Universidad Europea del Atlántico

Asignatura:

Estructuras de datos y algoritmos I

Índice

Capítulo 3 •• Pilas

Capítulo 4 •• Colas (Queue)

Capítulo 5 •• Árboles

Capítulo 6 •• Grafos

Capítulo 7 •• Ficheros

Capítulo 8 •• Proyecto

Introducción de la asignatura

La asignatura estructura de datos y algoritmia I, tiene como objetivo introducir al alumno los conceptos fundamentales de ambos mundos dentro de la programación.

Las estructuras de datos son formas de organizar conjuntos de datos elementales, con el objetivo de facilitar su manipulación dentro de los lenguajes de programación. Existen estructuras de datos simples y otras mucho más complejas que permitirán dar solución a situaciones reales complejas que se dan en el mundo real y que los programadores tendrán que solucionar.

Los algoritmos en el mundo de la programación son instrucciones ordenadas, bien definidas que dan soluciones a determinados problemas. Dentro de los algoritmos se utilizan estructuras de datos.

La asignatura cubrirá la implementación de las estructuras de datos y algoritmos con el uso del lenguaje Java. Esto es debido a su fácil uso y extensión dentro del mundo empresarial. También se usará el UI Eclipse para facilitar las labores de programación así como el framework JUnit que nos permitirá probar la calidad del código fuente creado por el alumno.

Al finalizar la asignatura, se desarrollará un proyecto final que resolverá un problema real aplicando los conocimientos adquiridos durante el curso.

Capítulo 1

ESTRUCTURAS DE DATOS FUNDAMENTALES

Una computadora desarrolla diferentes tipos de tareas, pero en la mayoría de los casos utiliza datos para la ejecución de las mismas. Existen dos principales categorías para los datos:

- Datos primitivos.
- Datos estructurados.

Para clasificar los tipos de datos se utiliza el número de posiciones de memoria que utiliza cada dato. Es un dato primitivo si utiliza una única posición de memoria y compuesto si ocupa más de una posición de memoria. Los datos estructurados a su vez estarán compuestos por la unión de datos primitivos o estructurados.

1.1. DATOS PRIMITIVOS

Los **datos primitivos** más frecuentes en java y la mayoría de los lenguajes son:

Datos primitivos			
Byte	Entero	1 byte = 8 bits	-128 a 127
Short	Entero	2 byte	-32.768 a 32.767
Int	Entero	4 bytes	-2.147.483.648 .. 2.147.483.647

Datos primitivos			
Long	Entero	8 bytes	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807
Float	Decimal simple	4 bytes	3.4e-038 .. 3.4e+038
Double	Decimal simple	8 bytes	1.7e-308 .. 1.7e+308
Char	Carácter simple	2 bytes	0 .. 65536
Boolean	True or false	1 byte	True or false

Los datos primitivos se caracterizan porque en el momento de su utilización se reserva una posición de memoria con el tamaño del tipo de dato. Por ejemplo 1 byte si queremos usar un booleano. Esa posición de memoria queda reservada y será la misma aunque el valor de la variable booleana instanciada cambie de true a false y viceversa.

Ejemplo: int numero = 589 ;

1.2. DATOS ESTRUCTURADOS (VECTORES, REGISTROS, CONJUNTOS)

Los tipos de datos estructurados son aquellos que se componen, en un principio, a partir de los tipos de datos elementales o a su vez por otros tipos de datos estructurados. Los tipos de datos estructurados más comunes son:

- Vectores y matrices (arrays).
- Registros.
- Conjuntos.

Es importante conocer y tener en cuenta las diferencias entre las diferentes estructuras de datos. La elección se basará en las siguientes características:

- Si el objeto va a estar formado por elementos del mismo tipo de dato, entonces se podrá optar por el uso de un array o también llamado vector.
- Por el contrario si el objeto estará formado por varios tipos simples se tendrá que optar por el uso de un registro.
- Una vez decidido los tipos de datos se tendrá que ver si es necesario guardar algún tipo de órden sobre estos datos. Si se opta por mantener un cierto órden, entonces se usarán vectores o conjuntos si no existe órden alguno.

Vamos a detallar a continuación los diferentes tipos de datos estructurados y algunos ejemplos de los mismos.

1.2.1. VECTORES O MATRICES (ARRAY)

Un **vector** es un conjunto de elementos del mismo tipo y finito, con lo que tendrá un número limitado de elementos. Cuando decimos del mismo tipo, esto significa que el tipo puede ser o bien simple o bien estructurado. No importa el tipo, pero si que todos los elementos en el array tiene que ser del mismo. En caso de que intentemos asignar un elemento con un tipo diferente al del array el programa nos dará un error al compilar.

El array también estará ordenado. Esto nos permitirá acceder a cualquier elemento del array mediante el uso de su posición en el mismo. La posición no será nada más que un número que empieza en 0 y va has N-1. Donde N es el número de elementos del array.

En el siguiente ejemplo vamos a inicializar un array de 5 elementos de tipo primitivo int. Asignaremos un valor en una posición, leeremos el valor de otra posición y por último escribiremos por consola todos los valores del array.

1. Su representación gráfica e instanciación: `int[] arr = new int[5];`

0	1	2	3	4

2. Asignamos valor a la posición 1: `arr[1] = 5;`

	5			
0	1	2	3	4

3. Leemos el valor de la posición 4: `int x = arr[4]`

4) Para recorrer un array el método más utilizado es en bucle **for**, debido a que conocemos el número de elementos del vector. Ahora recorremos e imprimimos por pantalla los valores del array.

```
for (int i=0; i < arr.length; i++){
      System.out.println(i);
      }
```

Una **matriz** no es más que un vector o array multidimensional. Hasta este momento hemos trabajado con vectores lineales o de una dimensión. Para los vectores unidimensionales se utiliza un solo índice. Al hablar de matrices pasamos al mundo de los vectores multidimensionales donde podemos tener dos o más dimensiones. Por cada dimensión necesitaremos un nuevo índice.

A continuación vamos a ver un **vector bidimensional**. Podemos entender los vectores bidimensionales como tablas donde una dimensión representará las filas y la otra las columnas. En el siguiente ejemplo vamos a crear un vector bidimensional de 3x3.

```
int[][] arr = new int[3][3];    arr[0][0] -> Representa fila=0, columna=0
```

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]

Recorrido de una matriz de 3x3

```
for (int i=0; i < arr1.length; i++){
      for (int j=0; i < arr2.length; i++){

      }
}
```

1.2.2. REGISTROS (RECORD)

Los registros son estructuras de datos muy utilizadas. A diferencia de los vectores, los registros pueden almacenar datos de distintos tipos. Cada elemento que compone un registro se denomina campo. Cada campo puede ser de cualquier tipo y podemos tener diferentes campos del mismo tipo.

Las estructuras son un tipo de dato compuesto que no se definen como tal en java. Lo más parecido es el tipo struct muy usado en en c++. En java para la definición de un registro utilizaremos una clase (class) llamada javabean. Un javabean es una clase que encapsula diferentes objetos en uno sólo el bean. Son serializables, no tienen argumentos en el contructor y el acceso a sus propiedades se hace usando getter and setter.

Ejemplo donde creamos un registro o javabean para almacenar datos de una persona.

```
public class PersonBean implements java.io.Serializable {

    private String name = null;
    private boolean deceased = false;


    public PersonBean() {
    }

    /**
     * Getter for property <code>name</code>
     */
    public String getName() {
        return name;
    }

    /**
     * Setter for property <code>name</code>.
     * @param value
     */
    public void setName(final String value) {
        name = value;
    }

    /**
     * Getter for property "deceased"
     * Different syntax for a boolean field (is vs. get)
     */
    public boolean isDeceased() {
        return deceased;
    }

    /**
     * Setter for property <code>deceased</code>.
     * @param value
     */
    public void setDeceased(final boolean value) {
        deceased = value;
    }

}
```

1.2.3. CONJUNTOS

Los conjuntos son una reunión de datos que representan el mismo tipo pero carecen de orden. Por lo tanto la principal diferencia con el vector es que en este caso los conjuntos no tienen un orden definido. Que los elementos de un conjunto carezcan de orden no significa que no se puedan utilizar tipos de datos que presenten orden entre sus datos como los int.

Lo que realmente significa es que no va a ser posible tener acceso a los elementos porque carecen de orden. Los conjuntos tienen operaciones propias de los mismos:

- Unión (A+ B): Elementos del conjunto A más los del B no incluidos en el conjunto A.
- Intersección (A * B): Elementos comunes al conjunto A y B.
- Diferencia (A- B): Elementos del conjunto A que no están en el B.

		Nombre	Tipo	Ocupa	Rango aproximado
Tipos de datos en Java	**Tipos primitivos** (sin métodos; no son objetos; no necesitan una invocación para ser creados)	byte	Entero	1 byte	-128 a 127
		short	Entero	2 bytes	-32768 a 32767
		int	Entero	4 bytes	$2*10^9$
		long	Entero	8 bytes	Muy grande
		float	Decimal simple	4 bytes	Muy grande
		double	Decimal doble	8 bytes	Muy grande
		char	Carácter simple	2 bytes	---
		boolean	Valor true o false	1 byte	---
	Tipos objeto (con métodos, necesitan una invocación para ser creados)	Tipos de la biblioteca estándar de Java	String (cadenas de texto) Muchos otros (p.ej. Scanner, TreeSet, ArrayList…)		
		Tipos definidos por el programador / usuario	Cualquiera que se nos ocurra, por ejemplo Taxi, Autobus, Tranvia		
		arrays	Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.		
		Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos pero como objetos.)	Byte		
			Short		
			Integer		
			Long		
			Float		
			Double		
			Character		

1.3. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Realizar un programa que almacene en una variable la cadena, “Este es mi primer ejercicio”. Divida esa cadena de caracteres por letras y las inserte en un vector(array) de tipo char. Elimine del array la vocal “e”. Imprima el resultado como cadena concatenada. Se realizará un test unitario para probar el correcto funcionamiento.

```
package ejercicio1;


public class Cadena {
  /**
   * Esta función procesa una cadena y borra el caracter "e"
   * @param cadena
   * @return String
   */
  public static String ProcessCadena(String cadena){
        String[] listchar = cadena.split("(?!^)");
        StringBuilder builder = new StringBuilder();

        for(int i =0; i< listchar.length ;i++ ){
              if (!listchar[i].toString().toLowerCase().equals(new
String("e")))
                    builder.append(listchar[i].toString());
        }

        return builder.toString();
  }
}

package ejercicio1;

import org.junit.Assert;
import org.junit.Test;

public class Ejercicio1Test {

  @Test //Prueba si una pila esta vacia
  public void EmptyTest() {
        String output = Cadena.ProcessCadena("Este es mi primer ejer-
```

```
    cicio");
            Assert.assertEquals("st s mi primr jrcicio", output);

      }


}
```

2. Realizar un programa que tome como entrada un vector de números desordenados [5,1,9,7,6,3] y los ordene. La salida sería [1,3,5,6,7,9].Intentar resolver el ejercicio usando el algoritmo bubble short. Después utilizar clases del framework de Java y comparar tiempos de ejecución.

```
package ejercicio2;

import java.util.Arrays;

public class Vector{

      public static int[] BubbleSort(int[] numbers){

            for(int b=0; b<=numbers.length;b++){
            for(int c=0; c<=numbers.length-2;c++){
                if(numbers[c]>numbers[c+1]){

                    int temp=0;
                    temp=numbers[c];

                    numbers[c]=numbers[c+1];
                    numbers[c+1]=temp;
                }
            }

        }
            return numbers;
      }

      public static int[] JavaSort(int[] numbers){
            Arrays.sort(numbers);
            return numbers;
      }

}
```

```
package ejercicio2;

import java.util.Random;

import org.junit.Test;

public class Ejercicio2Test {

	@Test
	//Genera numeros aleatorios en un array y los ordena
	public void BubbleSortTest() {
		int[] numArray = this.CreateArrayInt(1000);
		Vector.BubbleSort(numArray);
	}

	@Test
	//Genera numeros aleatorios en un array y los ordena
	public void ArraySortTest() {
		int[] numArray = this.CreateArrayInt(1000);
		Vector.JavaSort(numArray);
	}


	private int[] CreateArrayInt(int number){
		Random r = new Random();
		int count = 1000;

		int[] array = new int[count];
		for (int i=0; i < count ; i++){
			array[i] = r.nextInt();
		}
		return array;
	}

}
```

3. Realizar un programa que cree dos matrices de 3x3 de tipo int y devuelva tanto la suma como la multiplicación. Se realizarán dos test unitarios para comprobar el resultado.

```java
package ejercicio3;

public class Matrix {
	/**
	 * Suma 2 matrizes
	 * @param m1
	 * @param m2
	 * @return
	 */
	public static int[][] SumMatrix(int[][] m1, int[][] m2){
		int size = m1.length;
		int[][] sum = new int[size][size];
		for (int i = 0 ; i < size ; i++ )
		    for (int j = 0 ; j < size ; j++ )
		        sum[i][j] = m1[i][j] + m2[i][j];
		return sum;
	}

	/**
	 * Multiplica 2 matrizes
	 * @param m1
	 * @param m2
	 * @return
	 */
	public static int[][] MultiMatrix(int[][] m1, int[][] m2){
		int size = m1.length;
		int[][] sum = new int[size][size];
		for (int i = 0 ; i < size ; i++ )
		    for (int j = 0 ; j < size ; j++ )
		        sum[i][j] = m1[i][j] * m2[i][j];
		return sum;
	}
}
```

```java
package ejercicio3;

import java.util.Random;

import org.junit.Assert;
```

```
import org.junit.Test;

import ejercicio2.Vector;

public class Ejercicio3Test {

	@Test
	//Suma dos matrizes
	public void SumMatrixTest() {
		int[][] numArray = this.CreateArrayInt();
		int[][] sumArray = Matrix.SumMatrix(numArray, numArray);

		Assert.assertEquals(sumArray[0][0],2);
		Assert.assertEquals(sumArray[0][1],2);
		Assert.assertEquals(sumArray[0][2],2);
		Assert.assertEquals(sumArray[1][0],2);
		Assert.assertEquals(sumArray[1][1],2);
		Assert.assertEquals(sumArray[1][2],2);
		Assert.assertEquals(sumArray[2][0],2);
		Assert.assertEquals(sumArray[2][1],2);
		Assert.assertEquals(sumArray[2][2],2);
	}

	@Test
	//Multiplica 2 matrizes
	public void MulMatrixTest() {
		int[][] numArray = this.CreateArrayInt();
		int[][] mulArray = Matrix.MultiMatrix(numArray, numArray);

		Assert.assertEquals(mulArray[0][0],1);
		Assert.assertEquals(mulArray[0][1],1);
		Assert.assertEquals(mulArray[0][2],1);
		Assert.assertEquals(mulArray[1][0],1);
		Assert.assertEquals(mulArray[1][1],1);
		Assert.assertEquals(mulArray[1][2],1);
		Assert.assertEquals(mulArray[2][0],1);
		Assert.assertEquals(mulArray[2][1],1);
		Assert.assertEquals(mulArray[2][2],1);
	}

	private int[][] CreateArrayInt(){
```

```
        int[][] array = new int[3][3];
        array[0][0] = 1;
        array[0][1] = 1;
        array[0][2] = 1;
        array[1][0] = 1;
        array[1][1] = 1;
        array[1][2] = 1;
        array[2][0] = 1;
        array[2][1] = 1;
        array[2][2] = 1;
        return array;
    }

}
```

4. Definir una estructura de datos que permita almacenar la información correspondiente al pasaporte de una persona. Crear dos métodos adicionales. Uno que devuelva el nombre GetFullName y otro que devuelva la edad como número GetAge. Probar su correcto funcionamiento con test unitarios.

```
package ejercicio4;

import java.sql.Date;
import java.util.Calendar;

/**
 * Clase Personbean que encapsula todos los datos representativos de una persona.
 * @author SMaroto
 */
public class PersonBean  implements java.io.Serializable {

      private int number;
      private String name;
      private String surname;
      private Calendar birthDate;
      private String country;

      // Constructor sin argumentos
      public PersonBean() {
      }
```

```java
/**
 * Constructor del JavaBean con argumentos
 * @param name
 * @param birthDate
*/
public PersonBean(int number,String name, Calendar birthDate) {
      this.setNumber(number);
      this.setName(name);
      this.setBirthDate(birthDate);
}

/**
 * Constructor por copia
 * @param personBean
 */
public PersonBean(PersonBean personBean) {
      this.number = personBean.getNumber();
      this.name = personBean.getName();
      this.surname = personBean.getSurname();
      this.birthDate = personBean.getBirthDate();
      this.country = personBean.getCountry();
}

public int getNumber() {
      return number;
}

public void setNumber(int number) {
      this.number = number;
}

public String getName() {
      return name;
}

public void setName(String name) {
      this.name = name;
}

public Calendar getBirthDate() {
      return birthDate;
}

public void setBirthDate(Calendar birthDate) {
      this.birthDate = birthDate;
}
```

```
	public String getSurname() {
		return surname;
	}

	public void setSurname(String surname) {
		this.surname = surname;
	}

	public String getCountry() {
		return country;
	}

	public void setCountry(String country) {
		this.country = country;
	}

	public String GetFullName(){
		return this.name +  " " + this.surname;
	}

	public int getAge(){
		Calendar today = Calendar.getInstance();
	    int years = today.get(Calendar.YEAR) - this.birthDate.get(Cal-
endar.YEAR);

	    if (((today.get(Calendar.MONTH) + 1)  < (this.birth-
Date.get(Calendar.MONTH) +1)) &&
	            (today.get(Calendar.DAY_OF_MONTH) < this.birth-
Date.get(Calendar.DAY_OF_MONTH)))
	        years -= 1;

	        return years;


	}
}


package ejercicio4;

import java.util.Calendar;

import org.junit.Assert;
import org.junit.Test;
```

```
public class Ejercicio4Test {

	@Test
	//Crea un objecto person y valida sus datos
	public void CreatePersonBean() {
		Calendar c = Calendar.getInstance();
		c.set(1982, 5, 20);
		PersonBean person = new PersonBean(1256,"Peter Smith",c);

		Assert.assertEquals(person.getName(), "Peter Smith");
		Assert.assertEquals(person.getNumber(), 1256);
	}

	@Test
	//Crea un objecto person y valida su fecha de nacimento
	public void ValidatePersonBeanAge() {
		Calendar c = Calendar.getInstance();
		c.set(1982, 5, 20);
		PersonBean person = new PersonBean(1256,"Peter Smith",c);

		Assert.assertEquals(person.getNumber(), 1256);
		Assert.assertEquals(person.getAge(), 33);

	}

}
```

Capítulo 2

LISTAS ENLAZADAS

2.1. DEFINICIÓN Y DISEÑO

Dentro de las estructuras de datos, una de las más utilizadas es la de la estructura de datos tipo lista. Será utilizada como soporte para la implementación de otras muchas estructuras de datos que se verán más adelante.

La principal ventaja frente a otros tipos de listas como los vectores es su flexibilidad. También la forma de almacenar la información en memoria o disco, permitiendo que el orden de almacenamiento sea distinto al del recorrido.

Las listas enlazadas son de una enorme flexibilidad ya que pueden aumentar o disminuir su tamaño a preferencia del programador.

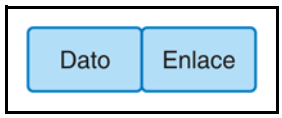
El acceso a cualquier elemento de la lista enlazada es otra de sus particularidades. Se podrá añadir o eliminar un elemento en cualquier posición dentro de la lista enlazada.

Cada elemento dentro de la lista enlazada se llama nodo y un nodo está compuesto por dos partes. La información que contiene el nodo y la posición al siguiente nodo o elemento de la lista.

Representación gráfica de un nodo.

- Dato: Información que contendrá el nodo.
- Enlace: Posición al siguiente nodo dentro de la lista.

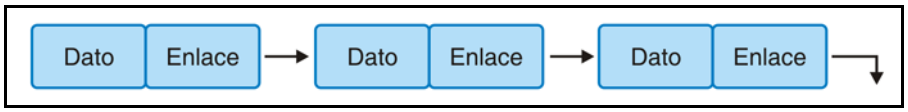
Nodo



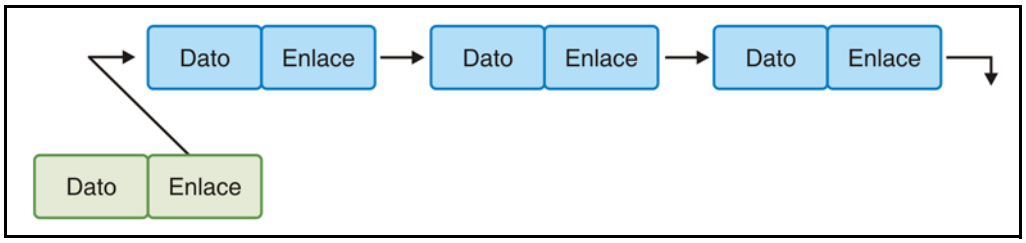
Existen dos nodos dentro de las listas enlazadas que son diferentes a los demás. El primer y el último nodo. El primer nodo no tendrá ningún otro nodo por detrás y el último nodo no tendrá ningún nodo por delante, su enlace apuntará a null.

2.1.1. LISTA ENLAZADA SIMPLE

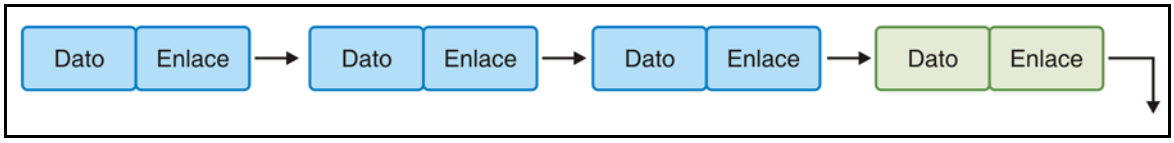
Lista enlazada



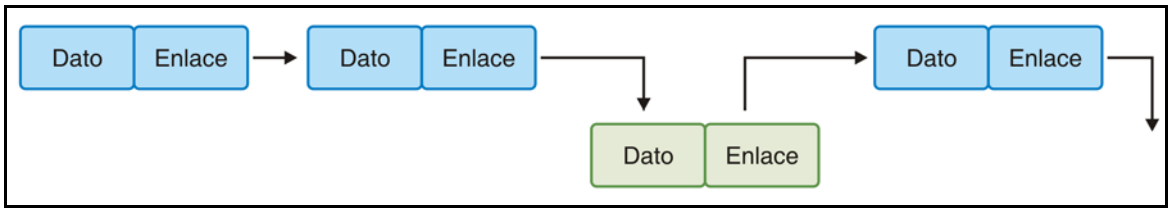
Lista enlazada - insertar inicio



Lista enlazada - insertar fin



Lista enlazada - insertar en el medio



Existen diferentes operaciones con listas enlazadas. Enumeramos las principales a continuación:

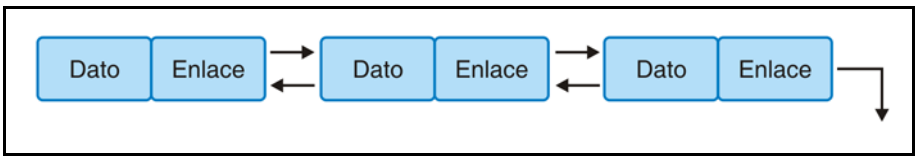
- Crear -> Constructor
- Capacidad - > Size
- EstaVacia -> Empty
- InsertarInicio -> InserIni
- InsertarFinal -> InsertEnd
- EliminarInicio -> DeleteIni
- ExtraerInicio -> GetFirst
- ListarTodo -> ListAll

2.1.2. LISTA DOBLEMENTE ENLAZADA

Dentro de las listas enlazadas existe otra variante que son las listas doblemente enlazadas. En estas cada nodo tiene un enlace al nodo anterior y posterior.

Representación gráfica de una lista doblemente enlazada.

Lista doblemente enlazada



2.2. IMPLEMENTACIÓN

Clases a implementar:

Clase Nodo

```
package data.uneatlantico.es;

public class Nodo {

	private String dato;
	private Nodo siguiente;
```

```
	public Nodo(String i){
		this.setDato(i);
	}

	public String getDato() {
		return dato;
	}

	public void setDato(String dato) {
		this.dato = dato;
	}

	public Nodo getSiguiente() {
		return siguiente;
	}

	public void setSiguiente(Nodo siguiente) {
		this.siguiente = siguiente;
	}

}
```

Clase Lista

```
package data.uneatlantico.es;

public class List {
	private Nodo first = null;

	public List(){
	}

	/**
	 * Devuelve el tamaño de la lista
	 * @return int
	 */
	public int Size(){
		if (this.first == null)
```

```
            return 0;

        int count = 1;
        Nodo iterator = first;
        while (iterator.getSiguiente() !=null){
            count++;
            iterator = iterator.getSiguiente();
        }
        return count;
    }

    /**
     * Devuelve true si esta vacia y false si tiene datos
     * @return boolean
     */
    public boolean IsEmpty(){
        return this.Size() > 0 ? false : true;
    }

    /**
     * Inserta un nodo al inicio de la lista
     */
    public void InsertIni(String dato){
        Nodo n = new Nodo(dato);
        if (this.first == null)
            this.first = n;
        else{
            n.setSiguiente(this.first);
            this.first = n;
        }
    }

    /**
     * Inserta un nodo al final de la lista
     */
    public void InsertEnd(String dato){
        Nodo n = new Nodo(dato);
        if (this.first == null)
            this.first = n;
        else{
            Nodo iterator = this.first;
```

```
            while (iterator.getSiguiente() != null){
                  iterator = iterator.getSiguiente();
            }
            iterator.setSiguiente(n);
      }
}

/**
 * Borra un nodo al principio de la lista
 */
public void DeleteIni(){
      if (this.first !=null){
            this.first = this.first.getSiguiente();
      }
}

/**
 * Obtiene el primer nodo
 */
public Nodo GetFirst(){
      return this.first;
}


/**
 * Devuelve un array con los datos de la lista.
 * @return String[]
 */
public String[] ListAll(){
      String[] list = new String[this.Size()];
      Nodo iterator = this.first;

      if (iterator == null)
            return list;

      int count =0;
      while (iterator != null){
            list[count] = iterator.getDato();
            count++;
            iterator = iterator.getSiguiente();
      }
```

```
            return list;
      }

}
```

Clase de prueba:

```
package testdata.uneatlantico.es;

import org.junit.Assert;
import org.junit.Test;
import data.uneatlantico.es.*;


public class ListTest {

      @Test //Prueba si una lista esta vacia
      public void EmptyTest() {
            List l = new List();
            Assert.assertTrue("Test failed for no empty",l.IsEmpty());
      }

      @Test //Prueba si una lista no esta vacia
      public void NoEmptyTest() {
            List l = new List();
            l.InsertIni("1");
            Assert.assertFalse("Test failed for no empty
list",l.IsEmpty());
      }

      @Test //Prueba el correcto tamano de una lista
      public void SizeTest() {
            List l = new List();
            Assert.assertEquals(l.Size(),0);

            l.InsertIni("1");
            l.InsertEnd("2");
            Assert.assertEquals(l.Size(),2);
      }
```

```
	@Test //El primer dato es vacio
	public void GetFirstEmptyTest() {
		List l = new List();
		Assert.assertNull(l.GetFirst());
	}

	@Test //Obtine el primer dato de la lista
	public void GetFirstTest() {
		List l = new List();
		l.InsertIni("1");
		l.InsertIni("2");
		Assert.assertEquals(l.GetFirst().getDato(),"2");
	}

	@Test //Lista los datos correctamente
	public void ListAllTest() {
		List l = new List();
		l.InsertIni("1");
		l.InsertEnd("2");
		l.InsertEnd("5");
		l.InsertIni("3");

		Assert.assertEquals(l.Size(),4);
		String[] larray = l.ListAll();
		Assert.assertEquals(larray[0],"3");
		Assert.assertEquals(larray[1],"1");
		Assert.assertEquals(larray[2],"2");
		Assert.assertEquals(larray[3],"5");
	}


}
```

Ahora vamos a crear una clase Nodo genérica y una clase lista genérica. Esto nos permite crear una única clase que pueda almacenar cualquier tipo de dato. Eso sí, todos los elementos de la lista tienen que ser del mismo tipo.

Clase NodoG:

```
package common.data.uneatlantico.es;

public class NodoG<T> {

	private T dato;
	private NodoG<T> siguiente;


	public NodoG(T i){
		this.setDato(i);
	}

	public T getDato() {
		return dato;
	}

	public void setDato(T dato) {
		this.dato = dato;
	}

	public NodoG<T> getSiguiente() {
		return siguiente;
	}

	public void setSiguiente(NodoG<T> siguiente) {
		this.siguiente = siguiente;
	}


}
```

Clase ListaG:

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public class ListG<T> {
	private NodoG<T> first = null;
```

```
public ListG(){
}

/**
 * Devuelve el tamaño de la lista
 * @return int
 */
public int Size(){
      if (this.first == null)
            return 0;

      int count = 1;
      NodoG<T> iterator = first;
      while (iterator.getSiguiente() !=null){
            count++;
            iterator = iterator.getSiguiente();
      }
      return count;
}

/**
 * Devuelve true si esta vacia y false si tiene datos
 * @return boolean
 */
public boolean IsEmpty(){
      return this.Size() > 0 ? false : true;
}

/**
 * Inserta un nodo al inicio de la lista
 */
public void InsertIni(T dato){
       NodoG<T> n = new NodoG<T>(dato);
       if (this.first == null)
             this.first = n;
       else{
             n.setSiguiente(this.first);
             this.first = n;
       }
}
```

```
/**
 * Inserta un nodo al final de la lista
 */
public void InsertEnd(T dato){
	NodoG<T> n = new NodoG<T>(dato);
	if (this.first == null)
		this.first = n;
	else{
		NodoG<T> iterator = this.first;
		while (iterator.getSiguiente() != null){
			iterator = iterator.getSiguiente();
		}
		iterator.setSiguiente(n);
	}
}

/**
 * Borra un nodo al principio de la lista
 */
public void DeleteIni(){
	if (this.first !=null){
		this.first = this.first.getSiguiente();
	}
}

/**
 * Obtiene el primer nodo
 */
public NodoG<T> GetFirst(){
	return this.first;
}


/**
 * Devuelve un array con los datos de la lista.
 * @return String[]
 */
public T[] ListAll(){
	T[] list = (T[])new Object[this.Size()];
	NodoG<T> iterator = this.first;
```

```
            if (iterator == null)
                  return list;

            int count =0;
            while (iterator != null){
                 list[count] = iterator.getDato();
                 count++;
                 iterator = iterator.getSiguiente();
            }

            return list;
      }

}
```

Clase ListGTest

```
package testdata.uneatlantico.es;

import org.junit.Assert;
import org.junit.Test;
import data.uneatlantico.es.*;


public class ListGTest {

      @Test //Prueba si una lista esta vacia
      public void EmptyTest() {
            ListG<Integer> l = new ListG<Integer>();
            Assert.assertTrue("Test failed for no empty",l.IsEmpty());
      }

      @Test //Prueba si una lista no esta vacia
      public void NoEmptyTest() {
            ListG<Integer> l = new ListG<Integer>();
            l.InsertIni(1);
            Assert.assertFalse("Test failed for no empty
list",l.IsEmpty());
      }

      @Test //Prueba el correcto tamano de una lista
```

```
	public void SizeTest() {
		ListG<Integer> l = new ListG<Integer>();
		Assert.assertEquals(l.Size(),0);

		l.InsertIni(1);
		l.InsertEnd(2);
		Assert.assertEquals(l.Size(),2);
	}

	@Test //El primer dato es vacio
	public void GetFirstEmptyTest() {
		ListG<Integer> l = new ListG<Integer>();
		Assert.assertNull(l.GetFirst());
	}

	@Test //Obtine el primer dato de la lista
	public void GetFirstTest() {
		ListG<Integer> l = new ListG<Integer>();
		l.InsertIni(1);
		l.InsertIni(2);
		Assert.assertEquals(l.GetFirst().getDato(),new Integer(2));
	}

	@Test //Lista los datos correctamente
	public void ListAllTest() {
		ListG<Integer> l = new ListG<Integer>();
		l.InsertIni(1);
		l.InsertEnd(2);
		l.InsertEnd(5);
		l.InsertIni(3);

		Assert.assertEquals(l.Size(),4);
		Object[] larray = l.ListAll();
		Assert.assertEquals((Integer)larray[0],new Integer(3));
		Assert.assertEquals((Integer)larray[1],new Integer(1));
		Assert.assertEquals((Integer)larray[2],new Integer(2));
		Assert.assertEquals((Integer)larray[3],new Integer(5));
	}

}
```

Ahora vamos a ver como la mismas clases cambian al utilizar iterfaces y clases abstractas. Esto es fundamental para desarrollar adecuadamente código que siga patrones adecuados.

Interface IDataStructure

```
package data.uneatlantico.es;


/**
 * Interface para estructuras de datos
 * @author SMaroto
 * @param <T>
 */
public interface IDataStructure<T> {
	public boolean IsEmpty();
	public int Size();
	public  T[] ListAll();


}
```

Clase abstracta DataStructure

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public abstract class DataStructure<T> implements IDataStructure<T> {
	public NodoG<T> first =null;

	@Override
	public abstract boolean IsEmpty();

	@Override
	public abstract int Size();

	@Override
	public abstract T[] ListAll();


}
```

Clase ListG heredando de clase abstracta

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public class ListG<T>  extends DataStructure<T>{
	private NodoG<T> first = null;

	public ListG(){
	}

	/**
	 * Devuelve el tamaño de la lista
	 * @return int
	 */
	public int Size(){
		if (this.first == null)
			return 0;

		int count = 1;
		NodoG<T> iterator = first;
		while (iterator.getSiguiente() !=null){
			count++;
			iterator = iterator.getSiguiente();
		}
		return count;
	}

	/**
	 * Devuelve true si esta vacia y false si tiene datos
	 * @return boolean
	 */
	public boolean IsEmpty(){
		return this.Size() > 0 ? false : true;
	}

	/**
	 * Inserta un nodo al inicio de la lista
	 */
	public void InsertIni(T dato){
		NodoG<T> n = new NodoG<T>(dato);
		if (this.first == null)
```

```
                this.first = n;
          else{
                n.setSiguiente(this.first);
                this.first = n;
          }
    }

    /**
     * Inserta un nodo al final de la lista
     */
    public void InsertEnd(T dato){
          NodoG<T> n = new NodoG<T>(dato);
          if (this.first == null)
                this.first = n;
          else{
               NodoG<T> iterator = this.first;
               while (iterator.getSiguiente() != null){
                     iterator = iterator.getSiguiente();
               }
               iterator.setSiguiente(n);
          }
    }

    /**
     * Borra un nodo al principio de la lista
     */
    public void DeleteIni(){
         if (this.first !=null){
               this.first = this.first.getSiguiente();
         }
    }

    /**
     * Obtiene el primer nodo
     */
    public NodoG<T> GetFirst(){
         return this.first;
    }

    /**
     * Devuelve un array con los datos de la lista.
     * @return T[]
     */
```

```
	@SuppressWarnings("unchecked")
	public T[] ListAll(){
		T[] list = (T[])new Object[this.Size()];
		NodoG<T> iterator = this.first;

		if (iterator == null)
			return list;

		int count =0;
		while (iterator != null){
			list[count] = iterator.getDato();
			count++;
			iterator = iterator.getSiguiente();
		}

		return list;
	}


}
```


2.3. CASO PRÁCTICO

Definir una estructura de datos de tipo lista enlazada que permita almacenar los deportistas inscritos en una prueba de maratón.

Cada deportista constará de la siguiente información:

- Nombre, Nacionalidad, Fecha de nacimiento, fecha inicio y fecha fin.
- Los deportistas estarán almacenados estáticamente en la lista. Lo cual significa que serán introducidos por el alumno en el código. A esto se le llama hardcoded.

Una vez creada la lista enlazada y almacenados los deportistas seleccionados por el alumno, se presentarán dos opciones por teclado:

1. Listar todos los deportistas inscritos en la prueba.
2. Listar la información de todos los deportistas que coincidan con un nombre dado por teclado
3. Terminar el programa.

El código deberá ser testeado usando JUnit framework.

Clase Person

```
package casopractico;

import java.util.Calendar;


/**
 * Clase Personbean que encapsula todos los datos representativos de una persona.
 * @author SMaroto
 */
public class PersonBean  implements java.io.Serializable {


      private int number;
       private String name;
       private Calendar birthDate;
       private String country;
       private Calendar startTime;
       private Calendar endTime;

       // Constructor sin argumentos
       public PersonBean() {
       }

      /**
       * Constructor del JavaBean con argumentos
       * @param name
       * @param birthDate
      */
      public PersonBean(int number,String name, Calendar birthDate) {
            this.setNumber(number);
            this.setName(name);
            this.setBirthDate(birthDate);
      }

      /**
       * Constructor por copia
       * @param personBean
       */
```

```
public PersonBean(PersonBean personBean) {
	this.number = personBean.getNumber();
	this.name = personBean.getName();
	this.birthDate = personBean.getBirthDate();
	this.country = personBean.getCountry();
}

public int getNumber() {
	return number;
}

public void setNumber(int number) {
	this.number = number;
}

public String getName() {
	return name;
}

public void setName(String name) {
	this.name = name;
}

public Calendar getBirthDate() {
	return birthDate;
}

public void setBirthDate(Calendar birthDate) {
	this.birthDate = birthDate;
}

public String getCountry() {
	return country;
}

public void setCountry(String country) {
	this.country = country;
}

public Calendar getEndTime() {
	return endTime;
```

```
	}

	public void setEndTime(Calendar endTime) {
		this.endTime = endTime;
	}

	public Calendar getStartTime() {
		return startTime;
	}

	public void setStartTime(Calendar startTime) {
		this.startTime = startTime;
	}



}
```

Clase Lista

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public class ListG<T>  extends DataStructure<T>{
	private NodoG<T> first = null;

	public ListG(){
	}

	/**
	 * Devuelve el tamaño de la lista
	 * @return int
	 */
	public int Size(){
		if (this.first == null)
			return 0;

		int count = 1;
		NodoG<T> iterator = first;
		while (iterator.getNext() !=null){
```

```
            count++;
            iterator = iterator.getNext();
      }
      return count;
}

/**
 * Devuelve true si esta vacia y false si tiene datos
 * @return boolean
 */
public boolean IsEmpty(){
      return this.Size() > 0 ? false : true;
}

/**
 * Inserta un nodo al inicio de la lista
 */
public void InsertIni(T data){
       NodoG<T> n = new NodoG<T>(data);
       n.setNext(this.first);
       this.first = n;
}

/**
 * Inserta un nodo al final de la lista
 */
public void InsertEnd(T data){
       NodoG<T> n = new NodoG<T>(data);
       if (this.first == null)
             this.first = n;
       else{
            NodoG<T> iterator = this.first;
            while (iterator.getNext() != null){
                  iterator = iterator.getNext();
            }
            iterator.setNext(n);
       }
}

/**
 * Borra un nodo al principio de la lista
 */
```

```
	public void DeleteIni(){
		if (this.first !=null){
			this.first = this.first.getNext();
		}
	}

	/**
	 * Obtiene el primer nodo
	 */
	public NodoG<T> GetFirst(){
		return this.first;
	}


	/**
	 * Devuelve un array con los datos de la lista.
	 * @return T[]
	 */
	@SuppressWarnings("unchecked")
	public T[] ListAll(){
		T[] list = (T[])new Object[this.Size()];
		NodoG<T> iterator = this.first;

		if (iterator == null)
			return list;

		int count =0;
		while (iterator != null){
			list[count] = iterator.getData();
			count++;
			iterator = iterator.getNext();
		}

		return list;
	}


}
```

Programa Principal

```
package casopractico;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Calendar;

import data.uneatlantico.es.ListG;

public class Program {

	private static ListG<PersonBean> list = new ListG<>();

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		PopulatePeople();

		String option = "";
		while (option.equals("") || !option.equals("3")){
			try {
				PrintOptions();
				BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
				option = br.readLine();
				switch(option){
				    case "1" :
				      System.out.println("");
					System.out.println("Listado :");
					System.out.println("-----------------
----------");

					for (Object person :
Program.list.ListAll()){
						PersonBean p =
(PersonBean)person;
	System.out.println(p.getNumber() + " , " + p.getName());
					}
				    break;
				case "2" :
```

```
                        System.out.println("Introducir nombre :");
                        String name = br.readLine();
                        for (Object person :
Program.list.ListAll()){
                                    PersonBean p =
(PersonBean)person;
                                    if
(p.getName().toLowerCase().contains(name))
      System.out.println(p.getNumber() + " , " + p.getName());
                                }
                          break;
                        case "3" :
                          System.out.println("Programa terminado");
                           break;
                        default :
                          System.out.println("Opción no reconocida
:");
                  }
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }

      }

      private static void PrintOptions(){
            System.out.println("----------------------------");
            System.out.println("Opciones de programa:");
            System.out.println("1: Listar todos los deportistas
inscritos");
            System.out.println("2: Listar deportistas por Identificador
introducido");
            System.out.println("3: Terminar");
            System.out.println("----------------------------");
      }


      /**
       * Este metodo inserta algunos deportistas hardcoded
       */
```

```
    private static void PopulatePeople(){
          Calendar c = Calendar.getInstance();
          c.set(1982, 5, 20);
          PersonBean person1 = new PersonBean(1,"Juan Smith",c);
          c.set(1950, 10, 25);
          PersonBean person2 = new PersonBean(2,"John Garcia",c);
          c.set(1990, 8, 10);
          PersonBean person3 = new PersonBean(3,"Pedro Smith",c);

          Program.list.InsertIni(person1);
          Program.list.InsertIni(person2);
          Program.list.InsertIni(person3);

    }


}
```


2.4. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Desarrollar el ejercicio anterior utilizando la estructura de tipo lista proporcionada por el framework de Java. La clase se encuentra en el paquete **java.util.List**.. Ejemplo **List<Deportista>**.
2. Utilizar el framework JUnit para hacer las siguientes pruebas unitarias.
   a. ImprimirElementosListaVaciaTest.java.
      i. Probar que cuando la lista esta vacía devuelve 0 elementos.
   b. ImprimirElementosInsertarDeportistaPrincipioTest.java
      i. Probar que cuando se añade un deportista al inicio este se encuentra en la lista al inicio.
   c. ImprimirElementosInsertarDeportistaFinalTest.java
      i. Probar que cuando se añade un deportista al final este se encuentra en la lista al final.
   d. ImprimirBorrrarDeportistaFinalTest.java
      i. Probar que cuando se borra un deportista del final de la lista este no se encuentre en la lista.

Capítulo 3

PILAS

3.1. DEFINICIÓN Y DISEÑO

La estructura de datos de tipo pila (stack en inglés) es similar a una lista enlazada. Lo que la diferencia es que en una pila existen restricciones en cuanto a la posición en la cual se puede realizar la inserción o eliminación de elementos.

A las pilas también se las llaman estructuras de datos LIFO (last in first out). Esto lo que indica es que el último elemento añadido siempre ser el primero que se recupere.

Las pilas tienen dos operaciones fundamentales: apilar (push) que inserta un elemento al principio y retirar (pop) que extrae el último elemento apilado.

Por lo tanto en una pila en cada momento sólo se tiene acceso al último elemento apilado en la parte superior de la pila. Este elemento se llama TOS (top of stack). Una vez retirado ese elemento el siguiente elemento pasar a ser el nuevo TOS.

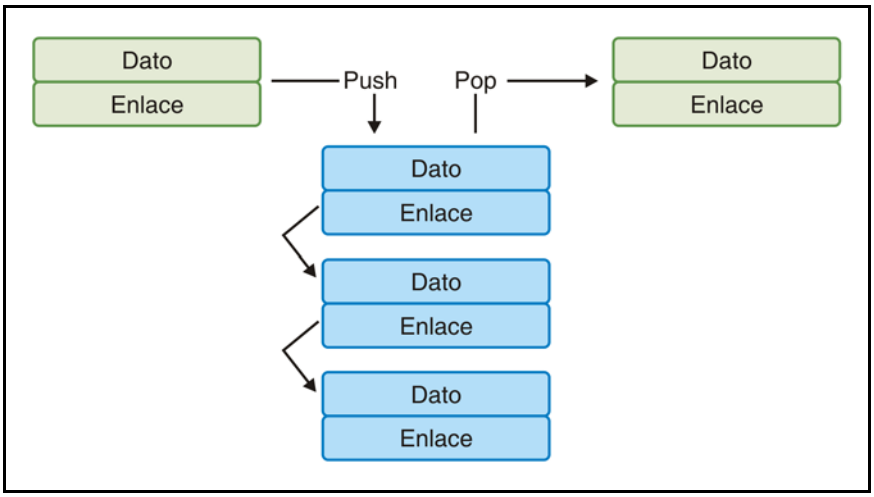
Uso de las pilas:

a) Un ejemplo para el uso de pilas es la gestión de ventanas de Windows, cuando cerramos una ventana siempre recuperamos la que teníamos detrás.

b) Otro ejemplo es en el análisis sintáctico no dependiente del contexto. En este caso tenemos que analizar las palabras que componen una sentencia en orden de aparición.

c) Llamas a subprogramas. Cuando un programa principal invoca a un subprograma, es necesario mantener el valor y estado de las variables que se retornarán al programa principal.

Pila - Stack



Existen diferentes operaciones con pilas. Enumeramos las principales a continuación:

- Crear (contructor).
- Capacidad (Size).
- EstaVacia (Empty).
- Apilar (push).
- Desapilar (pop).
- Cima (Top).

3.2. IMPLEMENTACIÓN

Clases a implementar:

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public class Stack<T> extends DataStructure<T> {
	NodoG<T> first = null;
```

```
/**
 * Apila un nuevo elemento en la parte superior de la pila
 */
public void  Push(T data) {
	NodoG<T> n = new NodoG<T>(data);
	n.setNext(this.first);
	this.first =n;
}

/**
 * Recupera el elemento superior de la pila
 * @return T
 */
public T Pop() {
	if (this.first == null)
		return null;
	T data = this.first.getData();
	this.first = this.first.getNext();

	return data;
}

/**
 * Devuelve el dato superior de la pila sin cambiar nada
 * @return
 */
public T Top() {
	if (this.first == null)
		return null;
	return this.first.getData();
}

@Override
public boolean IsEmpty() {
	return this.Size() > 0 ? false : true;
}

@Override
public int Size() {
	if (this.first == null)
		return 0;
```

```
        int count = 1;
        NodoG<T> iterator = first;
        while (iterator.getNext() !=null){
              count++;
              iterator = iterator.getNext();
        }
        return count;
  }

  @SuppressWarnings("unchecked")
  @Override
  public T[] ListAll() {
        T[] list = (T[])new Object[this.Size()];
         NodoG<T> iterator = this.first;

         if (iterator == null)
               return list;

         int count =0;
         while (iterator != null){
              list[count] = iterator.getData();
              count++;
              iterator = iterator.getNext();
         }

         return list;
  }


}
```

Clase test

```
package testdata.uneatlantico.es;

import org.junit.Assert;
import org.junit.Test;

import data.uneatlantico.es.Stack;
```

```
public class StackTest {

	@Test //Prueba si una pila esta vacia
	public void EmptyTest() {
		Stack<Integer> s = new Stack<Integer>();
		Assert.assertTrue("Test failed for no empty",s.IsEmpty());
	}

	@Test //Prueba si una pila no esta vacia
	public void NoEmptyTest() {
		Stack<Integer> s = new Stack<Integer>();
		s.Push(1);
		Assert.assertFalse("Test failed for no empty
list",s.IsEmpty());
	}

	@Test //Prueba el correcto tamano de una lista
	public void SizeTest() {
		Stack<Integer> s = new Stack<Integer>();
		Assert.assertEquals(s.Size(),0);

		s.Push(1);
		s.Push(2);
		Assert.assertEquals(s.Size(),2);
	}

	@Test //El primer dato es vacio
	public void GetFirstEmptyTest() {
		Stack<Integer> s = new Stack<Integer>();
		Assert.assertNull(s.Top());
	}

	@Test //Obtine el primer dato de la lista
	public void GetFirstTest() {
		Stack<Integer> s = new Stack<Integer>();
		s.Push(1);
		s.Push(2);
		Assert.assertEquals(s.Top(),new Integer(2));
	}
```

```
	@Test //Lista los datos correctamente
	public void ListAllTest() {
		Stack<Integer> s = new Stack<Integer>();
		s.Push(1);
		s.Push(2);
		s.Push(5);
		s.Push(2);
		s.Push(0);

		Assert.assertEquals(s.Size(),5);
		Object[] larray = s.ListAll();
		Assert.assertEquals((Integer)larray[0],new Integer(0));
		Assert.assertEquals((Integer)larray[1],new Integer(2));
		Assert.assertEquals((Integer)larray[2],new Integer(5));
		Assert.assertEquals((Integer)larray[3],new Integer(2));
		Assert.assertEquals((Integer)larray[4],new Integer(1));
	}

}
```

3.3. CASO PRÁCTICO

Una de las utilizaciones más comunes del uso de las pilas, es la evaluación de las expresiones matemáticas. En este capítulo vamos a desarrollar un programa capaz de interpretar si una expresión matemática está correctamente formada en términos de equilibrado de paréntesis, llaves y corchetes.

a. Ejemplo: “(Equilibrado (1))” es correcto.

b. Ejemplo: “(Equilibrado 2))” es incorrecto.

c. Ejemplo: “((Equilibrado 3))” es correcto.

```
package equilibrado;

import data.uneatlantico.es.Stack;

public class CadenaEquilibrada {

	private String cadena= "";

	public CadenaEquilibrada(String cadena){
```

```
            this.cadena = cadena;
      }

      public boolean EsCadeEquilibrada(){
            Stack<Character> locStack = new Stack<Character>();

            for(char c : this.cadena.toCharArray()){
                  if ( c == '(')
                        locStack.Push(new Character(c));
                  else if (c == ')'){
                        if (!locStack.IsEmpty())
                              locStack.Pop();
                        else
                              return false;
                  }
            }

            return locStack.Size() > 0 ? false : true;
      }
}
```



```
package equilibrado;

import org.junit.Assert;
import org.junit.Test;


public class CadenaEquilibradaTest {

      @Test
      //Suma dos matrizes
      public void TestCadenaEQuilibrada() {

            CadenaEquilibrada equi = new CadenaEquilibrada("(Equilibrado
(1))");
            Assert.assertTrue(equi.EsCadeEquilibrada());
```

```
        equi = new CadenaEquilibrada("(Equilibrado 2))");
        Assert.assertFalse(equi.EsCadeEquilibrada());

        equi = new CadenaEquilibrada("((Equilibrado 3))");
        Assert.assertTrue(equi.EsCadeEquilibrada());

    }

}
```

3.4. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Desarrollar un programa que lea un número por teclado y que lo escriba por pantalla en orden inverso. Para ello se utilizar una estructura de datos tipo pila. Se desarrollaron los correspondientes test unitarios.

   Ejemplo: entrada 12345 y salida 54312

2. Se va a desarrollar un programa que utilice la clase Stack del framework de Java como estructura de datos. El programa hará lo siguiente:

   a. Se introducirán una lista de aviones comerciales que sobrevuelan el espacio aéreo de Santander. Cada avión tiene los siguientes campos

      i. Identificador numérico.

      ii. Compaña.

      iii. Origen.

      iv. Destino.

      v. Fecha y hora salida.

      vi. Fecha y hora llegada.

   b. Se presentar un listado con todos los aviones que tengan como destino Madrid.

Capítulo 4

COLAS (QUEUE)

4.1. DEFINICIÓN Y DISEÑO

Al igual que la estructura de datos de tipo pila, la cola es un conjunto de elementos apilados unos encima de otros. Para poder entender la estructura de datos de tipo cola se puede recurrir a la imagen que todos tenemos de la cola de un concierto. En esta cola todos están esperando para comprar su entrada. Toda persona que quiera comprar una entrada se tendrá que incorporar por el final de la misma. Toda persona espera hasta que se encuentre al principio de la cola y pueda adquirir su entrada.

A las colas también se las llaman estructuras de datos FIFO (first in first out). Esto lo que indica es que el primer elemento añadido siempre será el primero que se recupere.

Al igual que las pilas son estructuras de datos que tienen restricciones en cuanto a la posición por la cual se puede realizar la extracción e inserción de elementos. La inserción o push se hará por el final y la extracción o pop por el principio.

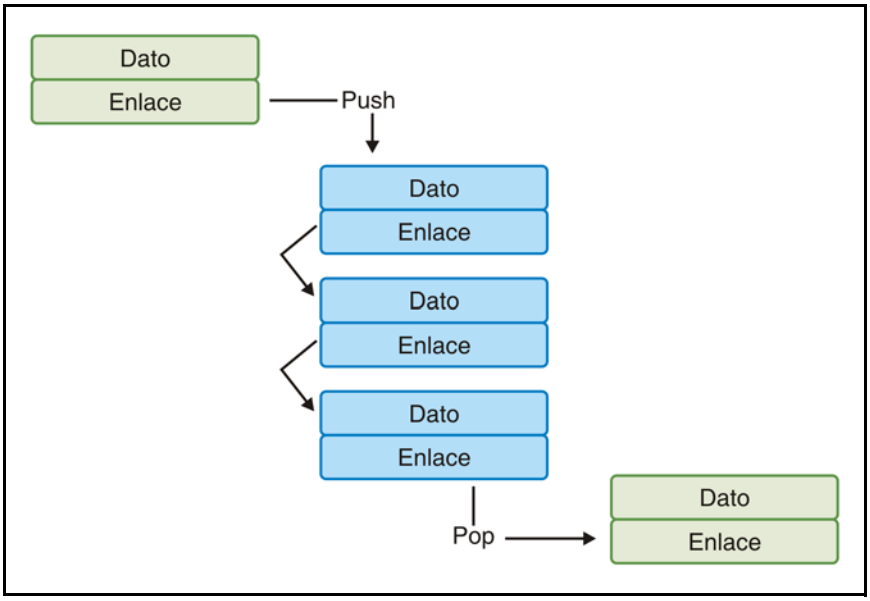
Uso de las colas:

a) Muy utilizadas en procesos informáticos donde se quiere dar total prioridad por orden de llegada. Redes de transporte como aviones esperando slot para aterrizar o despegar.

b) Sistemas de mensajería o bus de servicio dónde se quieren entregar mensajes a determinados destinatarios y se quiere asegurar que los mensajes se entregan en orden. Por ejemplo en una cuenta bancaria es importante ejecutar

las operaciones en orden. De esta manera nos aseguramos no tener una cuenta en números rojos y todavía extraer dinero de la misma.

Cola - Queue



Existen diferentes operaciones con pilas. Enumeramos las principales a continuación:

- Crear (contructor).
- Capacidad (Size).
- EstaVacia (Empty).
- Apilar (push).
- Desapilar (pop).

4.2. IMPLEMENTACIÓN

Clases a implementar:

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoG;

public class Queue<T> extends DataStructure<T> {
	NodoG<T> first = null;
```

```
	/**
	 * Apila un nuevo elemento en la parte superior de la cola (FIFO)
	 */
	public void  Push(T data) {
		NodoG<T> n = new NodoG<T>(data);
		n.setNext(this.first);
		this.first =n;
	}

	/**
	 * Recupera el elemento de la cola Basado en FIFO (first in first
out)
	 * @return T
	 */
	public T Pop() {
		NodoG<T> iterator = this.first;
		NodoG<T> iteratorAnt = iterator;

		if (iterator == null)
			return null;

		while (iterator.getNext() != null){//Stop at the last Node
			iteratorAnt = iterator;
			iterator = iterator.getNext();
		}

		if (this.Size() == 1 )
			this.first =null;
		else
			iteratorAnt.setNext(null);

		return iterator.getData();
	}

	@Override
	public boolean IsEmpty() {
		return this.Size() > 0 ? false : true;
	}

	@Override
```

```
	public int Size() {
		if (this.first == null)
			return 0;

		int count = 1;
		NodoG<T> iterator = first;
		while (iterator.getNext() !=null){
			count++;
			iterator = iterator.getNext();
		}
		return count;
	}

	@SuppressWarnings("unchecked")
	@Override
	public T[] ListAll() {
		T[] list = (T[])new Object[this.Size()];
		NodoG<T> iterator = this.first;

		if (iterator == null)
			return list;

		int count =0;
		while (iterator != null){
			list[count] = iterator.getData();
			count++;
			iterator = iterator.getNext();
		}

		return list;
	}

}
```

```
package testdata.uneatlantico.es;

import org.junit.Assert;
import org.junit.Test;

import data.uneatlantico.es.Queue;
```

```
public class QueueTest {

	@Test //Prueba si una cola esta vacia
	public void EmptyTest() {
		Queue<Integer> q = new Queue<Integer>();
		Assert.assertTrue("Test failed for no empty",q.IsEmpty());
	}

	@Test //Prueba si una cola no esta vacia
	public void NoEmptyTest() {
		Queue<Integer> q = new Queue<Integer>();
		q.Push(1);
		Assert.assertFalse("Test failed for no empty
list",q.IsEmpty());
	}

	@Test //Prueba el correcto tamano de una cola
	public void SizeTest() {
		Queue<Integer> q = new Queue<Integer>();
		Assert.assertEquals(q.Size(),0);
		q.Push(1);
		q.Push(2);
		q.Push(3);
		Assert.assertEquals(q.Size(),3);
	}

	@Test //Prueba si apilando y desapilando tiene el tañano correcto
	public void PushAndPopTestSizeEmpty() {
		Queue<Integer> q = new Queue<Integer>();
		q.Push(1);
		q.Pop();
		q.Push(5);
		q.Push(8);
		q.Pop();
		q.Pop();
		Assert.assertEquals(q.Size(),0);
	}

	@Test //Prueba si apilando y desapilando tiene el tañano correcto
	public void PushAndPopTestSizeNoEmpty() {
```

```
        Queue<Integer> q = new Queue<Integer>();
        q.Push(1);
        q.Pop();
        q.Push(5);
        q.Push(8);
        q.Pop();
        q.Pop();
        q.Push(8);
        q.Push(8);
        Assert.assertEquals(q.Size(),2);
    }

    @Test //Prueba si apilando y desapilando tiene el tañano correcto
    //y devuelve los valores correctos
    public void PushAndPopExpectedValues() {
        Queue<Integer> q = new Queue<Integer>();
        q.Push(1);
        Assert.assertEquals(q.Pop(),new Integer(1));
        q.Push(5);
        q.Push(8);
        Assert.assertEquals(q.Pop(),new Integer(5));
        Assert.assertEquals(q.Pop(),new Integer(8));
        q.Push(4);
        q.Push(8);
        Assert.assertEquals(q.Pop(),new Integer(4));
        Assert.assertEquals(q.Pop(),new Integer(8));
        Assert.assertEquals(q.Size(),0);
    }

}
```

4.3. CASO PRÁCTICO

Se va a proceder al desarrollo de un sistema para el proceso de transacciones bancarias. Cada transacción va a consistir de una cuenta origen, cuenta destino e importe. Cada cuenta está asociada a una persona.

El importe será con signo positivo en caso de ingreso en la cuenta destino y retirada en la cuenta de origen.

Para asegurarnos de que no se produzcan descubiertos indeseados, las operaciones se desarrollarán en estricto orden de entrada al sistema. Para este procesamiento utilizaremos una estructura de datos de tipo cola.

En programa almacenará una serie de órdenes introducidas hardcoded. Al final del procesamiento se tendrá que sacar un listado de todas las operaciones ejecutadas y el balance de la cuentas. En caso de que una cuenta se quede en números rojos, se cancelarán las operaciones sobre la misma haciéndose un rollback que deshaga la operación e imprima el problema por pantalla.

Cliente

```
import java.util.Calendar;

import bean.uneatlantico.es.PersonBean;

public class Cliente extends PersonBean {

	public Cliente(int number,String name, Calendar birthDate) {
		super(number,name,birthDate);
	}

}
```

Cuenta bancaria

```
/**
 * Clase que repesenta a una cuenta de banco
 * @author SMaroto
 *
 */
public class CuentaBancaria {
	private int numCuenta;
	private Cliente cliente;
	private Double saldo;

	public CuentaBancaria(int numCuenta,Cliente cliente,Double saldo){
		this.numCuenta = numCuenta;
		this.cliente = cliente;
		this.saldo = saldo;
	}
	public Cliente getCliente() {
```

```
		return cliente;
	}

	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

	public Double getSaldo() {
		return saldo;
	}

	public void setSaldo(Double saldo) {
		this.saldo = saldo;
	}

	public int getNumCuenta() {
		return numCuenta;
	}

	public void setNumCuenta(int numCuenta) {
		this.numCuenta = numCuenta;
	}

	public Double ActualizarSaldo(Double importe){
		if (this.saldo + importe < 0)
			return -1.00;
		this.saldo += importe;
		return this.saldo;
	}


}
```

Transacción

```
public class Transaccion  {

	private CuentaBancaria cuentaOrigen;
	private CuentaBancaria cuentaDestino;
	private Double importe;
```

```
	private String concepto;

	public Transaccion(CuentaBancaria origen,CuentaBancaria destino,
Double importe,String concepto) {
		this.cuentaOrigen = origen;
		this.cuentaDestino= destino;
		this.importe = importe;
		this.setConcepto(concepto);
	}

	public CuentaBancaria getCuentaOrigen() {
		return cuentaOrigen;
	}
	public void setCuentaOrigen(CuentaBancaria cuentaOrigen) {
		this.cuentaOrigen = cuentaOrigen;
	}

	public CuentaBancaria getCuentaDestino() {
		return cuentaDestino;
	}
	public void setCuentaDestino(CuentaBancaria cuentaDestino) {
		this.cuentaDestino = cuentaDestino;
	}

	public Double getImporte() {
		return importe;
	}

	public void setImporte(Double importe) {
		this.importe = importe;
	}

	public String getConcepto() {
		return concepto;
	}

	public void setConcepto(String concepto) {
		this.concepto = concepto;
	}


}
```

```
import data.uneatlantico.es.ListG;
import data.uneatlantico.es.Queue;

public class Program {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Queue<Transaccion> transacciones = new Queue<Transaccion>();
		ListG<CuentaBancaria> cuentas = new ListG<CuentaBancaria>();
		CargarTransacciones(transacciones,cuentas);
		EjecutarListarTransacciones(transacciones);
		ListarSaldos(cuentas);
	}

	//Este método carga de forma hardcoded las transacciones
	public static void CargarTransacciones(Queue<Transaccion>
transacciones,ListG<CuentaBancaria> cuentas){
		//Crear 3 clientes
		Cliente cli1 = new Cliente(1,"Juan Silva", null);
		Cliente cli2 = new Cliente(2,"Pedro Ronal",null);
		Cliente cli3 = new Cliente(3,"Juan Ruta",null);

		//Crear cuenta bancaria a los clientes
		CuentaBancaria cuentaClie1 = new CuentaBancaria(34569876,
cli1, 1000.0);
		cuentas.InsertEnd(cuentaClie1);
		CuentaBancaria cuentaClie2 = new CuentaBancaria(10500010,
cli2, 15500.0);
		cuentas.InsertEnd(cuentaClie2);
		CuentaBancaria cuentaClie3 = new CuentaBancaria(25658900,
cli3, -100.0);
		cuentas.InsertEnd(cuentaClie3);

		Transaccion t1 = new Transac-
cion(cuentaClie1,cuentaClie2,500.00,"Renta Enero");
		transacciones.Push(t1);
		Transaccion t2 = new Transac-
cion(cuentaClie1,cuentaClie2,300.00,"Gastos Enero");
		transacciones.Push(t2);
		Transaccion t3 = new Transac-
cion(cuentaClie2,cuentaClie3,1500.00,"Gastos extras");
		transacciones.Push(t3);
		Transaccion t4 = new Transac-
```

```
cion(cuentaClie2,cuentaClie1,500.00,"Gastos extras");
            transacciones.Push(t4);
            Transaccion t5 = new Transac-
cion(cuentaClie2,cuentaClie3,1500.00, "Pago");
            transacciones.Push(t5);

      }

      public static void EjecutarListarTransacciones(Queue<Transaccion>
transacciones){
            for(Object t : transacciones.ListAll()){
                  Transaccion transaccion = (Transaccion)t;
                  System.out.println("Transaccion : cc/origen " +
transaccion.getCuentaOrigen().getNumCuenta() +
                                          " a cc/destino " +
transaccion.getCuentaDestino().getNumCuenta() +
                                          " por importe : " +
transaccion.getImporte());

                  transaccion.getCuentaOrigen().ActualizarSaldo(-
transaccion.getImporte());
                  transaccion.getCuentaDestino().ActualizarSaldo(transac-
cion.getImporte());

            }

      }

      public static void ListarSaldos(ListG<CuentaBancaria> cuentas){
            for(Object t : cuentas.ListAll()){
                  CuentaBancaria cuenta = (CuentaBancaria)t;
                  System.out.println("Cuenta : cc " + cuenta.getNum-
Cuenta() +
                              " cliente " + cuenta.getCliente().getName() +
                               " saldo importe : " + cuenta.getSaldo());

            }

      }



}
```

4.4. EJERCICIOS DE EVALUACIÓN CONTINUA

1. En un aeropuerto existen 2 pistas de aterrizaje que van a ser utilizadas alternativamente por los aviones que están en espera de despegue. Cada avión constará de número de vuelo, compañía, destino del mismo y segundos que tarda en despegar. Crear una estructura de datos que almacene 10 aviones y que asigne por estricto orden de llegada de los aviones la pista desde la que despegará cada avión. Imprimir por pantalla para cada pista los aviones asignados a la misma.
2. Defina una estructura de datos que permita dar forma a las llamadas telefónicas que se realicen al departamento de atención al cliente de una operadora.
3. Basándonos en la estructura de datos anterior, realizar un sistema que introduciendo el número de operador asigne automáticamente una de las llamadas en espera. El sistema empieza con 5 llamadas en espera. Existirá la opción de imprimir las llamadas gestionadas por un determinado operador.

Capítulo 5

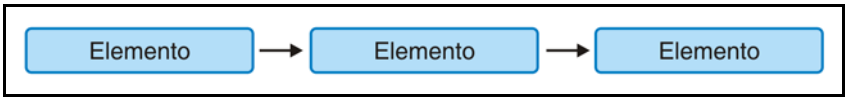
ÁRBOLES

5.1. DEFINICIÓN Y DISEÑO

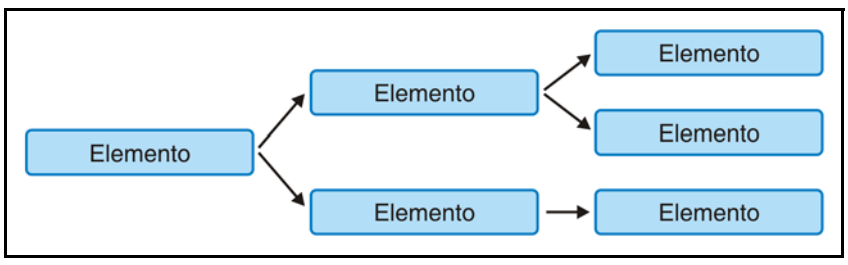
Hasta el momento hemos estado trabajando con estructuras de datos que se denominan lineales. Cada elemento está seguido por un único elemento. En el caso de los árboles cambiamos a estructuras no lineales. A cada elemento de la estructura no tiene porque seguirle un único elemento. Cada elemento puede estar seguido por varios elementos.

Por lo tanto un árbol es una estructura de datos no lineal que almacena información jerárquicamente.

Ejemplo Estructura lineal:



Ejemplo Estructura no lineal:



La estructura de tipo árbol se suele utilizar para representar estructuras jerárquicas del mundo real. Normalmente los elementos del árbol que están en la parte superior suelen tener mayor importancia que los que están en los niveles más bajos. El ejemplo más sencillo y representativo es un árbol genealógico.

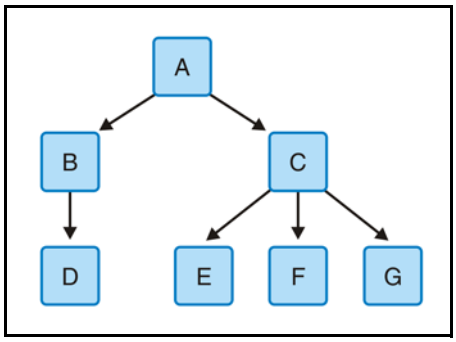
Para que se cumpla una estructura de tipo árbol, se tienen que cumplir:

- Existe un único nodo raíz. De este salen todo el resto de elementos del árbol.
- Todos los elementos hijos del nodo raíz serán elementos disjuntos. Esto decir sin elementos comunes. Cada hijo a su vez va a ser otra estructura de tipo árbol.

Características de un árbol:

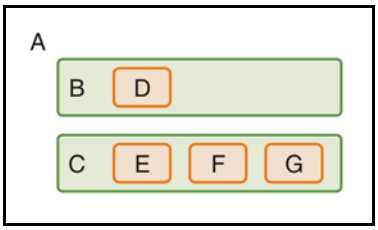
- Desde el nodo raíz se puede llegar a cualquier nodo del árbol progresando por las ramas y atravesando diferentes nodos. Esto irá conformando un **camino**.
- La relación entre dos nodos separados de forma inmediata por una rama se denomina **padre-hijo**. En un árbol un padre puede tener varios hijos pero un hijo sólo puede tener un padre.
- Se denomina **grado** al número de subárboles del nodo.
- Se establece que un nodo es **hoja** si no tiene descendientes.
- El **peso** de un árbol es el número de hojas o nodos terminales de un árbol.
- **Nivel de nodo** es el número de nodos que le anteceden en el árbol. Por ejemplo el nivel de nodo raíz es 1.
- **Profundidad de un árbol** es el mayor de los niveles de todos los nodos del árbol.

En el siguiente ejemplo vamos a analizar un árbol con sus correspondientes características y su posible representación.



1. Raíz: A
2. Subárboles de A: subárbol B,D y subárbol C,E,F,G
3. Hijos de A: B y C
4. Padre D: B
5. Grado A: 2, Grado B: 1, Grado C: 3, grado G:0, Grado D:0
6. Nivel A:1, Nivel B:2, Nivel D:3

Representación mediante conjuntos:



Representación mediante listas

(A (B (D)) (C (E) (F) (G)))

5.1.1. ÁRBOLES BINARIOS

Un árbol binario es aquel en el que ninguno de sus nodos puede tener más de dos subárboles. Por lo tanto cada nodo puede tener cero, un hijo o como máximo dos hijos, de aquí el nombre de binario. De esta manera se observa que sólo existirá un camino entre dos nodos. Los árboles binarios son muy utilizados como árboles binarios de búsqueda. BST del acrónimo Binary Search Tree.

Para crear un árbol binario de búsqueda se tiene que cumplir:

- En caso de tener subárbol izquierdo, la raíz R debe ser mayor que el valor máximo almacenado en el subárbol izquierdo, y que el subárbol izquierdo sea un árbol binario de búsqueda.
- En caso de tener subárbol derecho, la raíz R debe ser menor que el valor mínimo almacenado en el subárbol derecho, y que el subárbol derecho sea un árbol binario de búsqueda.

Para una fácil comprensión queda resumido en que es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores

menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

La fama de los árboles binarios se debe a que su recorrido en inorden visibiliza los elementos ordenados de forma ascendente y por lo tanto la búsqueda es muy eficiente.

Existen diferentes operaciones con árboles binarios:

- Crear (contructor).
- Capacidad (Size).
- EstaVacia (Empty).
- Buscar (Find).
- Insertar (Insert).
- Eliminar (Delete).

Preorden: (raíz, izquierdo, derecho). Para recorrer un árbol binario no vacío en preorden, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo de raíz:

1. Visite la raíz.
2. Atraviese el sub-árbol izquierdo.
3. Atraviese el sub-árbol derecho.

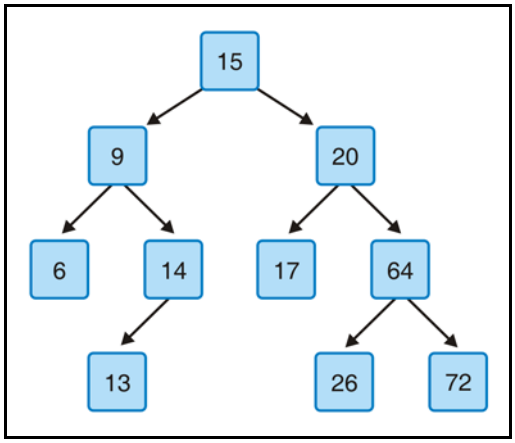
Inorden: (izquierdo, raíz, derecho). Para recorrer un árbol binario no vacío en inorden (simétrico), hay que realizar las siguientes operaciones recursivamente en cada nodo:

1. Atraviese el sub-árbol izquierdo.
2. Visite la raíz.
3. Atraviese el sub-árbol derecho.

Postorden: (izquierdo, derecho, raíz). Para recorrer un árbol binario no vacío en postorden, hay que realizar las siguientes operaciones recursivamente en cada nodo:

1. Atraviese el sub-árbol izquierdo.
2. Atraviese el sub-árbol derecho.
3. Visite la raíz.

Ejemplo recorridos árbol binario:



Preorden = [15, 9, 6, 14, 13, 20, 17, 64, 26, 72]

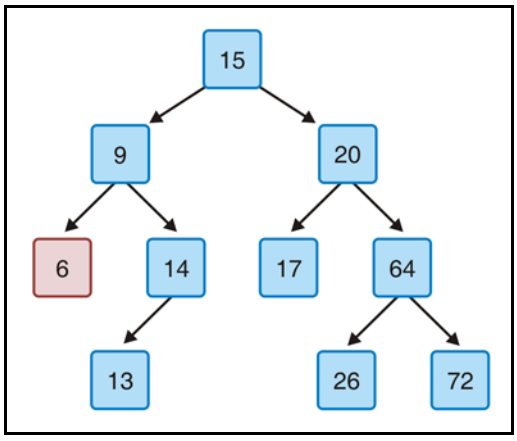
Inorden = [6, 9, 13, 14, 15, 17, 20, 26, 64, 72]

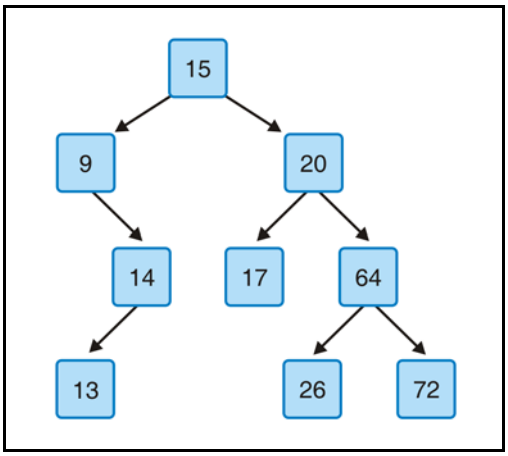
Postorden = [6, 13, 14, 9, 17, 26, 72, 64, 20, 15]

5.1.2. BORRAR NODOS EN UN ÁRBOL BINARIO

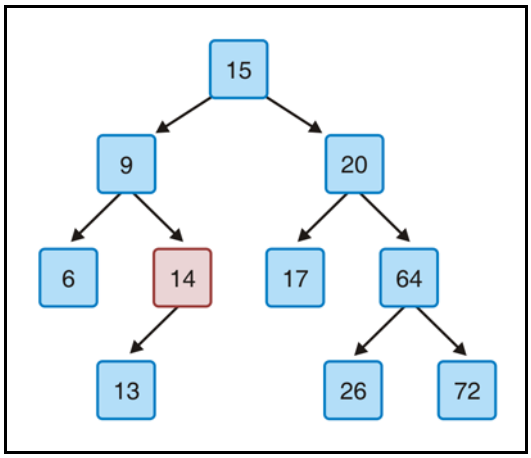
Borrar un nodo en un árbol binario de búsqueda no es una de las tareas más sencillas y se pueden presentar las siguientes situaciones.

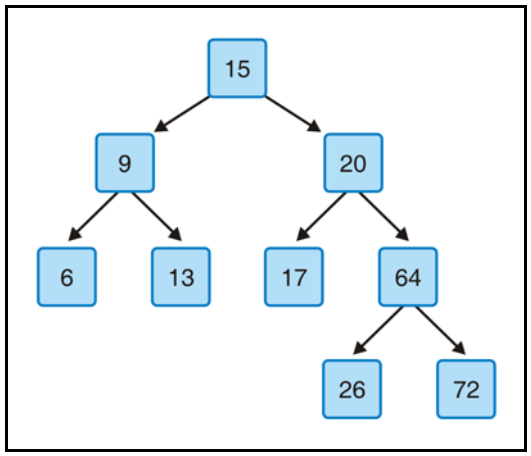
- Borrar un nodo sin hijos:
  - Lo único que hay que hacer es borrar el nodo y establecer el enlace de su padre a nulo. Por ejemplo si eliminamos el nodo 6 de nuestro árbol quedaría así.





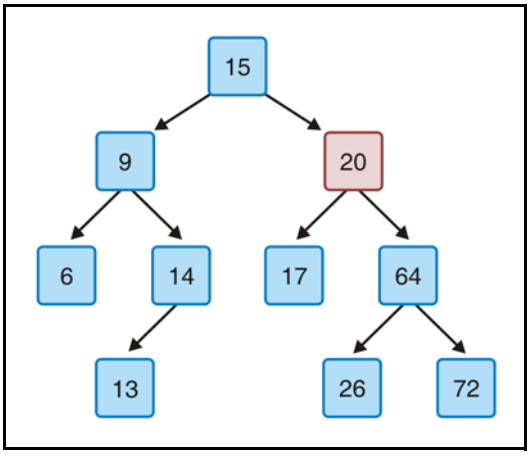
- Borrar un nodo con un subárbol hijo:
  - En este caso únicamente tenemos que borrar el nodo y el subárbol que tenía pasa a ocupar su lugar.

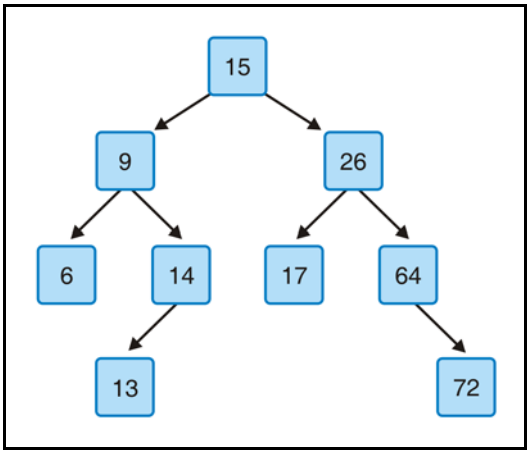




- Borrar un Nodo con dos subárboles hijos:

– Este es un caso algo complejo, tenemos que tomar el hijo derecho del Nodo que queremos eliminar y recorrer hasta el hijo más a la izquierda (hijo izquierdo y si este tiene hijo izquierdo repetir hasta llegar al último nodo a la izquierda), reemplazamos el valor del nodo que queremos eliminar por el nodo que encontramos (el hijo más a la izquierda), el nodo que encontramos por ser el más a la izquierda es imposible que tenga nodos a su izquierda pero si que es posible que tenga un subárbol a la derecha, para terminar solo nos queda proceder a eliminar este nodo de las formas que conocemos (caso 1, caso 2) y tendremos la eliminación completa.





5.2. IMPLEMENTACIÓN ÁRBOL BINARIO

Clases a implementar:

```
package common.data.uneatlantico.es;

public class NodoTree<T> {
```

```
	private T dato;
	private NodoTree<T> izquierdo;
	private NodoTree<T> derecho;

	public NodoTree(T dato){
		this.dato = dato;
		this.izquierdo =null;
		this.derecho =null;
	}


	public T getDato() {
		return dato;
	}

	public void setDato(T dato) {
		this.dato = dato;
	}

	public NodoTree<T> getIzquierdo() {
		return izquierdo;
	}

	public void setIzquierdo(NodoTree<T> izquierdo) {
		this.izquierdo = izquierdo;
	}

	public NodoTree<T> getDerecho() {
		return derecho;
	}

	public void setDerecho(NodoTree<T> derecho) {
		this.derecho = derecho;
	}


}

package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoTree;
```

```
public class BinaryTree<T> {

	private NodoTree<T> raiz;


	public BinaryTree(){
	}

	public NodoTree<T> getRaiz() {
		return raiz;
	}

	public void setRaiz(NodoTree<T> raiz) {
		this.raiz = raiz;
	}

	public void Insert(T dato){
		NodoTree<T> nodoNuevo = new NodoTree<T>(dato);

		if (this.raiz == null)
			this.raiz = nodoNuevo;
		else{
			NodoTree<T> nodoAux = raiz;
			NodoTree<T>  padre;
			while (true){
				padre = nodoAux;
				if (new Integer((int) dato) < new Integer((int)
nodoAux.getDato())){
					nodoAux = nodoAux.getIzquierdo();
					if(nodoAux == null){
						padre.setIzquierdo(nodoNuevo);
						return;//Salir del metodo
					}
				}else{
					nodoAux = nodoAux.getDerecho();
					if (nodoAux == null){
						padre.setDerecho(nodoNuevo);
						return;//Salir de metodo
					}
				}
```

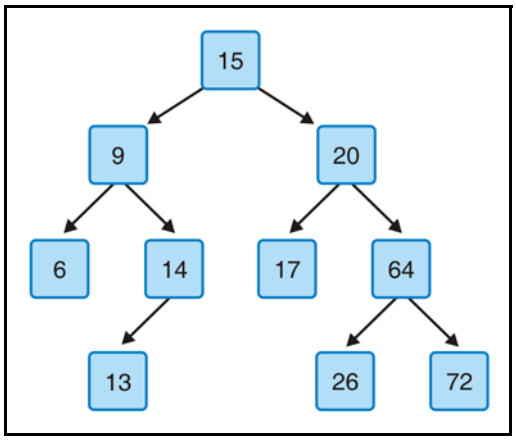
```
            }
        }
    }

    public void InOrden(){
        System.out.println("In Orden");
        this.InOrden(this.raiz);
    }

    //Izquierda raiz derecha
    public void PreOrden(){
        System.out.println("Pre Orden");
        this.PreOrden(this.raiz);
    }

    //Izquierda derecha raiz
    public void PostOrden(){
        System.out.println("Post Orden");
        this.PostOrden(this.raiz);
    }

    private void InOrden(NodoTree<T> raiz){
        if (raiz != null){
            InOrden(raiz.getIzquierdo());
            System.out.println(raiz.getDato());
            InOrden(raiz.getDerecho());
        }
    }

    private void PreOrden(NodoTree<T> raiz){
        if (raiz != null){
            System.out.println(raiz.getDato());
            PreOrden(raiz.getIzquierdo());
            PreOrden(raiz.getDerecho());
        }
    }


    private void PostOrden(NodoTree<T> raiz){
        if (raiz != null){
```

```
                PostOrden(raiz.getIzquierdo());
                PostOrden(raiz.getDerecho());
                System.out.println(raiz.getDato());
            }
        }




}

package testdata.uneatlantico.es;

import org.junit.Assert;
import org.junit.Test;

import data.uneatlantico.es.BinaryTree;


public class BinaryTreeTest {

    @Test //Prueba si un arbol  esta vacia
    public void EmptyTest() {
        BinaryTree<Integer> tree = new BinaryTree<Integer>();
        Assert.assertTrue("Test failed for no empty",tree.getRaiz()
== null);
    }

    @Test //Recorre arbol en Preorden
    public void PreOrdenTest() {
        BinaryTree<Integer> tree = new BinaryTree<Integer>();
        tree.Insert(15);
        tree.Insert(9);
        tree.Insert(6);
        tree.Insert(14);
        tree.Insert(13);
        tree.Insert(20);
        tree.Insert(17);
        tree.Insert(64);
        tree.Insert(26);
```

```
		tree.Insert(72);
		tree.PreOrden();
	}

	@Test //Recorre arbol en InOrden
	public void InOrdenTest() {
		BinaryTree<Integer> tree = new BinaryTree<Integer>();
		tree.Insert(15);
		tree.Insert(9);
		tree.Insert(6);
		tree.Insert(14);
		tree.Insert(13);
		tree.Insert(20);
		tree.Insert(17);
		tree.Insert(64);
		tree.Insert(26);
		tree.Insert(72);
		tree.InOrden();
	}

	@Test //Recorre arbol en PostOrden
	public void PostOrdenTest() {
		BinaryTree<Integer> tree = new BinaryTree<Integer>();
		tree.Insert(15);
		tree.Insert(9);
		tree.Insert(6);
		tree.Insert(14);
		tree.Insert(13);
		tree.Insert(20);
		tree.Insert(17);
		tree.Insert(64);
		tree.Insert(26);
		tree.Insert(72);
		tree.PostOrden();
	}

}
```

5.3. CASO PRÁCTICO

Desarrollar un árbol binario de búsqueda y una vez implementada la funcionalidad realizar las siguientes operaciones comprobando que funcione correctamente con test unitarios.



1. Crear el árbol con los elementos de la imagen en la misma forma.
2. Buscar en preorden el nodo 14 y presentar su camino.
3. Comparar la búsqueda en preorden e inorden y comparar el tiempo empleado para encontrar todos los nodos hoja del árbol. 13,26,72.

Binary Tree extendido

```
package data.uneatlantico.es;

import common.data.uneatlantico.es.NodoTree;

public class BinaryTree<T> {

	private NodoTree<T> raiz;
	private ListG<T> list;

	public BinaryTree(){
		this.list =  new ListG<>();
	}

	public NodoTree<T> getRaiz() {
		return raiz;
	}
```

```
	public void setRaiz(NodoTree<T> raiz) {
		this.raiz = raiz;
	}

	public void Insert(T dato){
		NodoTree<T> nodoNuevo = new NodoTree<T>(dato);

		if (this.raiz == null)
			this.raiz = nodoNuevo;
		else{
			NodoTree<T> nodoAux = raiz;
			NodoTree<T>  padre;
			while (true){
				padre = nodoAux;
				if (new Integer((int) dato) < new Integer((int)
nodoAux.getDato())){
					nodoAux = nodoAux.getIzquierdo();
					if(nodoAux == null){
						padre.setIzquierdo(nodoNuevo);
						return;//Salir del metodo
					}
				}else{
						nodoAux = nodoAux.getDerecho();
						if (nodoAux == null){
							padre.setDerecho(nodoNuevo);
							return;//Salir de metodo
						}
				}

			}
		}
	}


	public ListG<T> InOrden(){
		System.out.println("In Orden");
		this.list = new ListG<>();
		this.InOrden(this.raiz);
		return this.list;
	}
```

```
//Izquierda raiz derecha
public ListG<T> PreOrden(){
	System.out.println("Pre Orden");
	this.list = new ListG<>();
	this.PreOrden(this.raiz);
	return this.list;
}

//Izquierda derecha raiz
public ListG<T> PostOrden(){
	System.out.println("Post Orden");
	this.list = new ListG<>();
	this.PostOrden(this.raiz);
	return this.list;
}

public ListG<T> BuscarPreOrden(T dato){
	System.out.println("Buscar Pre Orden");
	this.list = new ListG<>();
	this.BuscarPreOrden(dato);
	return list;
}

private void InOrden(NodoTree<T> raiz){
	if (raiz != null){
		InOrden(raiz.getIzquierdo());
		System.out.println(raiz.getDato());
		this.list.InsertIni(raiz.getDato());
		InOrden(raiz.getDerecho());
	}
}

private void PreOrden(NodoTree<T> raiz){
	if (raiz != null){
		System.out.println(raiz.getDato());
		this.list.InsertIni(raiz.getDato());
		PreOrden(raiz.getIzquierdo());
		PreOrden(raiz.getDerecho());
	}
}

private void BuscarPreOrden(NodoTree<T> raiz,T datoBuscar){
	if (raiz != null ){
```

```
                System.out.println(raiz.getDato());
                this.list.InsertIni(raiz.getDato());
                if (raiz.getDato().equals(datoBuscar))
                      return;
                PreOrden(raiz.getIzquierdo());
                PreOrden(raiz.getDerecho());
          }
    }

    private void PostOrden(NodoTree<T> raiz){
          if (raiz != null){
                PostOrden(raiz.getIzquierdo());
                PostOrden(raiz.getDerecho());
                System.out.println(raiz.getDato());
                this.list.InsertIni(raiz.getDato());
          }
    }


}
```

Tests

```
package casopractico;

import org.junit.Assert;
import org.junit.Test;

import data.uneatlantico.es.BinaryTree;

public class BinaryTreeTests {

      private BinaryTree<Integer> tree = new BinaryTree<>();
      private static final Integer[] arr = new Integer[] { 15, 9, 6, 14,
13, 20, 17, 64, 26, 72};



      @Test //Prueba si un arbol esta vacio
      public void EmptyTest() {
            Assert.assertTrue("Test failed for no empty",tree.getRaiz()
== null);
      }
```

```
	@Test //Prueba si un arbol no esta vacio
	public void NoEmpyTest() {
		this.CrearArbol();
		Assert.assertTrue("Test failed for no empty",tree.getRaiz()
!= null);
		Assert.assertEquals(new Integer(tree.PreOrden().Size()),new
Integer(arr.length));
	}

	@Test //Prueba la búsqueda en PreOrden
	public void BuscarPreorden() {
		this.CrearArbol();
		Integer dato = new Integer(14);
		Assert.assertEquals(this.tree.BuscarPreOrden(dato),dato);
	}



	private void CrearArbol(){
		this.tree = new BinaryTree<>();
		for (Integer dato : arr){
			this.tree.Insert(dato);
		}
	}



}
```


5.4. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Desarrollar una función que reciba un árbol binario de búsqueda que contenga números enteros. La función devolverá el número de elementos pares mayores que el nodo raíz.

2. Realizar una función que reciba un árbol como entrada y devuelva el proceso de mayor consumo de CPU ha realizado. Para ello cada nodo del árbol estará formado por un conjunto de información referente a un sistema de máquinas virtuales VMWare.

   Cada nodo almacenará el nombre del proceso, la fecha inicio y fecha fin.

Capítulo 6

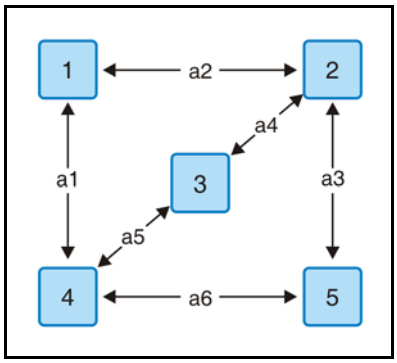
GRAFOS

6.1. DEFINICIÓN Y DISEÑO

Los grafos al igual que las estructuras de datos de tipo árbol son estructuras no lineales. Recordaremos que en este caso un elemento puede estar seguido por más de un elemento.

El concepto de grafo como estructura de datos desciende directamente desde su concepto matemático. Un grafo se representa como G = (V, E). Siendo V los vértices y E las aristas. Donde V es u conjunto finito y E es un conjunto que consta de los elementos de V.

Existen dos formas principales de representar los grafos. Mediante matrices de adyacencia o listas de adyacencia.



En este ejemplo tenemos: V=(1,2,3,4,5) y E=(a1,a2,a3,a4,a5,a6)

Representando este grafo como una matriz de adyacencia tenemos:

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

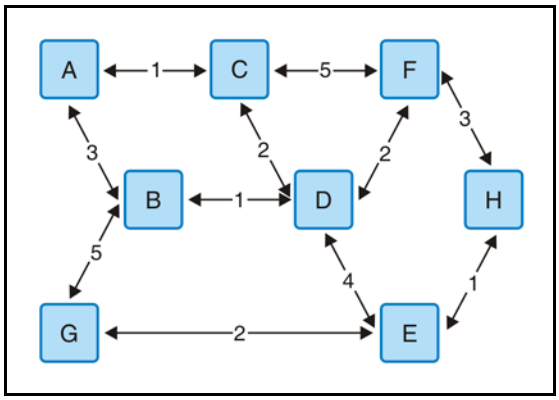
Lista de adyacencia:

- L(1) = 1 -> 2 -> 4
- L(2) = 2 -> 1 -> 3 -> 5
- L(3) = 3 -> 2 -> 4
- L(4) = 4 -> 1 -> 3 -> 5
- L(5) = 5 -> 2 -> 4

6.2. ALGORITMO DE DIJKSTRA

El algoritmo de Dijkstra o también denominado de camino más corto. Su objetivo es el de explorar todos los caminos más cortos que parten desde un vértice origen y llevan a todos los demás vértices. Cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, entonces el algoritmo se detiene.

Teniendo un grafo dirigido ponderado de N nodos no aislados, sea x el nodo inicial, un vector D de tamaño N guardará al final del algoritmo las distancias desde x al resto de los nodos.



Vamos a representar los etiquetados:

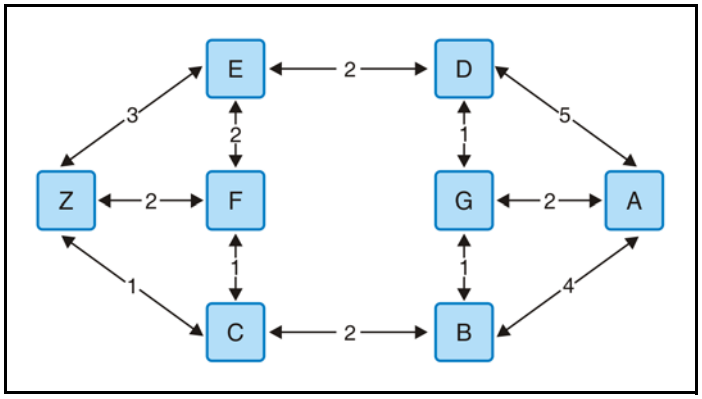
Vértice	Paso 1	Paso 2	Paso 3	Paso 4	Paso 5	Paso 6	Paso 7	Paso 8
A	(0,0)	--	*	*	*	*	*	*
B	(3,A)	--	(4,D) (3,A)	(3,A)	*	*	*	*
C	(1,A)	(1,A)	*	*	*	*	*	*
D	--	(3,C)	(3,C)	*	*	*	*	*
E	--	--	(7,D)	--	--	(7,D)	*	*
F	--	(6,C)	(6,C) (5,D)	--	(5,D)	*	*	*
G	--	--	--	(8,B)	--	(9,G) (8,B)	(8,B)	
H	--	--	--	--	(8,F)	(8,H)	*	(8,H)

6.3. IMPLEMENTACIÓN

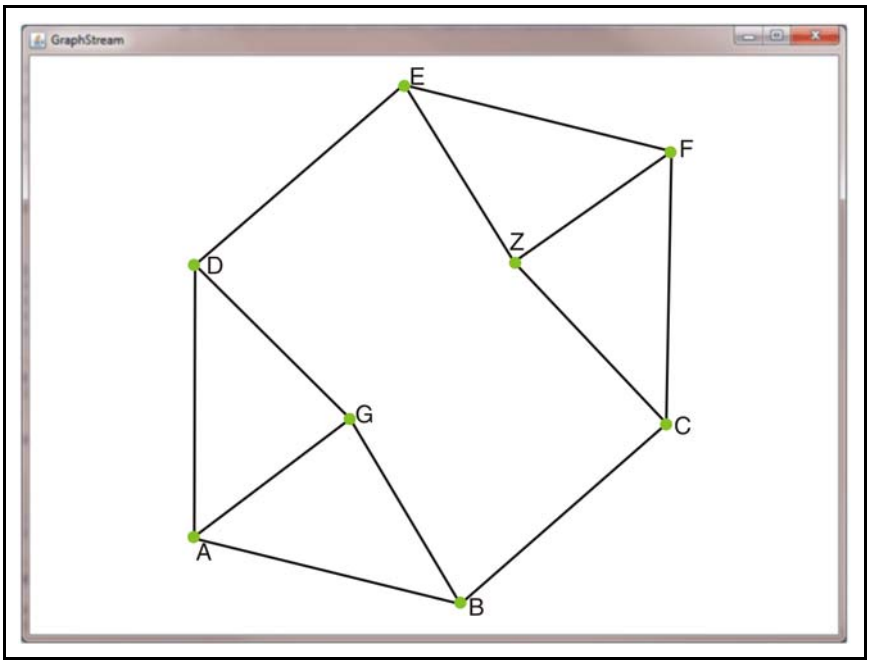
En esta parte vamos a analizar el uso de grafos con una librería ya construida previamente en java. Aprenderemos a utilizarla y a crear un grafo así como su representación gráfica. Seguir los siguientes pasos.

1. Descargarse la última versión de la librería gs-core.jar de la url http://graphstream-project.org/download/. En este jar se encuentra la funcionalidad más básica para trabajar con grafos.
2. Descargarse gs-algo.jar de l url http://graphstream-project.org/download/. Este otro jar nos permite trabajar con los diferentes algoritmos aplicables a grafos, como es el caso de Dijkstra.

3. Importar los jars en el proyecto de eclipse para poder empezar a trabajar.
4. Desarrollar el código necesario para crear el grafo de la siguiente imagen.
5. Cambiar el estilo del grafo con los nodos en verde y el texto en azul.



Este es el aspecto que debería tener el grafo obtenido



Código fuente

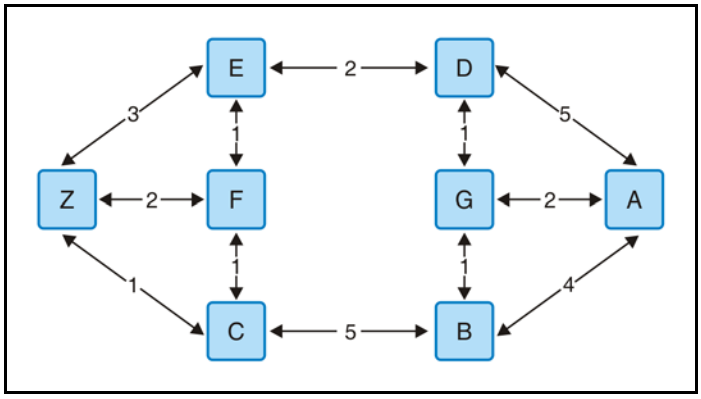
```
package graphstream.atlantico;
import org.graphstream.graph.*;
import org.graphstream.graph.implementations.SingleGraph;

public class Program {
```

```
private static String styleSheet =
        "node {" +
        "fill-color: blue;" +
        "}" ;


  public static void main(String[] args) {
        // TODO Auto-generated method stub
        Graph graph = CrearGrafo();

        Edge edge = Arista("F", "E", graph);
        if (edge != null)
              System.out.println("Arista : " + edge.toString());

        graph.display();
  }

  /**
   * Obtiene una arista entre dos nodos
   * @param nodo1
   * @param nodo2
   * @param graph
   * @return
   */
  private static Edge Arista(String nodo1, String nodo2,Graph graph){
        Node nodoTemp = graph.getNode(nodo1);
        if (nodoTemp.hasEdgeFrom(nodo2)){
              return nodoTemp.getEdgeFrom(nodo2);
        }
        return null;

  }

  /**
   * Crea un grafo y le asigna un nombre a cada uno de los nodos.
   * @return
   */
  private static Graph CrearGrafo(){
        Graph graph = new SingleGraph("Grafo");
```

```
            graph.addNode("A");
            graph.addNode("B");
            graph.addNode("C");
            graph.addNode("D");
            graph.addNode("E");
            graph.addNode("F");
            graph.addNode("G");
            graph.addNode("Z");
            //Nodo A
            graph.addEdge("AB", "A", "B");
            graph.addEdge("AG", "A", "G");
            graph.addEdge("AD", "A", "D");
            //Nodo B
            //graph.addEdge("BA", "B", "A");
            graph.addEdge("BG", "B", "G");
            graph.addEdge("BC", "B", "C");
            //Nodo D
            graph.addEdge("DG", "D", "G");
            graph.addEdge("DE", "D", "E");
            //Nodo E
            graph.addEdge("EF", "E", "F");
            graph.addEdge("EZ", "E", "Z");
            //Nodo C
            graph.addEdge("CF", "C", "F");
            graph.addEdge("CZ", "C", "Z");
            //Nodo F
            graph.addEdge("FZ", "F", "Z");

            for (Node node : graph) {
            node.addAttribute("ui.label", node.getId());
        }

            graph.addAttribute("ui.stylesheet", styleSheet);

            return graph;
    }

}
```

6.4. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Realizar la matriz de caminos mínimos siguiendo el algoritmo de Dijkstra del siguiente grafo. Utilizar para ello graphstream.



Capítulo 7

FICHEROS

7.1. DEFINICIÓN

Durante el desarrollo de aplicaciones software, es muy frecuente encontrarnos con la necesidad de almacenar información para que esta sea persistente en el tiempo. Para ello existen diferentes formatos de almacenamiento. Uno de los más frecuentes son los ficheros.

Un fichero es una colección de registros que se almacenan generalmente en disco. Un fichero suele tener una estructura determinada asociada a los datos almacenados. La definición de esa estructura y el formato de lectura y escritura viene definido por el programador.

Es muy común que asociada a un fichero exista una estructura de datos que represente la información en memoria y permita una gestión de esos datos mucho más flexible. El programador tendrá que desarrollar la interfaz de lectura y escritura de esas estructuras de datos en ficheros.

Por lo tanto en la gestión de ficheros intervienen tres niveles.

- El Programador define la estructura del fichero en base a las necesidades de los datos a almacenar. También implementará el código fuente que represente a esos datos y el formato de almacenamiento.
- El lenguaje de programación proporciona un conjunto de clases para la gestión interna de los ficheros y la comunicación con el sistema operativo.

- El **sistema operativo** es el encargado de gestionar la lectura y el almacenamiento en disco. Sus principales funciones son la asignación de espacio a los ficheros y el acceso a los ficheros ya guardados. El filesystem es el componente del sistema operativo encargado de esta operativa. Windows (NTFS, FAT 32), Unix (UFS,FFS).

Los ficheros se puede decir que tienen las siguientes características:

- La información se almacena de forma permanente.
- Se identifican mediante un nombre.
- Pueden tener derechos de acceso.
- Permiten acceso concurrente en determinados casos.
- La información se agrupa en registros lógicos.
- Acceso a los mismos puede ser secuencial o directo.

7.2. TIPOS DE FICHEROS

Existen diferentes formas de clasificar los ficheros:

1. Por el tipo de información que almacenan:
   a) Ficheros de texto
   b) Ficheros de audio
   c) Ficheros ejecutables
   d) Etc...
2. Por el formato del fichero:
   a) Ficheros de texto: Normalmente están formados por una secuencia de caracteres ASCII. El formato se suele organizar en líneas y se puede abrir y entender por cualquier editor de texto.
   b) Ficheros binarios: Los datos se almacenan en formato binario y tienen una estructura conocida por los programas que los usan.

La clasificación que más nos interesa desde el punto de vista de la asignatura es por el tipo de acceso. Desde esta perspectiva podemos hablar de:

- Ficheros secuenciales: Este tipo de ficheros se caracteriza porque para acceder a un determinado registro es necesario recorrerlo desde el principio.

- Ficheros de acceso directo: Para poder usar estos ficheros lo primero es que el hardware permita el acceso a los datos de manera direccionable. Hoy in día esto es posible en casi todos los dispositivos de almacenamiento. Este tipo de acceso es mucho más beneficioso que el secuencial, por la ventaja que se podrá acceder directamente al registro que se quiera.
- Ficheros indexados: Estos ficheros almacenan una clave y la dirección relativa a los registros. De esa manera sólo consultando el índice permiten un acceso mucho más rápido a la información. Estos ficheros se verán en la segunda parte de la asignatura con el framework lucene.

7.3. OPERACIONES CON FICHEROS

Vamos a intentar describir las operaciones más comunes con ficheros.

- Crear (Create).
- Borrar (Delete).
- Abrir (Open).
- Cerrar (Close).
- Leer (Read).
- Escribir (Write).
- Añadir (Append).
- Buscar (Seek).
- Renombrar (Rename).

7.4. IMPLEMENTACIÓN

Dentro de la manipulación de ficheros con Java tenemos principalmente ficheros de texto y ficheros binarios.

Ficheros de texto: Las clases **FileReader** y **FileWriter** son las que nos permiten leer y escribir ficheros de texto en Java.

```
FileWriter fw = new FileWriter("C:\\fichero.txt");

FileReader fr = new FileReader("C:\\fichero.txt");
```

También podemos usar las clases **BufferedReader** y **BufferedWriter**. Estas clases sirven también para leer y escribir ficheros pero optimizan las funciones.

```
BufferedReader br=new BufferedReader(new FileReader("C:\\fichero.txt"));

BufferedWriter bw=new BufferedWriter(new FileWriter("C:\\fichero.txt"));
```

Ficheros binarios: Las clases **DataInputStream** y **DataOutputStream** nos permiten escribir y leer datos en ficheros binarios, indicando un tipo. Los ficheros binarios se tienen que leer como se escriben.

7.5. CASO PRÁCTICO

Desarrollar un programa que nos permita la gestión de la administración de loterías del estado y Europa. Para ello vamos a implementar la siguiente funcionalidad.

1. El programa tiene 3 módulos. Vender lotería, cerrar lotería y comprobar número premiado.
2. Se tendrá que desarrollar una estructura de datos para representar un boleto de lotería. Cada boleto tiene un número, fecha, tipo (estatal, europea).
3. El programa presentará las siguientes opciones por pantalla:
   a) Comprar boleto lotería. Se generará un número aleatorio entre 1 y 1.000.000. No se puede vender el mismo número dos veces. Al comprar el boleto se debe almacenar de forma persistente los números vendidos y el dinero recaudado. Tendremos que ofertar el tipo de lotería y a poder ser almacenar los datos de forma independiente. Sólo se puede vender lotería del día en curso.
   b) Cerrar venta lotería. Cuando se selecciona esta opción no se permite la venta de más boletos y se selecciona un número premiado entre los vendidos.
   c) Comprobar número premiado. Permite introducir un número por teclado y la fecha y el sistema te verifica si el número esta premiado. En caso de que esté premiado se calculará en valor obtenido como premio. Será 50% del dinero recaudado.

Clase Boleto

```
package loteria;

import java.util.Calendar;

/**
 * Clase que representa un boleto de loteria
 * @author SMaroto
 */
public class Boleto {
	private int num;
	private Calendar fecha;
	private TipoBoleto tipo;

	public Boleto(int num,Calendar fecha,TipoBoleto tipo){
		this.num = num;
		this.fecha = fecha;
		this.tipo = tipo;
	}

	public int getNum() {
		return num;
	}
	public void setNum(int num) {
		this.num = num;
	}
	public Calendar getFecha() {
		return fecha;
	}
	public void setFecha(Calendar fecha) {
		this.fecha = fecha;
	}
	public TipoBoleto getTipo() {
		return tipo;
	}
	public void setTipo(TipoBoleto tipo) {
		this.tipo = tipo;
	}


}
```

Enum EstadoLoteria

```
package loteria;

public enum EstadoLoteria {
	ABIERTO,
	CERRADO;

}
```

Enum TipoBoleto

```
package loteria;

public enum TipoBoleto {
	ESTATAL,
	EUROPEA;

}
```

Clase GestionFichero

```
package loteria;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
/**
 * Clase que gestiona lectura y escritura a disco
 * @author SMaroto
 *
 */
public class GestionFichero {
	private File file = null;

	public GestionFichero(String pathFichero){
		file = new File(pathFichero);
		if (!file.exists())
			try {
				file.createNewFile();
			} catch (IOException e) {
```

```
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }
            }

            public void EscribirLineaFichero(String linea) throws IOException{
                  BufferedWriter output = null;
                  try {
                        output = new BufferedWriter(new FileWriter(file));
                        output.write(linea);
                    output.write(linea);
                    output.close();
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }
                  finally{
                        output.close();
                  }

            }

}

package loteria;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.HashMap;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

public class DictionarioLoterias {
```

```
	private  HashMap<String, HashMap<TipoBoleto,List<Integer>>>
dictionarioLoterias;
	SimpleDateFormat sdf = null;

	public DictionarioLoterias(){
		this.dictionarioLoterias = new HashMap<>();
	    this.sdf = new SimpleDateFormat("dd/MM/yyyy");
	}

	public Integer ComprarBoleto(TipoBoleto tipoBoleto){
		//Generar numero aleatorio
		return GenerarNumero(tipoBoleto);
	}

	public void CerrarLoteriaSalvarEnDisco(){
		//Salvo a disco la loteria vendida ese dia
		HashMap<TipoBoleto,List<Integer>> loteriaHoy =
dictionarioLoterias.get(GetHashKeyFechaHoy());
		try {
			GestionFichero fichero = new
GestionFichero("c:\\loterias.txt");
			fichero.EscribirLineaFichero(
	GenerarStringLoteria(TipoBoleto.ESTATAL,loteriaHoy));
			fichero.EscribirLineaFichero(
	GenerarStringLoteria(TipoBoleto.EUROPEA,loteriaHoy));

		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public boolean EsBoletoPremiado(String num){
		return true;
	}

	public void CargarLoteriaDeDisco(){
		//Salvo a disco la loteria vendida ese dia

	}
```

```
	private String GenerarStringLoteria(TipoBoleto
tipoBoleto,HashMap<TipoBoleto,List<Integer>> loteriaHoy){
		StringBuilder builder = new StringBuilder();
		builder.append(tipoBoleto.toString() + ",");

		if (loteriaHoy.get(tipoBoleto) == null)
			return builder.append("\n").toString();

		for (Integer i: loteriaHoy.get(tipoBoleto)) {
			builder.append(i + ",");
		}
		builder.append("\n");
		return builder.toString();
	}

	//dictionarioLoterias.get(GetHashKeyFechaHoy()).get(tipoBo-
leto).contains(num)){
	//Genera un numero de loteria que no se haya vendido
	private Integer GenerarNumero(TipoBoleto tipoBoleto){
		Integer num = null;
		while (num == null){
			num = new Integer(ThreadLocalRandom.current().nex-
tInt(1, 1000000));
			if (dictionarioLoterias.isEmpty() ||
	dictionarioLoterias.get(GetHashKeyFechaHoy()).isEmpty()){
				 dictionarioLoterias.put(GetHashKeyFechaHoy(),
CrearDictionario(tipoBoleto,num));

			}else if
(dictionarioLoterias.get(GetHashKeyFechaHoy()).get(tipoBoleto) == null){
				List<Integer> lista = new ArrayList<Integer>();
				lista.add(num);
	dictionarioLoterias.get(GetHashKeyFechaHoy()).put(tipoBo-
leto,lista);
			}else if
(!dictionarioLoterias.get(GetHashKeyFechaHoy()).get(tipoBoleto).con-
tains(num)){
	dictionarioLoterias.get(GetHashKeyFechaHoy()).get(tipoBo-
leto).add(num);
			}
```

```
            }
            return num;
      }

      private HashMap<TipoBoleto,List<Integer>> CrearDictionario(TipoBo-
leto tipoBoleto,Integer num){
            List<Integer> lista = new ArrayList<Integer>();
            lista.add(num);
            HashMap<TipoBoleto,List<Integer>> fechaListNum =  new
HashMap<>();
            fechaListNum.put(tipoBoleto,lista);
            return fechaListNum;
      }


      private String GetHashKeyFechaHoy(){

            return sdf.format(Calendar.getInstance().getTime());
      }

      private String GetHashKeyFecha(int day,int month,int year){
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
            return sdf.format(Calendar.getInstance().getTime());
      }

}

package loteria;


import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Calendar;

import java.util.HashMap;



public class Program {
```

```
	private static EstadoLoteria ESTADO;
	private static DictionarioLoterias dictionarioLoterias = new Dic-
tionarioLoterias();
	private static BufferedReader br =
			new BufferedReader(new InputStreamReader(System.in));

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		ESTADO = EstadoLoteria.ABIERTO;
		String option ="";
		while (option.equals("") || !option.equals("5")){
			PrintOptions();
			try {
				option = br.readLine();
				switch(option){
				case "1" :
					ComprarBoleto(TipoBoleto.ESTATAL);
					break;
				case "2" :
					ComprarBoleto(TipoBoleto.EUROPEA);
					break;
				case "3" :
					CerrarLoteria();
					break;
				case "4" :
					ComprobarBoletoPremiado();
					break;
				case "5" :
```

```
                        //terminando programa
                        break;
                  default :
                        System.out.println("Opción no reconocida
:");
                  }
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }

}
private static void PrintOptions(){
      System.out.println("---------------------------");
      System.out.println("Opciones de loteria:");
      System.out.println("1: Comprar boleto nacional");
      System.out.println("2: Comprar boleto europeo");
      System.out.println("3: Cerrar venta loteria");
      System.out.println("4: Comprobar boleto premiado");
      System.out.println("---------------------------");
  }

  private static void ComprarBoleto(TipoBoleto tipoBoleto){
      if(ESTADO == EstadoLoteria.ABIERTO){
            Integer num = dictionarioLoterias.ComprarBoleto(tipoBo-
leto);
            System.out.println("Boleto comprado número : " + num );
```

```
            }else
                  System.out.println("Loteria cerrada. Hay que esperar al
siguiente día para vender");
      }

      private static void CerrarLoteria(){
            if (ESTADO == EstadoLoteria.CERRADO){
                  System.out.println();
                  System.out.println("Loteria cerrada con anterioridad");
                  System.out.println();
            }

            System.out.println();
            System.out.println("Cerrando venta lotería");
            System.out.println();
            ESTADO = EstadoLoteria.CERRADO;
            dictionarioLoterias.CerrarLoteriaSalvarEnDisco();
      }

      private static void ComprobarBoletoPremiado() throws IOException{
            String num ="";
            System.out.println();
            System.out.println("Introduce el número premiado");
            System.out.println();
            num = br.readLine();
            if (dictionarioLoterias.EsBoletoPremiado(num)){
                  System.out.println();
                  System.out.println("Boleto premiado");
```

```
                System.out.println();
            }else{
                System.out.println();
                System.out.println("Boleto no premiado");
                System.out.println();
}
```

7.6. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Dados dos ficheros de texto que se crearán manualmente que almacenen números enteros de menor a mayor, generar un tercer fichero que contenga los números de los ficheros anteriores sin duplicados y que estén ordenados de mayor a menor.
2. Crear un fichero que permita almacenar la información relativa al dinero disponible de un cajero automático. El cajero puede almacenar billetes de 10, 20, 50 Euros. El cajero empieza vacío y existe la posibilidad de recargar y retirar dinero.
3. Crear manualmente un fichero que almacene las notas obtenidas en asignaturas por alumnos. Desarrollar un programa que liste los alumnos y las notas obtenidas por asignatura.

Capítulo 8

PROYECTO

El proyecto final va a ser un programa complejo en el que se va a intentar solucionar un problema bastante común en el mundo real utilizando grafos. Para ello vamos a sumar varias dimensiones a nuestro proyecto. Por un lado el mundo de los grafos, el mundo de la visualización y de la información Linked Data.

Linked Data es un proyecto perteneciente a W3C consortium que es la base para la constitución de la denominada Web Semántica. La idea es que se proporcione información en la web que permita que los datos en la misma están dotados de significado y que a su vez estén relacionados entre sí. Para ello se utiliza RDF que es una especie de XML más completo.

Nuestro proyecto va a extraer información de un repositorio Linked Data como es Open Flights. Aquí podemos encontrar datos de los países, aerolíneas y rutas comerciales entre los diferentes países y aerolíneas.

Lo que se va a desarrollar por lo tanto es un grafo donde los nodos van a ser representados por los aeropuertos, y las aristas van a ser las rutas entre esos aeropuertos. Vamos a representar visualmente esa relación y permitir realizar algunas consultas simples. Por ejemplo: Cuál es el camino más corto entre dos puntos?. Cuáles son los destinos posibles desde un aeropuerto?.

Para ello vamos a cargar los datos en un grafo utilizando la librería GraphStream. Esta librería nos va a permitir ejecutar algoritmos como Dijkstra para encontrar los caminos más cortos.

Árboles

Enlace web: https://www.youtube.com/watch?v=h_8WTkH1OV4

Enlace web: https://www.youtube.com/watch?v=ZKnwBJ8q2TE

Grafos

Enlace web: https://www.youtube.com/watch?v=S_5W5bImTH4

Enlace web: https://www.youtube.com/watch?v=LLx0QVMZVkk

Enlace web: http://openflights.org/demo/openflights-routedb-2048.png

Enlace web: http://openflights.org/data.html