar: O(log n)



Buscar: O(n)

Interfaz *sortedSet<E>*

```
public interface SortedSet<E> extends Set<E>
{
   // Range-view
   SortedSet<E> subSet(E fromElement, E toElement);  //[fromEle, toEle)
   SortedSet<E> headSet(E toElement);                //[first(), toEle)
   SortedSet<E> tailSet(E fromElement);              //[fromEle, last()]
   // Endpoints
   E first();
   E last();
    // Comparator access
   Comparator<? super E> comparator();
}
```

Ejemplo de uso de *TreeSet*

```
/**
 *Output:
 F E D C B A
 */
import java.util.Comparator;
import java.util.TreeSet;

class MyComparator implements Comparator<String>
{
  public int compare(String a, String b)
 {
     //orden inverso para cadenas de caracteres
     return b.compareTo(a);
 }
```

not (negativo si a<b
positivo si a>b
0 si a==b)

```
public class MainClass {
  public static void main(String args[]) {
   TreeSet<String> ts = new TreeSet<String>(new
       MyComparator());

   ts.add("C");
   ts.add("A");
   ts.add("B");
   ts.add("E");
   ts.add("F");
   ts.add("D");

   for (String element : ts)
         System.out.print(element + " ");
   System.out.println();
  }
}
```

Multiconjuntos - Bag

- Son conjuntos en los cuales puede haber elementos repetidos
- Se pueden hacer versiones ordenadas o sin ordenar
- Es fácil hacer multiconjuntos basados tipos que permiten la repetición de elementos, como las listas.
  - Sólo hay que implementar el método **add** de manera adecuada.
- Sin embargo, hay tipos de datos que no permiten repetir elementos, como las tablas hash o los árboles de búsqueda.
  - En este caso puede usarse un truco basado en guardar **pares** de la forma: **(elemento, número_de_repeticiones)**

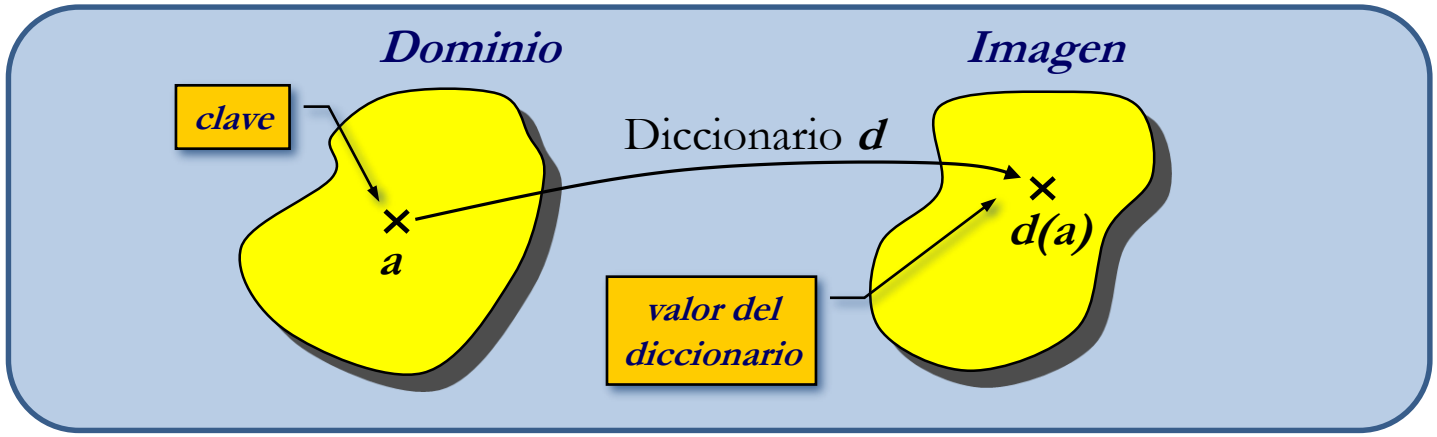
Por ejemplo: {5, 9, 3, 5, 8, 5, 3, 5} → [(5,4), (9,1), (3,2), (8,1)]

Diccionarios

- Un **diccionario** (tabla, aplicación, mapping, map, array asociativo) es una **colección de pares** clave/valor (**asociaciones**).
- Permite representar una **aplicación entre dos conjuntos** cuando no es posible describir ésta mediante un algoritmo (en caso contrario, sería suficiente con definir la función correspondiente).

Diccionarios (II)

- Los valores del conjunto origen o dominio de la aplicación se denominan **claves**.
- Las claves son **únicas** (no se pueden repetir)
- Las claves tienen **valores** asociados en el conjunto imagen o codominio.



Diccionarios (III)

Diccionario inglés-español

(book, libro)
(dog, perro)
(house, casa)
…

Diccionario DNI-notas

(10222333, (5;7;B))
(15000111, (6;3,R))
…

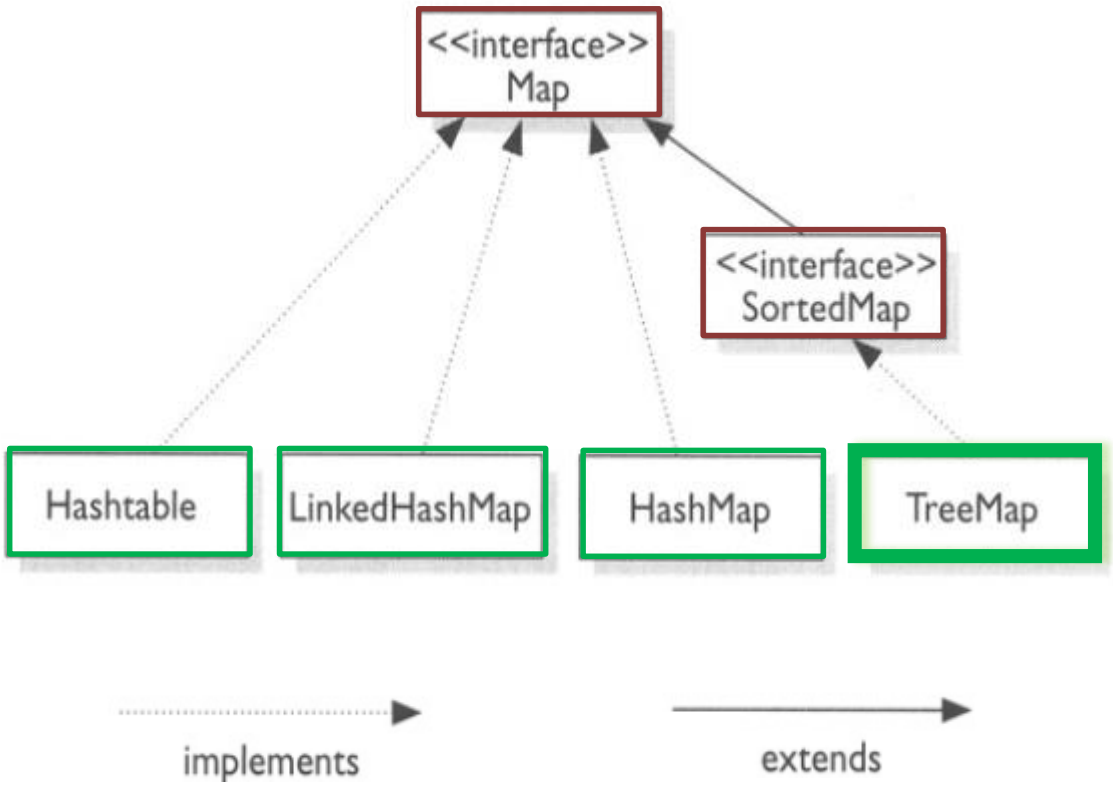
Diccionario número-cuadrado

(2, 4)
(10, 100)
(4, 16)
…

Variantes de Diccionarios

- Diccionarios ordenados
  - El orden se estable atendiendo a las **claves**
- Multidiccionarios/multimap
  - Son diccionarios en los cuales las claves **no son únicas** (se pueden repetir claves)
  - A veces no se repiten realmente las claves, sino que se tiene una sola clave pero con varios valores asociados

Diccionarios en JAVA



Interfaz Map<K,V> (I)

```
public interface Map<K,V> {
   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();
   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();
```

¿?

Interfaz *Map<K,V>* (II)

```
...
   // Collection Views
   public Set<K> keySet();
   public Collection<V> values();
   public Set<Map.Entry<K,V>> entrySet();

   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
}
```

Tipo de dato interno para guardar las asociaciones (pares clave/valor) **Map.Entry<K,V>**

Interfaz *Map<K,V>* (III)

Resumen de métodos	
V	put(K key, V value) Asocia el valor y clave especificados en este diccionario (opcional).
V	get(Object key) Retorna el valor asociado a la clave especificada o null si este diccionario no tiene un valor asociado a la misma.
V	remove(Object key) Elimina la **asociación** para la clave especificada si existe en este diccionario (opcional).
boolean	containsKey(Object key) Retorna true si este diccionario contiene una asociación para la clave especificada.
boolean	containsValue(Object value) Retorna true si este diccionario contiene una o más claves con el valor especificado.
boolean	isEmpty() Retorna true si este diccionario no contiene ninguna asociación
int	size() Retorna el número de asociaciones clave/valor en este diccionario.
void	clear() Elimina todas las asociaciones de este diccionario (opcional).

Inserta y modifica

Interfaz *Map<K,V>* (IV)

Resumen de métodos	
void	putAll(Map<? extends K,? extends V> m) Copia todas las asociaciones del diccionario especificado en esta asociación (opcional).
boolean	equals(Object o) Compara si el objeto especificado y este diccionario son iguales.
int	*hashCode()* Retorna el valor del código *hash* para este diccionario como la suma de los códigos *hash* de cada asociación que contiene.
Set<Map.Entry<K,V>>	entrySet() Retorna una vista Set de las asociaciones contenidas en este diccionario
Set<K>	keySet() Retorna la vista Set de las clave contenidas en este diccionario
Collection<V>	values() Retorna una vista Collection con todos los valores contenidos en este diccionario

Interfaz *Map.Entry<K,V>*

Tipo de dato interno para guardar las asociaciones (pares clave/valor)

Resumen de métodos	
K	getKey() Retorna la clave correspondiente a esta entrada.
V	getValue() Retorna el valor correspondiente a esta entrada.
V	setValue(V value) Reemplaza el valor correspondiente a esta entrada por el valor especificado (opcional).
int	*hashCode()* Retorna el código *hash* para esta entrada.
boolean	equals(Object o) Compara el objeto especificado con esta entrada (par clave/valor) para igualdad. Dos asociaciones son iguales cuando tanto la clave como el valor coinciden.

Ejemplo de uso de *Diccionarios*

```
Map<Integer,String> mapA = new HashMap<Integer,String>();        // …=TreeMap<Integer,String>();
mapA.put(1, "valor_1");
mapA.put(3, "valor_3");
mapA.put(2, "valor_2");
String value = (String) mapA.get(2);                             //value="valor_2"
Iterator<Integer> iterator1 = mapA.keySet().iterator();          //iterador sobre las claves
Iterator<String> iterator2 = mapA.values().iterator();           //iterador sobre los valores

while(iterator1.hasNext()) {
   Object key   = iterator1.next();       //Integer
   Object value = mapA.get(key);          //String
   System.out.print("("+key + ", "+value +") ");
 }
System.out.println();
```

Salida: (1, valor_1) (2, valor_2) (3, valor_3)

¡OJO!
HashMap no es ordenado, pero la vista de conjunto si

Ejemplo de uso de *Diccionarios* (II)

```
mapA.put(3, "valor_3_modificado");
mapA.put(0, "valor_0");
mapA.remove(2);
 for(Object key : mapA.keySet()) {
    Object value = mapA.get(key);
    System.out.print("("+key + ", "+value +") ");
 }
 System.out.println();

 Set<Map.Entry<Integer,String>> pairset=mapA.entrySet();
 for(Map.Entry<Integer,String> par : pairset) {
    Integer key=par.getKey();
    String value = par.getValue();
    System.out.print("("+key + ", "+value +") ");
}
```

Salida: ((0, valor_0) (1, valor_1) (3, valor_3_modificado))

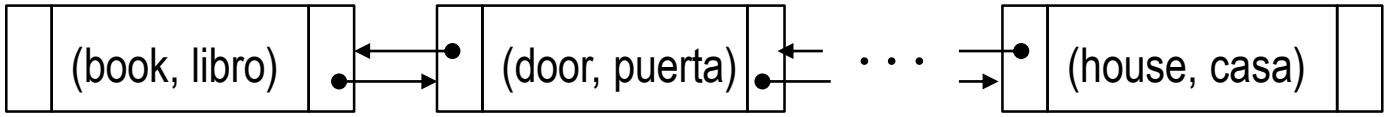
Representaciones de Diccionarios

- Diccionarios basados en listas
- Diccionarios basados en tablas hash
- Diccionarios basados en árboles binarios de búsqueda (**ordenados**)
- …

Diccionarios basados en listas

- Se utilizaría algún tipo de lista (por composición) cuyos elementos serían asociaciones clave/valor
  - Se precisa una clase para los pares o asociaciones

  Ej: Con listas doblemente enlazadas



- El coste de las operaciones básicas de inserción, borrado y búsqueda sería: **O(n)**

Diccionarios con tablas hash

- Los elementos del diccionario se guardan en una tabla hash de asociaciones clave/valor.
- (Las tablas hash se estudiarán en el tema siguiente)
- El coste de las operaciones del diccionario depende del coste de las operaciones de la tabla hash
  - Inserción, borrado, búsqueda: **O(1) (en media)**
- Java proporciona las clases
  - HashMap<K,V> y LinkedHashMap<K,V> que implementan la interfaz Map<K,V>

Diccionarios con árboles (abb+)

- Los elementos del conjunto se guardan en algún tipo de árbol de búsqueda binario **equilibrado** (**por composición**)
- El orden se establece en base a las claves
- El coste de las operaciones del conjunto depende del coste de las operaciones del árbol → **coste logarítmico**
  - Inserción, borrado y búsqueda: **O(log n)**
- Java proporciona la clase
  - TreeMap<K,V> que implementa la interfaz sortedMap<K,V>

Interfaz *SortedMap<K,V>*

Resumen de métodos (adiciones respecto a Map)	
Comparator<? super K>	comparator() Retorna el comparador utilizado para ordenar las claves de este diccionario, o null si se utiliza el *orden natural* de sus claves.
K	firstKey() Retorna la primera (menor) clave de este diccionario.
K	lastKey() Retorna la última (mayor) clave de este diccionario.
SortedMap<K,V>	subMap(**K** fromKey, **K** toKey) Retorna una vista de la parte del diccionario cuyas claves están en el rango [fromKey, toKey).
SortedMap<K,V>	headMap(**K** toKey) Retorna una vista de la parte del diccionario cuyas claves son estrictamente menores que la clave especificada. [first(), toKey)
SortedMap<K,V>	tailMap(**K** fromKey) Retorna una vista de la parte del diccionario cuyas claves son mayores o iguales que la clave especificada. [fromKey, last()]

Ejemplo de uso de *TreeMap<K,V>*

```
SortedMap<Integer,String> mapB = new TreeMap<Integer,String>();
SortedMap<Integer,String> mapC;
mapB.put(1, "valor_1");
mapB.put(3, "valor_3");
mapB.put(2, "valor_2");
mapB.put(4, "valor_4");
```

mapB: ((1, valor_1) (2, valor_2) (3, valor_3) (4, valor_4))

```
mapC=mapB.subMap(2,4);
```

mapC: ((2, valor_2) (3, valor_3))

```
mapC=mapB.tailMap(2);
```

mapC: ((2, valor_2) (3, valor_3) (4, valor_4))

```
mapC=mapB.headMap(3);
```

mapC: ((1, valor_1) (2, valor_2))

Multimaps en JAVA

- Java no proporciona una clase específica para multidiccionarios
- Se puede "simular" usando un diccionario donde una clave lleve asociada una **colección de valores**

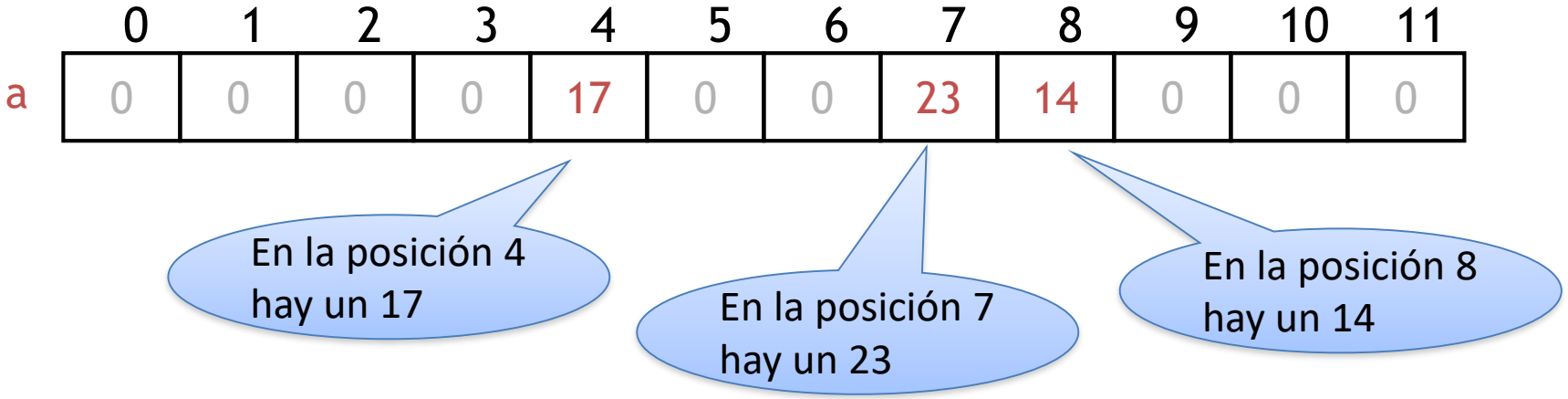
Multidiccionario "real"
(book, libro)
(book, cuaderno)
(book, guía)
(house, casa)
…

Multidiccionario "simulado"
(book, {libro, cuaderno, guía})
(house, {casa})
…

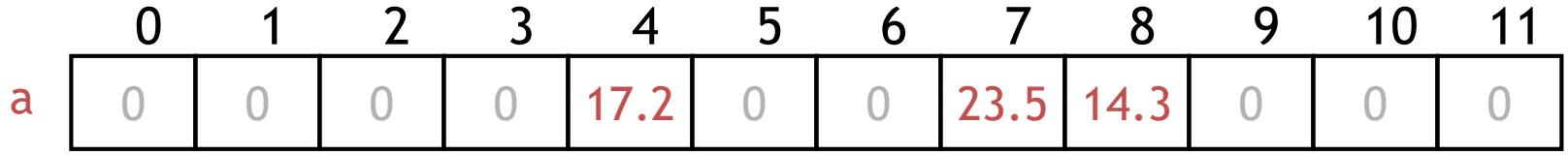
```
Map<String, List<String>> mm1 = new HashMap<String, List<String>>();
Map<String, List<String>> mm2 = new TreeMap<String, Set<String>>();
```

Aplicaciones de Diccionarios

- Vectores esparcidos (sparse array, sparse vector) son los que tienen muchos elementos con valores **nulos**
  - IDEA: **Guardar sólo los valores no nulos para ahorrar espacio**



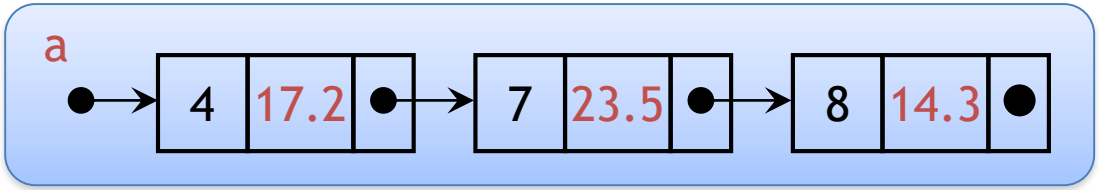
Aplicaciones de Diccionarios (II)



- ✓ Formato CRS (compressed row storage)

```
int [] indices = { 4, 7, 8}
double[] valores = { 17.2, 23.5, 14.3 }
```

- ✓ Con listas enlazadas



- ✓ Con diccionarios

a ((4, 17.2), (7, 23.5), (8, 14.3))

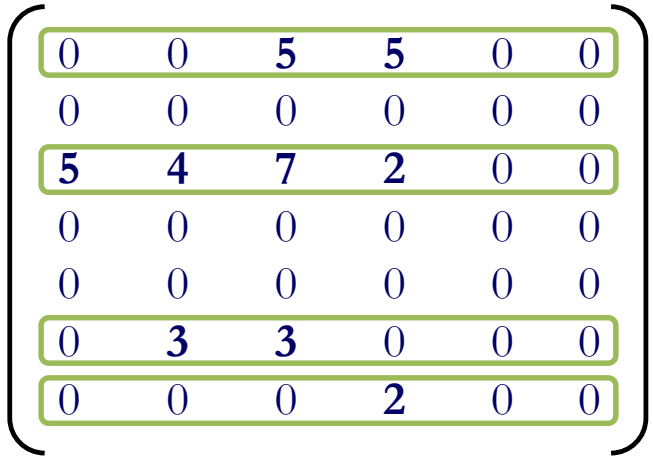
Map<Integer, E> data = new TreeMap<Integer,E>()

Las claves son las posiciones

Aplicaciones de Diccionarios (III)

- Matrices esparcidas

$$\begin{pmatrix} 0 & 0 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 4 & 7 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$




```
Map<Integer, spaseVector<E>> data = new TreeMap<Integer,sparseVector<E>>()
```

```
Map< … > data = new TreeMap<Integer,TreeMap<Integer, E>>()
```

- Valor de “data” para el ejemplos anterior:

0	0	**5**	**5**	0	0
0	0	0	0	0	0
5	**4**	**7**	**2**	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	**3**	**3**	0	0	0
0	0	0	**2**	0	0

```
data=( (0, ( (2,5) (3,5) ) )
       (2, ( (0,5) (1,4) (2,7) (3,2) ) )
       (5, ( (1,3) (2,3)  ) )
       (6, ( (3,2) ) ) )
```

Las claves son los números de fila (en orden)

Las claves son los números de columna (en orden)

Aplicaciones de Diccionarios (IV)

- Referencias cruzadas
  - Un generador de referencias cruzadas es un programa que selecciona palabras de un fichero de entrada y las guarda en orden con los números de línea en donde aparecen.
  - Los números de línea también deben estar en orden.

```
public class MyListSetMain {
 public static void main() {
    MyListSet<Integer> s=new MyListSet<Integer>();
    s.add(25);
    s.add(30);
    s.add(2);
}
```

Integer	3
MyListSet	3
MyListSetMain	1
add	4,5,6
class	1
main	2
new	3
public	1,2
s	3,4,5,6
static	2
void	2

Aplicaciones de Diccionarios (V)

- Referencias cruzadas

Integer	3
MyListSet	3
MyListSetMain	1
add	4,5,6
class	1
main	2
new	3
public	1,2
s	3,4,5,6
static	2
void	2

Las claves son las palabras (en orden)

Los valores son cjtos. de líneas (en orden)

```
Map<String, TreeSet<Integer>> data = new TreeMap<String,TreeSet<Integer>>()
```