Sesión 15: Cadena simplemente enlazada (2/2)

Hoja de problemas

Programación 2

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

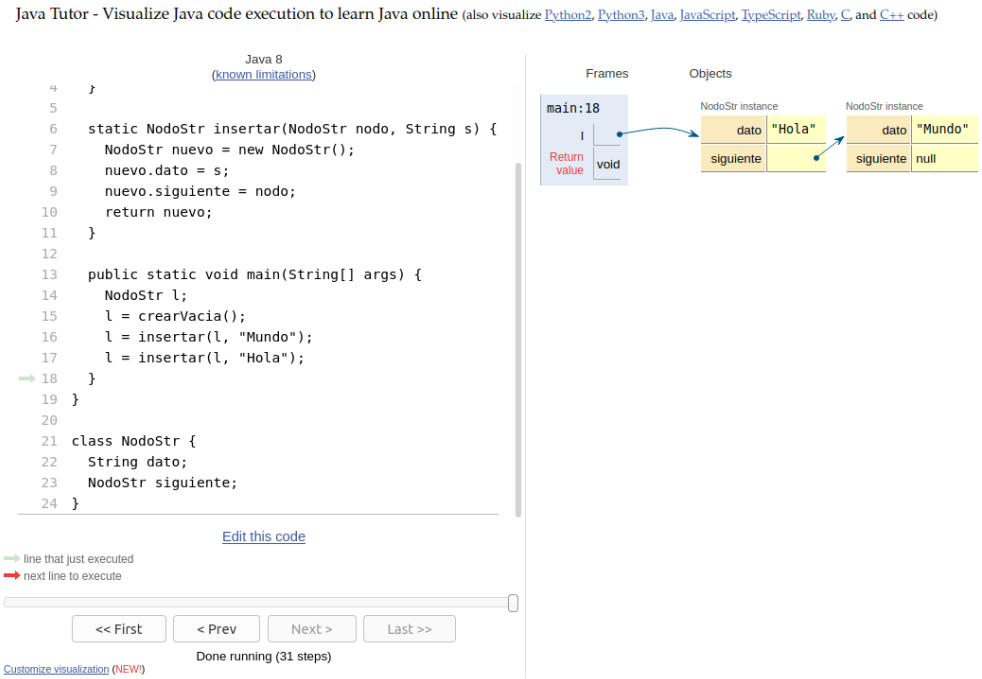
2024-2025

Ejercicio 1. Repasa las transparencias de la sesión 14 sobre cadenas enlazadas. En esta sesión tendrás que implementar todas las *operaciones* sobre la clase `NodoStr` propuestas en dichas transparencias.

Ejercicio 2. El objetivo de estos ejercicios es que entiendas la estructura de datos de las cadenas simplemente enlazadas. Para ello, vas a preparar un único fichero que podrás tanto en tu máquina como en una herramienta de visualización de ejecución de programas. La herramienta en cuestión es **JavaVisualizer**[1], una herramienta Web que puedes usar desde

`https://cscircles.cemc.uwaterloo.ca/java_visualize/`

La herramienta te muestra el estado de tu programa paso y podrás ver pantallas como esta, en la que podrás observar cómo se construyen las cadenas y detectar errores en tu código:



[1] Basada en Java Tutor (`http://pythontutor.com/java.html`).

NodoStr **Ejercicio 3.** Antes de empezar vamos a preparar nuestro código tanto para compilarlo localmente como para subirlo a **JavaVisualizer**. Crea un fichero `OperacionesNodo.java` y transcribe el siguiente código:[2]

```
// Clase para representar los nodos de una cadena enlazada
class NodoStr {
  String dato;
  NodoStr siguiente;
}

// Clase funciones (static) que manejan cadenas enlazadas + main para ''jugar'' con las mismas
public class OperacionesNodo {
  public static void main(String[] args) {
    NodoStr l = null;
  }
}
```

Conéctate a `https://cscircles.cemc.uwaterloo.ca/java_visualize/`, copia y pega el texto y pon en marcha la visualización con el botón *Visualize execution*.

Ejercicio 4. Recuerda que estamos usando la clase `NodoStr` para representar *cadenas enlazadas de strings* y que conceptualmente dicha estructura de datos vienen a representar listas de *strings*. En este documento, normalmente usaremos la palabra *cadena* para referirnos a las variables e instancias del tipo `NodoStr`.

crearVacia **Ejercicio 5.** En `OperacionesNodo`, implementa la función[3] `crearVacia` que devuelva una cadena vacía y añade un `main` para poder probarlo. Ejecuta con JavaVisualizer.

```
public class OperacionesNodo {
  static NodoStr crearVacia() {
    return null;
  }

  public static void main(String[] args) {
    NodoStr l;
    l = crearVacia();
  }
}
```

esVacia **Ejercicio 6.** Añade (a `OperacionesNodo`) la función `esVacia` siguiendo esta signatura:

```
  static boolean esVacia(NodoStr cadena)
```

Modifica el `main` para imprimir el resultado de `esVacia` sobre una lista vacía. Ejecuta con JavaVisualizer.

```
  public static void main(String[] args) {
    NodoStr l;
    l = crearVacia();
    System.out.println(esVacia(l));
  }
```

[2] Puede que te llame la atención ver dos clases en el mismo fichero fuente. No es lo habitual aunque Java lo admite (siempre y cuando sólo una de las clases sea pública), y va a ser de gran ayuda a la hora de usar JavaVisualizer ya que sólo admite un único fichero.

[3] Una función es un *método de clase*, es decir **`static`**.

insertarP **Ejercicio 7.** Empieza el espectáculo: implementa la función `insertarP`. La función toma como argumentos una cadena y un *string* y devuelve una cadena cuyo primero es el *string* recibido y el resto es la cadena recibida. La signatura que debes respetar es:

```
static NodoStr insertarP(NodoStr cadena, String s)
```

Prueba a ampliar el `main` para añadir un primer elemento `"B"` y luego otro primer elemento `"A"` para que finalmente `l` sea la lista `["A", "B"]`. Ejecuta con JavaVisualizer.

```
public static void main(String[] args) {
  NodoStr l;
  l = crearVacia();
  System.out.println(esVacia(l));
  l = insertarP(l, "B");
  l = insertarP(l, "A");
  System.out.println(esVacia(l));
}
```

insertarU **Ejercicio 8.** Implementa la función `insertarU`. La función toma como argumentos una cadena y un *string* y devuelve *la cadena* añadiendo el *string* al final de la misma como último elemento. La signatura que debes respetar es:

```
static NodoStr insertarU(NodoStr cadena, String s)
```

Prueba a ampliar el `main` para añadir como último elemento elemento `"C"` para que finalmente `l` sea la lista `["A", "B", "C"]`.

```
public static void main(String[] args) {
  NodoStr l;
  l = crearVacia();
  System.out.println(esVacia(l));
  l = insertarP(l, "B");
  l = insertarP(l, "A");
  System.out.println(esVacia(l));
  l = insertarU(l, "C");
}
```

imprimir **Ejercicio 9.** Implementa la función `imprimir`. La función toma como argumento una cadena y la *imprime* en la salida estádar (`System.out`). La signatura que debes respetar es:

```
static void imprimir(NodoStr cadena)
```

Para implementar dicha función implementa antes la función `formatear` que devuelva el *string* que representa la cadena. La signatura que debes respetar es:

```
static String formatear(NodoStr cadena)
```

La forma en la que deberá formatearse la cadena deberá ser compatible con la representación *matemática* de listas vista en clase (corchetes y datos separados por comas). Ejemplos:

`[]`	lista vacía
`["Hola"]`	lista con un *string*
`["Hola","mundo"]`	lista con dos *strings*

Prueba a imprimir la cadena `l` del ejemplo anterior:

```
public static void main(String[] args) {
  NodoStr l;
  l = crearVacia();
  System.out.println(esVacia(l));
  l = insertarP(l, 3);
  l = insertarP(l, 2);
  System.out.println(esVacia(l));
  l = insertarU(l, 5);
  imprimir(l);
}
```

En pantalla deberías ver

```
["A", "B", "C"]
```

Imprimir **Ejercicio 10.** Prueba tu implementación de la función `imprimir`:

```
l = crearVacia()
for(i = 0; i < 10; i++) {
  l = insertarU(l, Integer.toString(i))
}
imprimir(l);
```

En pantalla deberías ver:

```
["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
```

`longitud` **Ejercicio 11.** Implementa la función `longitud`. La función toma como argumento una cadena y devuelve su longitud. La signatura que debes respetar es:

```
static int longitud(NodoStr cadena)
```

Aserciones **Ejercicio 12.** A estas alturas ya estás en disposición de añadir pruebas muy interesantes como por ejemplo esta:

```
l = crearVacia();
for(i = 0; i < 10; i++) {
  l = insertarU(l, Integer.toString(i))
}
n = longitud(l);
Assert.assertEquals(n, 10);
```

Te toca recuperar el código de la *minibiblioteca* `Assert` (ver anexo al final).

JavaVisualizer **Ejercicio 13.** Abre la URL `https://cscircles.cemc.uwaterloo.ca/java_visualize/`, copia y pega el fichero `OperacionesNodo.java` (quita la aserción) y *visualiza la ejecución* (botón *Visualize Execution*). Podrás avanzar en la ejecución de tu programa paso a paso y a la derecha de tu código podrás ver una representación del estado de tu programa: variables, instancias, referencias, etc.

Presta mucha atención a los cambios de estado que provoca la ejecución de cada sentencia e intenta detectar errores en tu código.

`primero` **Ejercicio 14.** Implementa la función `primero`. La función toma como argumento una cadena y devuelve el primer nodo de la cadena. Si la cadena está vacía la función puede fallar. La signatura que debes respetar es:

```
static String primero(NodoStr cadena)
```

ultimo **Ejercicio 15.** Implementa la función `ultimo`. La función toma como argumento una cadena y devuelve el último nodo de la cadena. Si la cadena está vacía la función puede fallar. La signatura que debes respetar es:

```
static String ultimo(NodoStr cadena)
```

borrarP **Ejercicio 16.** Implementa la función `borrarP`. La función toma como argumento una cadena y devuelve una cadena eliminando el primer nodo. La signatura que debes respetar es:

```
static void borrarP(NodoStr cadena)
```

borrarU **Ejercicio 17.** Implementa la función `borrarU`. La función toma como argumento una cadena y devuelve una cadena eliminando el último nodo. La signatura que debes respetar es:

```
static void borrarU(NodoStr cadena)
```

Más pruebas **Ejercicio 18.** No dejes de añadir más *aserciones* a tu programa principal para probar que tus funciones están correctamente implementadas. De hecho, recuerda que hacer *test driven development* es una muy buena práctica así que. . . **empieza por los tests**.

Recuerda que las cadenas son una estructura recursiva y que el caso *base* es **`null`**. No dejes de probar las funciones cuando tienen que trabajar **con listas vacías** o listas con un elemento.

Ejercicio 19. Cada vez que implementes una función, vuelve al visualizador JavaVisualizer y asegúrate que entiendes lo que tu código hace. Pide ayuda si no entiendes algo.

obtener **Ejercicio 20.** Implementa la función `obtener`. La función toma como argumentos una cadena y un *índice* (un entero entre 0 y la longitud de la cadena menos uno) y devuelve el *string* que ocupa la posición indicada por el índice en la cadena. La signatura que debes respetar es:

```
static String obtener(NodoStr cadena, int i)
```

cambiar **Ejercicio 21.** Implementa la función `cambiar`. La función toma como argumentos una cadena, un índice i y un *string* y cambiar el elemento i-ésimo de la cadena por el *string*. La signatura que debes respetar es:

```
static void cambiar(NodoStr cadena, int i, String s)
```

Ejercicio 22. Observa que la signatura de `cambiar` dice que no devuelve nada (**`void`**). ¿Podría haber sido `static NodoStr cambiar(NodoStr cadena, int i, String s)`?

insertar **Ejercicio 23.** Implementa la función `insertar`. La función toma como argumentos una cadena, un índice i y un *string* y devuelve la cadena tras insertar el *string* en la posición i (dejando la cadena con un elemento más). La signatura que debes respetar es:

```
static NodoStr insertar(NodoStr cadena, int i, String s)
```

💬 **Ejercicio 24.** Observa que la signatura de `insertar` dice que devuelve `NodoStr`.
¿Podría haber sido `static void insertar(NodoStr cadena, int i, String s)`?

`buscar` **Ejercicio 25.** Implementa la función `buscar`. La función toma como argumentos una cadena y un *string* y devuelve un booleano: **`true`** si el string está en la cadena, **`false`** en otro caso. La signatura que debes respetar es:

```
static boolean buscar(NodoStr cadena, String s)
```

`indice` **Ejercicio 26.** Implementa la función `indice`. La función toma como argumentos una cadena y un *string* y devuelve un entero: `-1` si el string no está en la cadena, el índice que ocupa en otro caso. La signatura que debes respetar es:

```
static int indice(NodoStr cadena, String s)
```

`buscar` v2 **Ejercicio 27.** Reimplementa la función `buscar` en función de `indice`.

`borrar` **Ejercicio 28.** Implementa la función `borrar`. La función toma como argumentos una cadena y un índice i y borra el nodo i-ésimo de la cadena. La signatura que debes respetar es:

```
static NodoStr borrar(NodoStr cadena, int i)
```

💬 **Ejercicio 29.** Observa que la signatura de `borrar` dice que devuelve `NodoStr`.
¿Podría haber sido `static void borrar(NodoStr cadena, int i)`?

`buscarYborrar` **Ejercicio 30.** Implementa la función `buscarYborrar`. La función toma como argumentos una cadena y un *string* y devuelve una cadena tras borrar el *string* de la cadena inicial. Si el *string* no está devuelve la misma cadena. La signatura que debes respetar es:

```
static NodoStr buscarYborrar(NodoStr cadena, String s)
```

`insertarO` **Ejercicio 31.** Implementa la función `insertarO`. La función toma como argumentos una cadena y un *string* y devuelve la cadena tras insertar **en orden** el *string* en la cadena inicial. La signatura que debes respetar es:

```
static NodoStr insertarO(NodoStr cadena, String s)
```

`ordenar` **Ejercicio 32.** Ahora vas a tener que implementar una función de ordenación de cadenas enlazadas. La inspiración vendría a partir de que puedes hacer un recorrido de la cadena elemento a elemento (ya has hecho varios como por ejemplo `imprimir` y, uno a uno, usar la función `insertarO` para ir insertando en orden en una cadena nueva. La signatura que debes respetar es:

```
static NodoStr ordenar(NodoStr cadena)
```

La verdad es que el algoritmo sugerido no es muy eficiente, su complejidad será del orden de n^2 (el tiempo de ejecución es proporcional a n^2) donde n es la longitud de `cadena`.

`ordenar` v2 **Ejercicio 33.** En esta ocasión tienes que intentar resolver el problema anterior (32) sin usar para nada la estructura de datos. Es decir, no puedes acceder a `dato` y `siguiente`. La inspiración vendría a partir de que tienes funciones como `primero` y `borrarP`.

⚠ Recapitulando **Ejercicio 34.** A estas alturas tenemos una importante colección de funciones (*métodos static*). Recapitulemos sus *signaturas*:

```
static NodoStr crearVacia()
static boolean esVacia(NodoStr cadena)
static NodoStr insertarP(NodoStr cadena, String s)
static NodoStr insertarU(NodoStr cadena, String s)
static int longitud(NodoStr cadena)
static String formatear(NodoStr cadena)
static void imprimir(NodoStr cadena)
static String primero(NodoStr cadena)
static String ultimo(NodoStr cadena)
static NodoStr borrarP(NodoStr cadena)
static NodoStr borrarU(NodoStr cadena)
static String obtener(NodoStr cadena, int i)
static void cambiar(NodoStr cadena, int i, String s)
static NodoStr insertar(NodoStr cadena, int i, String s)
static boolean buscar(NodoStr cadena, String s)
static int indice(NodoStr cadena, String s)
static NodoStr borrar(NodoStr cadena, int i)
static NodoStr buscarYborrar(NodoStr cadena, String s)
static NodoStr insertarO(NodoStr cadena, String s)
static NodoStr ordenar(NodoStr cadena)
```

⚠ Pruebas **Ejercicio 35.** A continuación te regalo un conjunto de pruebas bastante potentes sobre la colección de funciones:

```
public static void main(String[] args) {
  test1();
  test2();
  test3();
  test4();
  test5();
  test6();
  test7();
  test8();
  test9();
  System.out.println("Todos los tests han pasado");
}

private static void test1() {
  NodoStr l = crearVacia();
  l = insertarP(l, "mundo");
  l = insertarP(l, "hola");
  Assert.assertEquals("hola", obtener(l, 0));
  Assert.assertEquals("mundo", obtener(l, 1));
  Assert.assertEquals(2, longitud(l));
  Assert.assertEquals("hola", primero(l));
  Assert.assertEquals("mundo", ultimo(l));
}

private static void test2() {
  NodoStr l = crearVacia();
  l = insertarU(l, "uno");
  l = insertarU(l, "dos");
```

```
  l = insertarU(l, "tres");
  Assert.assertEquals(3, longitud(l));
  cambiar(l, 1, "DOSS");
  Assert.assertEquals("DOSS", obtener(l, 1));
  l = borrar(l, 1);
  Assert.assertEquals("tres", obtener(l, 1));
}

private static void test3() {
  NodoStr l = crearVacia();
  l = insertarP(l, "c");
  l = insertarP(l, "b");
  l = insertarP(l, "a");
  Assert.assertTrue(buscar(l, "b"));
  Assert.assertFalse(buscar(l, "z"));
  Assert.assertEquals(2, indice(l, "c"));
  Assert.assertEquals(-1, indice(l, "x"));
}

private static void test4() {
  NodoStr l = crearVacia();
  l = insertarP(l, "x");
  l = insertarP(l, "y");
  l = insertarP(l, "z");
  l = buscarYborrar(l, "y");
  Assert.assertEquals("z", obtener(l, 0));
  Assert.assertEquals("x", obtener(l, 1));
  Assert.assertEquals(2, longitud(l));
}

private static void test5() {
  NodoStr l = crearVacia();
  l = insertarO(l, "m");
  l = insertarO(l, "a");
  l = insertarO(l, "z");
  l = insertarO(l, "c");
  Assert.assertEquals("a", obtener(l, 0));
  Assert.assertEquals("c", obtener(l, 1));
  Assert.assertEquals("m", obtener(l, 2));
  Assert.assertEquals("z", obtener(l, 3));

  NodoStr l2 = crearVacia();
  l2 = insertarP(l2, "d");
  l2 = insertarP(l2, "b");
  l2 = insertarP(l2, "a");
  l2 = insertarP(l2, "c");
  l2 = ordenar(l2);
  Assert.assertEquals("a", obtener(l2, 0));
  Assert.assertEquals("b", obtener(l2, 1));
  Assert.assertEquals("c", obtener(l2, 2));
  Assert.assertEquals("d", obtener(l2, 3));
```

```
}

private static void test6() {
  NodoStr vacia = crearVacia();
  Assert.assertTrue(esVacia(vacia));
  Assert.assertEquals(0, longitud(vacia));
  Assert.assertFalse(buscar(vacia, "a"));
  Assert.assertEquals(-1, indice(vacia, "a"));
  Assert.assertError(() -> primero(vacia));
  Assert.assertError(() -> ultimo(vacia));
  Assert.assertError(() -> obtener(vacia, 0));
  Assert.assertError(() -> cambiar(vacia, 0, "x"));
  Assert.assertError(() -> borrar(vacia, 0));
  Assert.assertEq(null, buscarYborrar(vacia, "x"));

  NodoStr uno = insertarP(crearVacia(), "hola");
  Assert.assertFalse(esVacia(uno));
  Assert.assertEquals(1, longitud(uno));
  Assert.assertEquals("hola", primero(uno));
  Assert.assertEquals("hola", ultimo(uno));
  Assert.assertTrue(buscar(uno, "hola"));
  Assert.assertEquals(0, indice(uno, "hola"));
  Assert.assertEquals("hola", obtener(uno, 0));

  cambiar(uno, 0, "adios");
  Assert.assertEquals("adios", obtener(uno, 0));

  NodoStr borrado = borrar(uno, 0);
  Assert.assertTrue(esVacia(borrado));

  NodoStr borradoY = buscarYborrar(insertarP(null, "unico"), "unico");
  Assert.assertTrue(esVacia(borradoY));

  NodoStr o = insertarO(null, "x");
  Assert.assertEquals("x", obtener(o, 0));

  NodoStr ordenado = ordenar(insertarP(null, "y"));
  Assert.assertEquals("y", obtener(ordenado, 0));
}

private static void test7() {
  NodoStr l = crearVacia();
  for (int i = 0; i < 100; i++) {
    l = insertarU(l, "elem" + i);
  }
  Assert.assertEquals(100, longitud(l));
  Assert.assertEquals("elem0", obtener(l, 0));
  Assert.assertEquals("elem99", obtener(l, 99));

  l = insertar(l, 100, "fin");
  Assert.assertEquals("fin", obtener(l, 100));
```

```
    Assert.assertEquals(101, longitud(l));

    l = borrar(l, 100);
    Assert.assertEquals(100, longitud(l));
    Assert.assertEquals("elem99", obtener(l, 99));

    cambiar(l, 0, "inicio");
    cambiar(l, 99, "final");
    Assert.assertEquals("inicio", obtener(l, 0));
    Assert.assertEquals("final", obtener(l, 99));

    Assert.assertEquals(0, indice(l, "inicio"));
    Assert.assertEquals(99, indice(l, "final"));
  }

  private static void test8() {
    NodoStr l = crearVacia();
    Assert.assertError(() -> obtener(l, 1));
    Assert.assertError(() -> cambiar(l, 1, "Z"));
    Assert.assertError(() -> borrar(l, 1));
    l = insertarU(l,"A");
    Assert.assertError(() -> obtener(l, 2));
    Assert.assertError(() -> cambiar(l, 2, "Z"));
    Assert.assertError(() -> borrar(l, 2));
    l = insertarU(l,"C");
    Assert.assertError(() -> obtener(l, 3));
    Assert.assertError(() -> cambiar(l, 3, "Z"));
    Assert.assertError(() -> borrar(l, 3));
  }

  private static void test9() {
    NodoStr l = crearVacia();
    Assert.assertEquals("[]", formatear(l));
    l = insertarU(l, "A");
    Assert.assertEquals("[\"A\"]", formatear(l));
    l = insertarU(l, "B");
    Assert.assertEquals("[\"A\", \"B\"]", formatear(l));
    l = insertarU(l, "C");
    Assert.assertEquals("[\"A\", \"B\", \"C\"]", formatear(l));
  }
```

Anexo: biblioteca Assert

```
public class Assert {
  private Assert() { }

  public static void assertTrue(boolean c) {
    if (!c)
      throw new RuntimeException("\nERROR: la condición no es cierta");
  }
```

```
public static void assertFalse(boolean c) {
  if (c)
    throw new RuntimeException("\nERROR: la condición es cierta");
}

public static void assertEq(Object a, Object b) {
  if (a != b)
    throw new RuntimeException(String.format(
            "\nERROR: Los objetos 0x%08x y 0x%08x NO son el MISMO objeto",
            System.identityHashCode(a), System.identityHashCode(b)
          ));
}

public static void assertNotEq(Object a, Object b) {
  if (a == b)
    throw new RuntimeException(String.format(
            "\nERROR: Los objetos 0x%08x y 0x%08x SON el MISMO objeto",
            System.identityHashCode(a), System.identityHashCode(b)
          ));
}

public static void assertEquals(Object a, Object b) {
  if (!a.equals(b))
    throw new RuntimeException(String.format(
            "\nERROR: Los objetos \"%s\" (0x%08x) y \"%s\" (0x%08x) NO son IGUALES",
            a, System.identityHashCode(a), b, System.identityHashCode(b)
          ));
}

public static void assertNotEquals(Object a, Object b) {
  if (a.equals(b))
    throw new RuntimeException(String.format(
            "\nERROR: Los objetos \"%s\" (0x%08x) y \"%s\" (0x%08x) SON IGUALES",
            a, System.identityHashCode(a), b, System.identityHashCode(b)
          ));
}

public static void assertError(Code code) {
  Throwable error = null;
  try { code.run(); } catch (Exception e) { error = e; }
  if (error == null)
    throw new RuntimeException("\nERROR: Se esperaba un error que no se ha producido");
}

public static void assertNotError(Code code) {
  Throwable error = null;
  try { code.run(); } catch (Exception e) { error = e; }
  if (error != null)
    throw new RuntimeException("\nERROR: No se esperaba el error " + error);
}
```

```
  public static interface Code {void run() throws Exception;}
}
```