Colecciones e iteradores

Algoritmos y Estructuras de Datos

Departamento de Computación, FCEyN, UBA

9 de mayo de 2024

Colecciones

Una **colección** representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

Colecciones

Una **colección** representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

Conocemos los siguientes tipos de colecciones:

- Secuencia
- Conjunto
- Multiconjunto
- Diccionario

Colecciones

Una **colección** representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

Conocemos los siguientes tipos de colecciones:

- Secuencia
- Conjunto
- Multiconjunto
- Diccionario
- ¿Vector?

Colecciones

Una **colección** representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

Conocemos los siguientes tipos de colecciones:

- Secuencia
- Conjunto
- Multiconjunto
- Diccionario
- ~~¿Vector?~~ El vector es una posible implementación del TAD Secuencia

Operaciones sobre colecciones

Preguntas típicas sobre colecciones:

- Dado un elemento, ¿está en la colección?
  `pertenece`(x, conj)
  `esta`(dicc, x)
- Listar todos los elementos de una colección.
- Encontrar el elemento más chico de la colección.
- *etc.*

Conjunto

Colección (finita) de elementos sin distinguir orden ni multiplicidad.

```
TAD Conjunto<T> {
    obs elems: conj<T>

    proc conjVacio(): Conjunto<T>

    proc pertenece(in c: Conjunto<T>, in T e): bool

    proc agregar(input c: Conjunto<T>, in e: T)

    proc sacar(inout c: Conjunto<T>, in e: T)

    proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
    ...
}
```

Conjunto (Java)

```
public interface Set<T> extends Collection<T>{
    boolean contains(T elem); // Determina si el conjunto
                              // contiene a elem
    boolean add(T elem); // Agrega a elem al conjunto
    boolean remove(T elem); // Quita a elem del conjunto
    int size(); // Devuelve el tamaño del conjunto
};
```

Conjunto (Java): ejemplo de uso

```java
import java.util.Set;
import java.util.HashSet;
public class EjemploConjunto {
  public static void main(String[] args)
  {
    Set<String> s = new HashSet<String>();
    s.add("Hola");
    s.add("mundo");
    s.add("Hola");
    System.out.println(s.contains("Hola")); // true
    System.out.println(s.contains("Chau")); // false
    System.out.println(s.size()); // 2
    System.out.println(s); // [Hola, mundo]
  }
}
```

Interfaz de Collection según oracle

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

---

public interface **Collection<E>**
extends Iterable<E>

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose Collection implementation classes (which typically implement Collection indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose Collection implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw UnsupportedOperationException if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the collection. For example, invoking the addAll(Collection) method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Métodos de Collection según oracle

All Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Ensures that this collection contains the specified element (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns true if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.
int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns true if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size()** Returns the number of elements in this collection.

Diccionario

Tabla que asocia *claves* a *significados*.

```
TAD Diccionario<K, V> {
    obs m: dict<K, V>

    proc diccionarioVacio(): Diccionario<K, V>

    proc esta(d: Diccionario<K, V>, k: K): bool

    proc definir(inout d: Diccionario<K, V>,
                 in k: K k, in v: V)

    proc obtener(in d: Diccionario<K, V>, in k: K): V
    ...
}
```

Diccionario (Java)

```
public interface Map<K, V> {
  boolean containsKey(K clave); // Determina si el dicc
                                // contiene a clave
  V get(K clave); // Obtiene el valor asociado a clave
  V put(K clave, V valor); // Define el par clave-valor
};
```

Diccionario (Java): ejemplo de uso

```
import java.util.Map;
import java.util.HashMap;
public class EjemploDiccionario {
  public static void main(String[] args)
  {
    Map<Integer, String> seleccion = new
↪   HashMap<Integer, String>();
    seleccion.put(9, "Álvarez");
    seleccion.put(7, "De Paul");
    seleccion.put(10, "Messi");
    seleccion.put(19, "Otamendi");

    System.out.println(seleccion.get(10)); // Messi
    System.out.println(seleccion.containsKey(21));
    // false
    System.out.println(seleccion);
    // {19=Otamendi, 7=De Paul, 9=Alvarez, 10=Messi}
    }
}
```

Tipos paramétricos

```
public interface Set<T> extends Collection<T> {...}
public interface Map<K,V> {...}
```

¿Qué tipos de datos son `T`, `K` y `V`?

Tipos paramétricos

```
public interface Set<T> extends Collection<T> {...}
public interface Map<K,V> {...}
```

¿Qué tipos de datos son `T`, `K` y `V`?
Se los llaman **tipos paramétricos:** constituyen *variables de tipo*.

- Es decir, `T`, `K` y `V` pueden tomar como valor cualquier tipo.
- Nos permiten definir una interfaz *genérica*, que puede ser implementada por distintos tipos de datos.

Tipos paramétricos

```
public interface Set<T> extends Collection<T> {...}
public interface Map<K,V> {...}
```

¿Qué tipos de datos son `T`, `K` y `V`?
Se los llaman **tipos paramétricos:** constituyen *variables de tipo*.

- Es decir, `T`, `K` y `V` pueden tomar como valor cualquier tipo.
- Nos permiten definir una interfaz *genérica*, que puede ser implementada por distintos tipos de datos.
- No obstante, si nuestra implementación requiere de un orden, entonces `T` debe ser *comparable* (i.e., definir una relación de orden total).
  - Esto se logra a través de la interfaz `Comparable<T>`, *sobrecargando* el método `CompareTo(T otro)`.

Recorriendo colecciones

¿Cómo recorremos una colección?

Recorriendo colecciones

¿Cómo recorremos una colección?

```
import java.util.*;
class RecorriendoColecciones
{
  public static void main(String[] arg)
  {
    List<String> seleccion = new Vector<String>();
    seleccion.add("Messi");
    seleccion.add("Di María");
    for (String jugador : seleccion)
      System.out.println(jugador);

    Map<Integer, String> seleccion = new HashMap<Integer, String>();
    seleccion.put(10, "Messi");
    seleccion.put(11, "Di María");

    for (Map.Entry<Integer, String> jugador : seleccion.entrySet())
      System.out.println(jugador.getKey().toString() + " " +
↪  jugador.getValue());
    } // 10 Messi
      // 11 Di María
}
```

Recorriendo colecciones

No obstante, la **estructura subyacente** a una colección puede estar implementada de muchas maneras distintas.

- Esta estructura es **privada** y, por lo tanto, invisible para el usuario.

Recorriendo colecciones

No obstante, la **estructura subyacente** a una colección puede estar implementada de muchas maneras distintas.

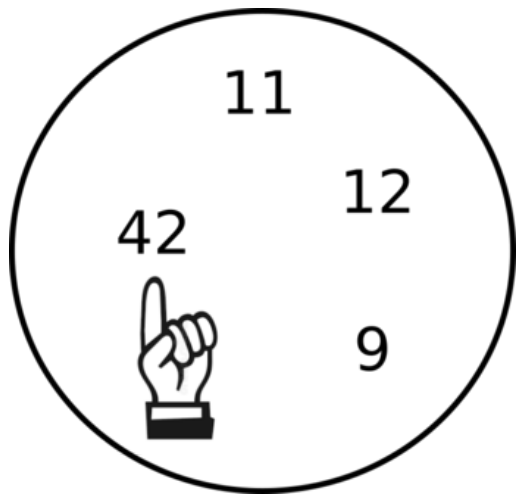
- Esta estructura es **privada** y, por lo tanto, invisible para el usuario.
- Entonces... ¿cómo podemos recorrer una colección sin conocer su estructura?

Iteradores

Un **iterador** es una manera abstracta de recorrer colecciones, independientemente de su estructura.

Informalmente

$$\text{iterador} = \text{colección} + \text{dedo}$$



Iteradores

Operaciones con iteradores:

- ¿Está posicionado sobre un elemento?
- Obtener el elemento actual.
- Avanzar al siguiente elemento.
- Retroceder al elemento anterior. *(Bidireccional)*

Iteradores en Java

Como corresponde, Java provee de una interfaz para iteradores:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
}
```

- *Obtener* y *avanzar* se combinan en el método `next()`.
- Nosotros vamos a ser los responsables de implementarlo sobre nuestra estructura de datos.

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext(); → true
it.next();

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next(); $\rightarrow$ 1

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next(); $\rightarrow$ 2

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next(); $\rightarrow$ 0

Interpretando al iterador en una secuencia

1	2	0	3	0

```
Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next(); → 3
```

Interpretando al iterador en una secuencia

1	2	0	3	0

```
Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next();
it.next(); → 0
```

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next();
it.next();
it.hasNext(); → false

Implementando el iterador de Vector

Supongamos la siguiente implementación de la clase `Vector`:

```
public class Vector<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

Implementando el iterador de Vector

Supongamos la siguiente implementación de la clase `Vector`:

```
public class Vector<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

¿Dónde va a estar implementado `Iterador`?

Implementando el iterador de Vector

Supongamos la siguiente implementación de la clase `Vector`:

```
public class Vector<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

¿Dónde va a estar implementado `Iterador`?
¡Dentro de la clase! ¿Por qué?

Implementando el iterador de Vector

```
public class Vector<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  private class Iterador implements Iterator<T>{
    ...
  }
}
```

Implementación del iterador de Vector

```
public class Vector<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  private class Iterador implements Iterator<T>{
      int dedito;
      Iterador(){
        dedito = 0;
      }
      public boolean hasNext(){
        return dedito != size;
      }
      public T next(){
        int i = dedito;
        dedito = dedito + 1;
        return elementos[i];
      }
    }
  }
```

Usando nuestro iterador

```
import java.util.Vector;
import java.util.Iterator;
public class VectorIteratorExample {
  public static void main(String[] args) {
     Vector<String> vector = new Vector<String>();

     vector.add("Manzana");
     vector.add("Naranja");
     vector.add("Durazno");

     // Ahora podemos usar for-each!
     for (String fruit : vector) {
       System.out.println(fruit);
     }

     Iterator it = vector.iterator();
     while(it.hasNext()){
       System.out.println(it.next());
     }
  }
}
```

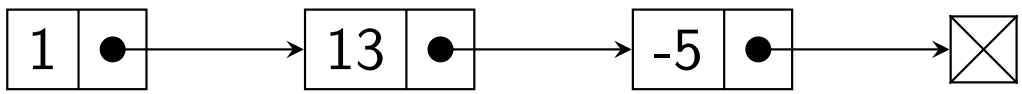
Listas simplemente enlazadas

Listas simplemente enlazadas

Una *lista simplemente enlazada* es una estructura que sirve para representar una secuencia de elementos, distinta del vector.

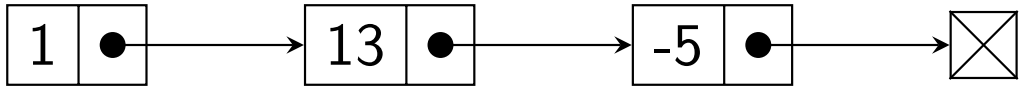
```
public interface List<T> extends Collection<T> {...}
public class Vector<T> implements List<T> {...}
public class LinkedList<T> implements List<T> {...}
```

Gráficamente



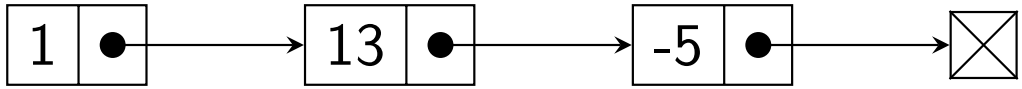
Cada elemento de la secuencia se representa mediante un *nodo*, que contiene un elemento y una referencia al siguiente nodo.

Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Cuál es su invariante de representación?

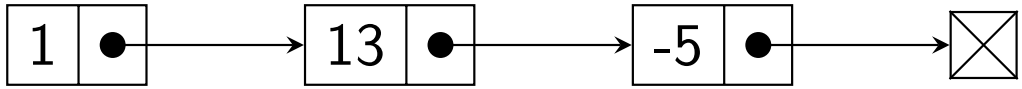
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Cuál es su invariante de representación?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.

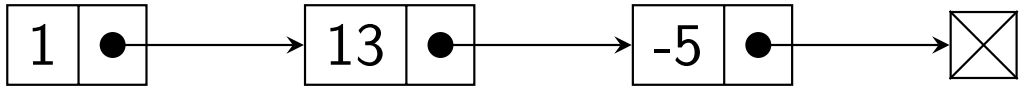
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Cuál es su invariante de representación?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.

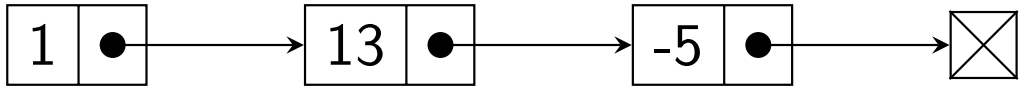
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Cuál es su invariante de representación?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.
3. Todos los nodos de la lista apuntan al siguiente, excepto el último.

Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Cuál es su invariante de representación?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.
3. Todos los nodos de la lista apuntan al siguiente, excepto el último.
4. El último nodo apunta a `null`.

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?

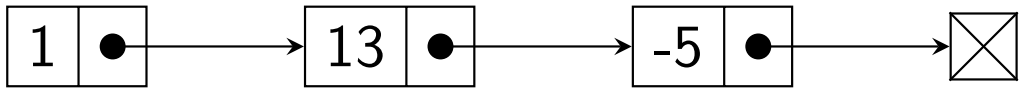
Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?
Perdemos el *acceso aleatorio* a los elementos.

Lista de Enteros



Implementemos la clase `ListaDeInts`, sobre una lista simplemente enlazada, con los siguientes métodos:

```
class ListaDeInts implements SecuenciaDeInts {
      private ...

      ListaDeInts();
      ListaDeInts(ListaDeInts otro);
      void agregarAtras(int elem);
      void agregarAdelante(int elem);
      void eliminar(int indice);
      ...
}
```

Lista de Enteros: estructura y constructores

```
class ListaDeInts implements SecuenciaDeInts {
    private Nodo primero;

    private class Nodo {
        int valor;
        Nodo sig;

        Nodo(int v) { valor = v; }
    }

    public ListaDeInts() {
        primero = null;
    }

    public ListaDeInts(ListaDeInts otra) {
        Nodo actual = otra.primero;
        while (actual != null) {
            agregarAtras(actual.valor);
            actual = actual.sig;
        }
    }
```

Lista de Enteros: agregando elementos

```
public void agregarAdelante(int elem) {
    Nodo nuevo = new Nodo(elem);
    nuevo.sig = primero;
    primero = nuevo;
}

public void agregarAtras(int elem) {
    Nodo nuevo = new Nodo(elem);
    if (primero == null) {
        primero = nuevo;
    } else {
        Nodo actual = primero;
        while (actual.sig != null) {
            actual = actual.sig;
        }
        actual.sig = nuevo;
    }
}
```

Lista de Enteros: eliminando un elemento

```
public void eliminar(int i) {
    Nodo actual = primero;
    Nodo prev = primero;
    for (int j = 0; j < i; j++) {
        prev = actual;
        actual = actual.sig;
    }
    if (i == 0) {
        primero = actual.sig;
    } else {
        prev.sig = actual.sig;
    }
}
```

Jerarquía de colecciones en Java