Recursividad estructural

Lista

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

4 de enero de 2021

Definiciones

Temas

Lista

Definición (Lista)

Una *lista* es una secuencia de cero a más elementos de un tipo determinado T. Se representa como una sucesión de elementos separados por comas:

$$a_0, a_1, ..., a_{n-1}$$

donde $n \geqslant 0$ y cada a_i es de tipo T. Una lista sin elementos, con $n = 0$ es una *lista vacía*.

Ejemplos de listas

- $\emptyset$.
- $\underset{0}{\text{Perro}}$.
- $\underset{0}{\text{Gato}}, \underset{1}{\text{Perro}}$.
- $\underset{0}{\text{Bananas}}, \underset{1}{\text{Manzanas}}, \underset{2}{\text{Toronjas}}, \underset{3}{\text{Uvas}}, \underset{4}{\text{Peras}}$.

Temas

Definición recursiva de lista

Definición (Lista)

Una *lista* es:

1. Una lista vacía $\emptyset$.
2. Un dato seguido de una **lista**.

Ejemplos de listas serían:

- $\emptyset$.
- "Perro" $\rightarrow \emptyset$.
- "Gato" $\rightarrow$ "Perro" $\rightarrow \emptyset$.
- "Bananas" $\rightarrow$ "Manzanas" $\rightarrow$ "Toronjas" $\rightarrow$ "Uvas" $\rightarrow$ "Peras" $\rightarrow \emptyset$.

Temas

Recursividad estructural

- Una estructura definida en forma recursiva se puede programar utilizando *recursividad estructural*.
- Para ello la estructura contendrá **referencias** a objetos de su mismo tipo.
- La traducción a lenguajes funcionales es inmediata, pero con los orientados a objetos es un poco más laborioso. Por ello comencemos con una traducción lo más literal posible y luego añadiremos el encapsulamiento.

Lista recursiva en Java

Código 1: Lista recursiva

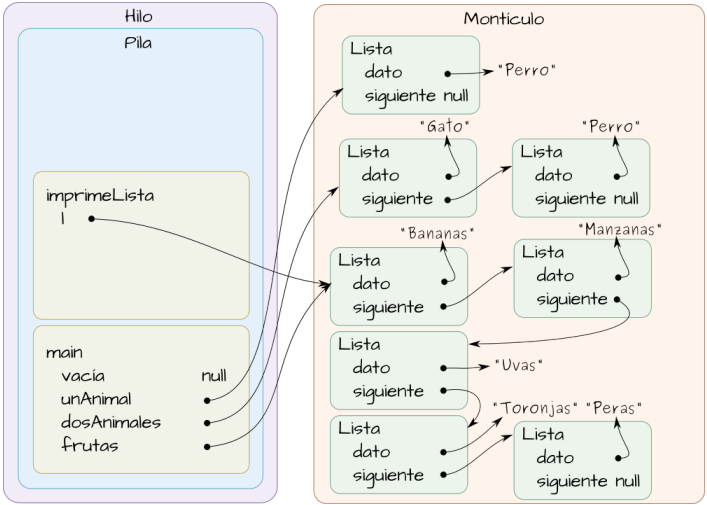
```
1  public class Lista {
2    private Object dato;
3    private Lista siguiente;
4
5    /** Construye una lista con un dato, seguida de otra lista. */
6    public Lista(Object dato, Lista siguiente) {
7      this.dato = dato;
8      this.siguiente = siguiente;
9    }
10
11   public Object getDato() {
12     return dato;
13   }
14
15   /** Versión funcional de un método que trabaja con la lista. */
16   public static void imprimeLista(Lista l) {
17     if(l == null) return;           // Caso base, escrito explícitamente.
18     else {
19       System.out.println(l.dato);
20       imprimeLista(l.siguiente);
21     }
22   }
23 }
```

Construcción manual de listas

- $\emptyset$.
- “Perro” $\rightarrow \emptyset$.
- “Gato” $\rightarrow$ “Perro” $\rightarrow \emptyset$.
- “Bananas” $\rightarrow$ “Manzanas” $\rightarrow$ “Toronjas” $\rightarrow$ “Uvas” $\rightarrow$ “Peras” $\rightarrow \emptyset$.

Código 2: Uso de listas

```
1  public class UsoLista {
2    public static void main(String[] args) {
3      Lista vacía = null;
4      Lista unAnimal = new Lista("Perro", null);
5      Lista dosAnimales  = new Lista("Gato", new Lista("Perro", null));
6      Lista frutas = new Lista("Bananas",
7                      new Lista("Manzanas",
8                       new Lista("Toronjas",
9                        new Lista("Uvas", new Lista("Peras", null)))));
10     imprimeLista(frutas);
11   }
12 }
```


Hilo
Pila
imprimeLista
l
main
vacía
null
unAnimal
dosAnimales
frutas
Monticulo
Lista
dato
siguiente
null
"Perro"
"Gato"
"Perro"
"Bananas"
"Manzanas"
"Uvas"
"Toronjas"
"Peras"

Implementación con orientación a objetos

Temas

2. Implementación con orientación a objetos
   - Tipo de dato abstracto (TDA)
   - Listas y Nodos
   - Listas y nodos versión iterativa

Datos

Retomamos la primera definición:

Definición (Lista)

Una *lista* es una secuencia de cero a más elementos de un tipo determinado T. Se representa como una sucesión de elementos separados por comas:

$$a_0, a_1, ..., a_{n-1}$$

donde $n \geqslant 0$ y cada a_i es de tipo T. Una lista sin elementos, con $n = 0$ es una *lista vacía*.

Operaciones

Y agregamos las operaciones:

- *construir*: $\emptyset \rightarrow$ ListaVacía. Crea una lista vacía.
- *agregaAlFinal*: Lista, Elemento $\rightarrow$ Lista. Agrega el elemento al final de la lista, si no es nulo.
- *inserta*: Lista, Posición, Elemento $\rightarrow$ Lista. Inserta al elemento (si no es nulo) en la posición indicada, devolviendo una referencia a la misma lista pero modificada. (Versión destructiva)[1]
- *recupera*: Lista, Posición $\rightarrow$ Elemento. Devuelve al elemento en la posición indicada.

[1] También es posible definir estos métodos de forma que se devuelva una nueva lista pero sin el elemento indicado. La ventaja es que no se pierde la lista original, la desventaja es que se utiliza mucha más memoria.

- *borra*: Lista, Elemento $\rightarrow$ Lista. Borra la primer ocurrencia del elemento indicado.
- *busca*: Lista, Elemento $\rightarrow$ Posición. Indica la posición es que se encuentra el elemento indicado o -1 no está.
- *destruir*: Lista $\rightarrow \emptyset$. Vacía la lista.
- *imprime*: Lista $\rightarrow \emptyset$. Imprime el contenido de la lista.

Temas

2. Implementación con orientación a objetos
   - Tipo de dato abstracto (TDA)
   - Listas y Nodos
   - Listas y nodos versión iterativa

Listas simplemente ligadas

Para encapsular los detalles de la implementación del TDA lista, se definen ahora dos clases:

1. `Lista`, que presentará la interfaz pública.
2. `Nodo`, donde se hará la programación recursiva.

De modo que las operaciones anteriores se pueden declarar en una interfaz y el programador elige cómo desea implementarlas.

IListaa

Código 3: Interfaz IListaa

```
1  public inteface IListaa {
2    // La interfaz no puede definir el contructor.
3    public IListaa agregaAlFinal(Object elemento);
4    public IListaa inserta(int pos, Object elemento);
5    public Object recupera(int pos);
6    public IListaa borra(Object elemento);
7    public int busca(Object elemento);
8    public IListaa destruir();
9    public void imprime();
10 }
```

Lista simplemente ligada

- La forma más sencilla de programar una lista en `Java` es como una *lista simplemente ligada*.
- En esta versión, la estructura recursiva es un `Nodo` que contiene referencias a:
  - El dato
  - La dirección del siguiente `Nodo`.
- La clase `ListaSimple` sólo tiene la referencia al primer nodo y desdes ahí se realizan todas las operaciones. A este primer nodo se le llama **cabeza**.

Uso de una IList

Código 4: Uso de una IList

```
1  public class UsoLista {
2    public static void main(String[] args) {
3      ILista l = new ListaSimple();   // Crea lista vacía.
4      l.agregaAlFinal("Bananas");
5      l.agregaAlFinal("Manzanas");
6      l.agregaAlFinal("Toronjas");
7      l.agregaAlFinal("Uvas");
8      l.agregaAlFinal("Peras");
9      l.imprimeLista();
10
11     // Aprovecha que agregaAlFinal devuelve una referencia a esta lista.
12     ILista l2 = new ListaSimple();
13     l2.agregaAlFinal("Gato").agregaAlFinal("Perro").agregaAlFinal("Canario");
14     l2.imprimeLista();
15   }
16 }
```

Clase ListaSimple

Código 5: ListaSimple

```
1  public class ListaSimple implements ILista {
2    private Nodo cabeza;
3
4    /** Construye la lista vacía. */
5    public ListaSimple() { cabeza = null; }
6
7    public ILista agregaAlFinal(Object elemento) {
8      if (elemento == null) throw IllegalArgumentException();  // Precondición: no se aceptan datos nulos.
9      // Agregar si la lista es vacía
10     if (cabeza == null) {
11       cabeza = new Nodo(elemento);
12     } else cabeza.agrega(elemento);
13     return this;
14   }
15 }
```

Clase Nodo

Código 6: Nodo

```
1  /** Sólo se accede a esta clase y sus métodos dentro del paquete. */
2  class Nodo {
3    private Object elemento;
4    private Nodo siguiente;
5
6    // Constructores, se asume que elemento no es null.
7    Nodo(Object elemento) { this.elemento = elemento; }
8    Nodo(Object elemento, Nodo siguiente) {
9      this.elemento = elemento;
10     this.siguiente = siguiente;
11   }
12
13   Object getElemento() { return elemento; } // Getter (método de lectura)
14   Nodo getSiguiente() { return siguiente; } // Getter
15
16   void setSiguiente(Nodo siguiente) { this.siguiente = siguiente; } // Setter (método de escritura)
17
18   agrega(Object elemento) {                  // Se asume que elemento no es null
19     if (this.siguiente == null) {            // Caso base
20       this.siguiente = new Nodo(elemento);
21     } else siguiente.agrega(elemento);       // Llamada recursiva
22   }
23 }
```

Insertar

- Para insertar un dato en cualquier posición de una lista ligada es necesario recorrer todos los elementos desde la **cabeza** hasta llegar al elemento anterior a la posición donde se realizará la inserción.
- La complejidad en tiempo es proporcional a la posición del nuevo elemento.
  - En el **mejor caso** es constante, cuando se incerta al inicio de la lista (posición 0).
  - El **peor caso** es cuando se inserta al final, pues se recorre toda la lista, entonces la complejidad es n, la longitud de la lista. [2]
  - En el **caso promedio**, la complejidad es, en promedio, la mitad de la longitud de la lista, lo cual también se considera $\mathcal{O}(n)$.

[2] Si agregar al final es una operación común se puede agregar otro atributo que apunte al final de la lista, para que se pueda realizar la inserción en tiempo constante.

Código para insertar

Código 7: Insertar en ListaSimple

```
1  public class ListaSimple implements ILista {
2    private Nodo cabeza;
3    ...
4
5    public IListaa inserta(int pos, Object elemento) {
6      if (pos < 0) throw new IndexOutOfBoundsException("pos␣=␣" + pos);
7      if (elemento == null) throw new IllegalArgumentException();
8      if (pos == 0) {   // Caso especial: pos = 0, inserta en la cabeza.
9        cabeza = new Nodo(elemento, cabeza);
10     } else {
11       if (cabeza == null) throw new IndexOutOfBoundsException("pos␣=␣" + pos);
12       cabeza.inserta(pos, elemento, 0); // La cabeza está en la posición 0.
13     }
14     return this;
15   }
16 }
```

Código 8: Insertar en Nodo

```
1   class Nodo {
2     private Object elemento;
3     private Nodo siguiente;
4     ...
5     void inserta(int pos, Object elemento, int índice) {
6       if (pos == índice + 1) {  // Caso base: elemento va después de este nodo.
7         siguiente = new Nodo(elemento, siguiente);
8       } else {
9         if (siguiente == null) throw new IndexOutOfBoundsException("índice = " + índice);
10        else siguiente.inserta(pos, elemento, índice + 1);
11      }
12    }
13
14  }
```

Borrar

Para borrar al elemento indicado debemos realizar varios pasos:

- Encontrar el nodo donde está guardado.
- Necesitaremos la dirección del nodo que está antes de él, porque ahora ese nodo apuntará al que esté después del que queremos borrar.
- Desconectar al nodo que queremos borrar.
- ¡Ojo! Si el que queremos borrar es la cabeza, es un caso particular.

Equals e ==

Ahora que se buscarán elementos iguales es momento de señalar una presición:

- El operador == en Java nos indica si dos variables hacen referencia al mismo objeto. Si dos objetos contienen la misma información, de todos modos devolverá falso.

```
1  String a1 = "Cadena";
2  String b1 = a1;
3  boolean misma1 = (a1 == b1); // misma1 es True
4
5  String a2 = "Chocolate";
6  String b2 = "Chocolate";
7  boolean misma2 = (a2 == b2); // misma2 es False
```

Equals

- Para comparar el contenido se usa el método `equals`.

```
1 boolean misma3 = a1.equals(b1); // misma3 es True
2 boolean misma4 = a2.equals(b2); // misma4 es True
```

- Cuidado, para mandar llamar un método, el objeto no debe ser `null`.

```
1 String s = null;
2 boolean misma5 = s.equals(null); // Lanza NullPointerException
```

- Por ello se debe verificar siempre si se desea detectar al mismo objeto o a objetos que contengan los mismos datos para elegir qué método usar.

Código para borrar

Código 9: Borrar en ListaSimple

```
1  public class ListaSimple implements ILista {
2    private Nodo cabeza;
3    ...
4
5    public ILista borra(Object elemento) {
6      if (elemento == null) throw new IllegalArgumentException();
7      if (cabeza == null) return this; // Elemento no está en la lista vacía.
8      else if (cabeza.getDato().equals(elemento)) {
9        Nodo temp = cabeza;
10       cabeza = cabeza.getSiguiente();  // Brinca
11       temp.setSiguiente(null);         // Limpia
12     } else cabeza.borra(elemento);     // Dato borrar, this es la referencia al anterior
13     return this;
14   }
15 }
```

Código 10: Borrar en Nodo

```
1  class Nodo {
2    private Object elemento;
3    private Nodo siguiente;
4    ...
5
6    void borra(Object elemento) {
7      if (siguiente == null) return;            // Elemento no estaba
8      // Caso base, el elemento está adelante de mí.
9      if (siguiente.elemento.equals(elemento)) {
10       Nodo temp = siguiente;                  // El que borraré
11       siguiente = temp.siguiente;             // Brinca
12       temp.siguiente = null;                  // Limpia
13     } else siguiente.borra(elemento);         // Llamada recursiva
14   }
15 }
```

Temas

2. Implementación con orientación a objetos
   - Tipo de dato abstracto (TDA)
   - Listas y Nodos
   - Listas y nodos versión iterativa

Versión iterativa

- Aunque la estructura es recursiva, también es posible implementar sus métodos de forma iterativa.
- Frecuentemente la implementación iterativa consumirá menos recursos computacionales que la recursiva.

Clase Lista con métodos iterativos I

Código 11: ListaSimple

```
1  /** Clase lista donde un nodo contiene referencias al dato y el siguiente nodo. */
2  public class ListaSimple implements ILista {
3    /** Referencia al primer nodo. */
4    private Nodo cabeza;
5
6    /** Construye la lista vacía. */
7    public ListaSimple() { cabeza = null; }
8
9    /** Agrega el elemento al final de la lista y devuelve esta lista. */
10   public ILista agregaAlFinal(Object elemento) {
11     // Agregar si la lista es vacía
12     if (cabeza == null) {
13       cabeza = new Nodo(elemento);
14     } else {
15       Nodo temp = cabeza; // Muy importante usar variable auxiliar.
16       while(temp.siguiente != null) { // Obtener dirección del último nodo.
17         temp = temp.siguiente;
18       }
19       temp.siguiente = new Nodo(elemento); // El nuevo elemento va al final del último que teníamos.
20     }
21   }
22
```

Clase Lista con métodos iterativos II

```
23    /** Inserta al elemento en la posición indicada y devuelve esta lista. */
24    public IListaa inserta(int pos, Object elemento) {
25      if (elemento == null) throw new IllegalArgumentException();
26      if (pos == 0) {
27        cabeza = new Nodo(elemento, cabeza);
28        return this;
29      }
30      int índice = 0;
31      Nodo temp = cabeza;
32      while(índice < pos - 1) {       // Recorrerse al siguiente nodo
33        temp = temp.getSiguiente();
34        índice++;                     // Índice del nodo temp
35        if (temp == null) throw new IndexOutOfBoundsException();
36      }
37      temp.setSiguiente(new Nodo(elemento, temp.getSiguiente()));
38      return this;
39    }
40
41    /** Elimina de la lista la primer aparición del elemento indicado. */
42    public IListaa borra(Object elemento) {
43      if (elemento == null) throw new IllegalArgumentException();
44      if (cabeza == null) return this; // Elemento no está en la lista vacía.
45      else if (cabeza.getDato().equals(elemento)) {
46        Nodo temp = cabeza;
47        cabeza = cabeza.getSiguiente();       // Brinca
```

Clase Lista con métodos iterativos III

```
48          temp.setSiguiente(null);              // Limpia
49        } else {
50          Nodo anterior = cabeza;
51          Nodo actual = cabeza.getSiguiente();
52          while(actual != null) {               // Busca elemento
53            if (actual.getDato().equals(elemento)) {    // Borrar
54              anterior.setSiguiente(actual.getSiguiente());  // Brinca
55              actual.setSiguiente(null);                  // Limpia
56              return this;                                // Terminamos
57            } else {
58              anterior = actual;
59              actual = actual.getSiguiente();
60            }
61          }
62          // Si llega aquí el objeto no estaba
63        }
64        return this;
65      }
66    }
```

Licencia