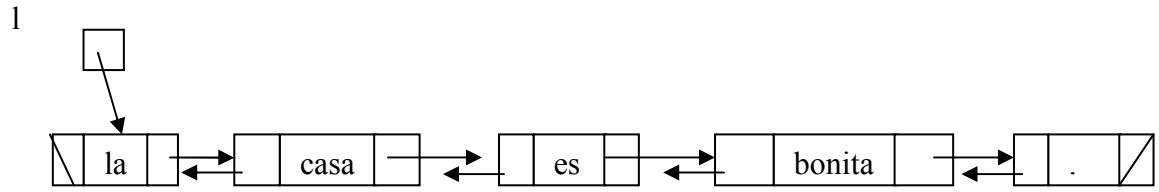
ESTRUCTURAS DE DATOS Y ALGORITMOS

ENERO 2012

1. (1,5 puntos) Tenemos una lista doblemente ligada que contiene una lista de palabras:



Se quiere implementar la siguiente función:

```
public DoubleNode<T> {
   T data;
   DoubleNode<T> next;
   DoubleNode<T> prev;
}

public DoubleLinkedList<T> {

   DoubleNode<T> first;
}

public ListaDePalabras extends DoubleLinkedList<String> { // herencia

   public SimpleLinkedList<String> obtenerContexto(String pal, Integer n)
   // pre: la palabra "pal" se encuentra en la lista
   //      n >= 0
   // post: devuelve una lista que contiene la palabra buscada junto con
   //      las "n" palabras anteriores y posteriores a "pal"
   //      En caso de que la palabra "pal" aparezca varias veces, se tendrá
   //      en cuenta únicamente la primera aparición
   //      En caso de que no haya n palabras delante o detrás de "pal"
   //      se devolverán solamente las que haya
}
```

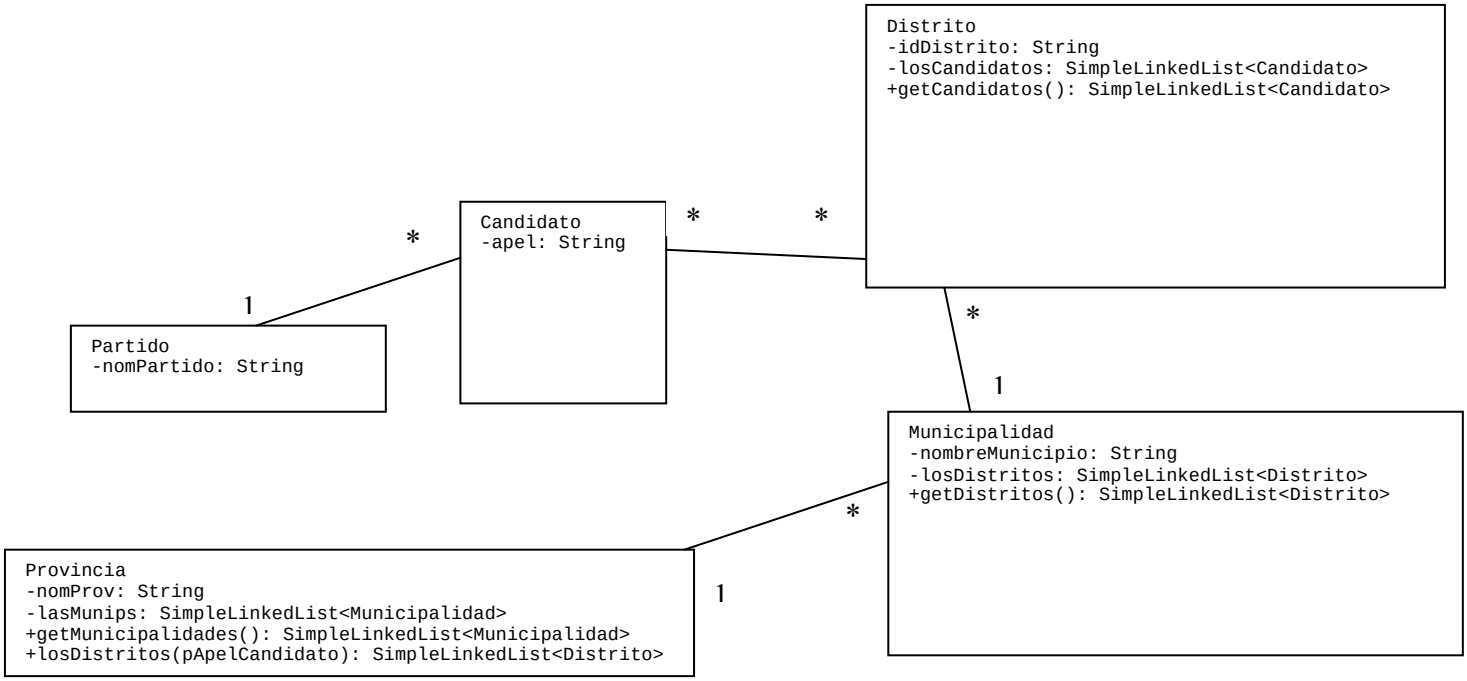
Por ejemplo, la llamada a obtenerContexto("es", 1) devolverá la lista <"casa", "es", "bonita">.

Se pide:

- Implementar el algoritmo
- Establecer de manera razonada el coste del algoritmo.

2. Elecciones (1,5 puntos)

Dado el siguiente diagrama de clases:



Y dada la siguiente clase:

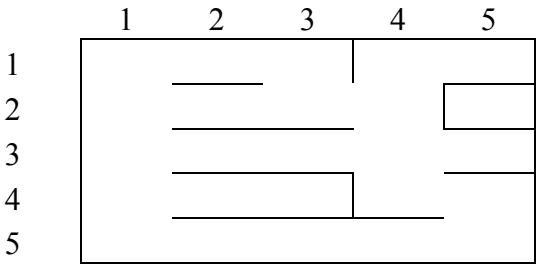
```
public class Candidatura {
   String partido;
   String apelCandidato;
   String idDistrito;
   String municipio;
   String prov;
}
```

Se pide implementar el siguiente método de la clase Eleccion, calculando **razonadamente** su coste:

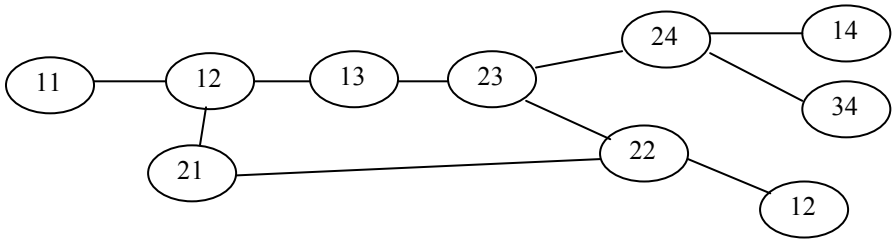
```
public class Eleccion {

   SimpleLinkedList<Partido> partidos;     // todos los partidos
   SimpleLinkedList<Candidato> candidatos; // todos los candidatos
   SimpleLinkedList<Distrito> distritos;   // todos los distritos
   SimpleLinkedList<Municipio> municipios; // todos los municipios
   SimpleLinkedList<Provincia> provincias; // todas las provincias

   public añadirCandidaturas(SimpleLinkedList<Candidatura> lc)
   // Pre: las listas de partidos, candidatos, distritos,
   //      municipios y provincias se encuentran vacías
   // Post: se han añadido las candidaturas dadas, actualizando
   //       todos los datos
}
```

3. Laberinto (1,5 puntos)

La siguiente figura muestra un ejemplo de un laberinto.



Se puede representar el laberinto por medio de un grafo, que presentamos parcialmente en la siguiente figura:



Cada nodo representa una casilla y queda identificado por un string que indica la fila y la columna (por ejemplo, el nodo "23" indica que corresponde a la fila 2, columna 3). El grafo contiene un arco entre dos nodos si corresponden a dos casillas consecutivas y es posible pasar de una a otra.

Queremos desarrollar un algoritmo que, dados dos puntos del laberinto, nos devuelva un camino de longitud mínima que conecta esos dos puntos. El algoritmo devolverá:

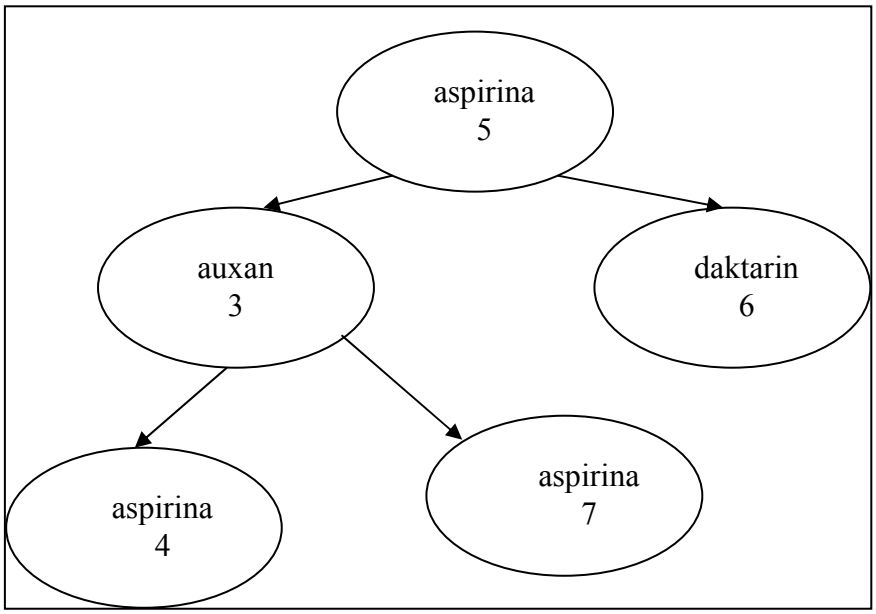
- La lista vacía si no hay ningún camino que los une
- En caso de que haya varios caminos de longitud mínima, bastará con devolver cualquiera de ellos

Por ejemplo, la llamada buscarCamino("11", "35") devolverá la lista <11, 12, 13, 23, 24, 34, 35> (obsérvese que hay otros caminos de longitud igual o superior a este).

```
public class GraphAL<T> implements GraphADT<T>
{
   protected final int DEFAULT_CAPACITY = 100;
   protected int numVertices;   // number of vertices in the graph
   protected LinkedList<Integer>[] adjList;   // adjacency list
   protected T[] vertices;   // values of vertices

}

public class GrafoLaberinto extends GraphAL<String> {
   public SimpleLinkedList<String> buscarCamino(String inicio, String fin)
}
```

4. Conversión de árbol a tabla hash (1,5 puntos)

Dado un árbol binario que contiene datos de ventas de productos (en cada nodo aparece un par del tipo (producto, unidades vendidas)), queremos un subprograma que vuelque los elementos del árbol a una tabla hash, obteniendo un solo elemento para cada producto, cuyo valor asociado será el total de las ventas de ese producto.



Dado el árbol ejemplo, la tabla hash resultante contendrá el siguiente resultado:

0		
1	daktarin	6
2		
3	aspirina	16
4	auxan	4
5		

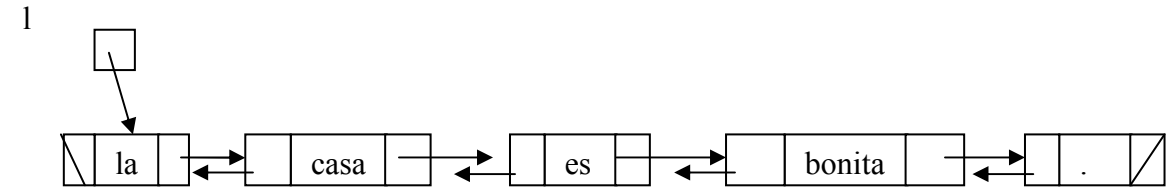
```
public class BinaryTreeNode<T> {
       protected T content;
       protected BinaryTreeNode<T> left;
       protected BinaryTreeNode<T> right;
}
public class BinaryTree<T> {
   protected int count;
   protected BinaryTreeNode<T> root;
}

public class Medicina {
   int ventas;
   String nombre;
}

public class ArbolMedicinas extends BinaryTree<Medicina> {

   public HashMap<String, Integer> convertirArbolEnTH()
   // Post: el resultado es una table hash con los elementos del árbol, donde
   //     cada elemento aparece una sola vez (el dato asociado a cada producto
   //     es el total de ventas de ese producto)
}
```

DATU-EGITURAK ETA ALGORITMOAK

URTARRILA 2012

1. Testuingurua lortu (1,5 puntu)

Hitz-zerrenda bat gordeko duen estekadura bikoitzeko zerrenda dugu:



Funtzio hau inplementatu nahi da:

```
public DoubleNode<T> {
   T data;
   DoubleNode<T> next;
   DoubleNode<T> prev;
}

public DoubleLinkedList<T> {

   DoubleNode<T> first;
}

public HitzZerrenda extends DoubleLinkedList<String> { // herentzia

   public SimpleLinkedList<String> testuinguruaLortu(String hitz, Integer n)
   // aurre: "hitz" hitza zerrendan dago
   //      n >= 0
   // post: "hitz" hitza eta bere aurreko eta ondorengo "n" hitzak
   //        dituen zerrenda bueltatuko da
   //      "hitz" hitza behin baino gehiagotan agertuko balitz,
   //      orduan lehen agerpena hartuko da kontuan
   //      Hitzaren aurrean (edo atzean) "n" hitz baino gutxiago
   //      baldin badaude, orduan daudenak bueltatuko dira
}
```

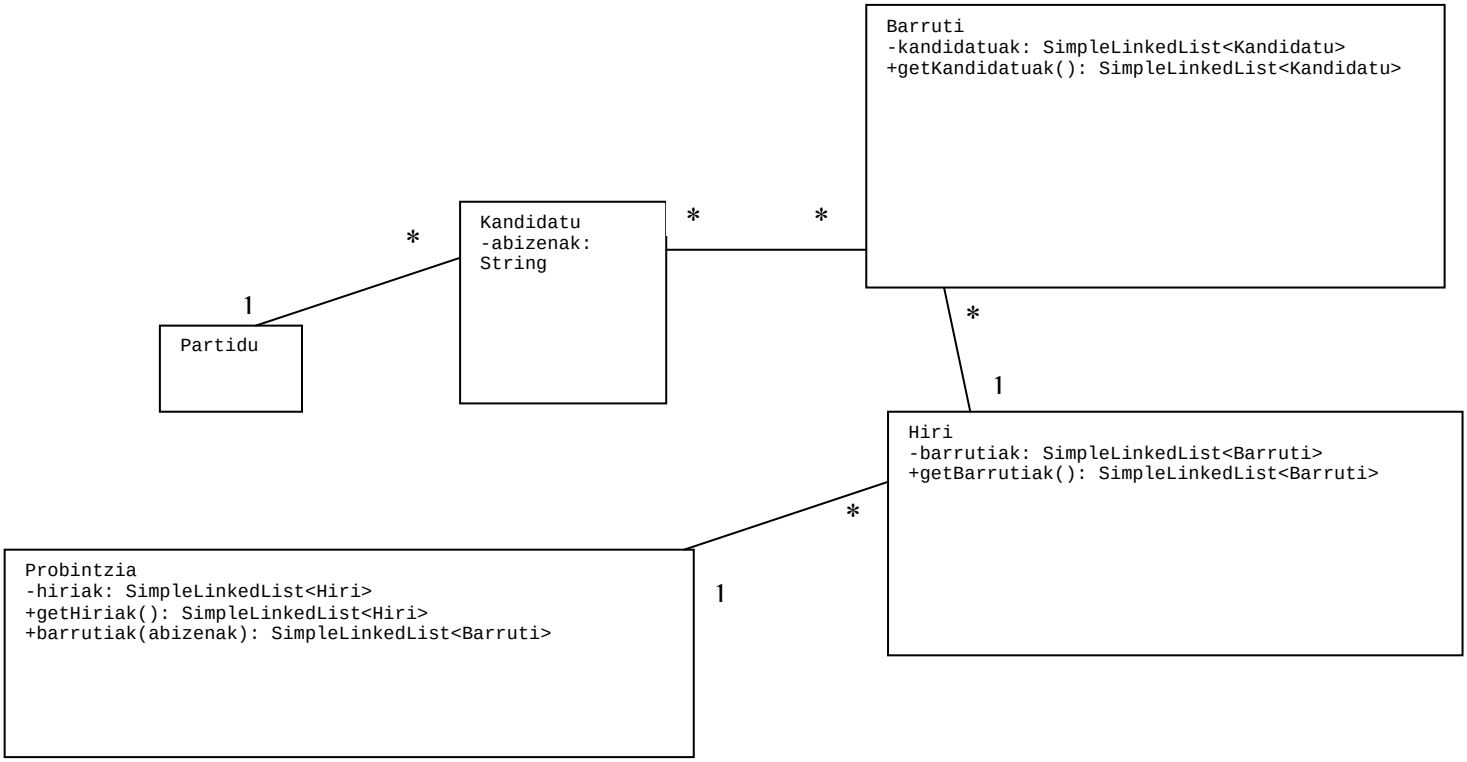
Adibidez, testuinguruaLortu("es", 1) deiak lista hau bueltatuko luke: <"casa", "es", "bonita">.

Ondokoa eskatzen da:

- Algoritmoa inplementatu
- Modu arrazoituan algoritmoaren kostua kalkulatu.

2. Hauteskundeak (1,5 puntu)

Ondoko klase-diagrama emanda:



Eta klase hau emanda:

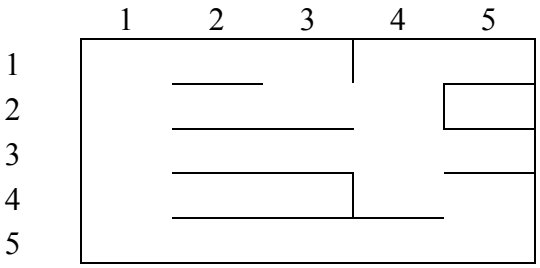
```
public class Kandidatura {
   String partidu;
   String abizenakKandidatu;
   String idBarruti;
   String hiri;
   String prob;
}
```

Hauteskunde klasearen ondorengo metodoa inplementatzea eskatzen da, bere kostua era arrazoituan kalkulatuz:

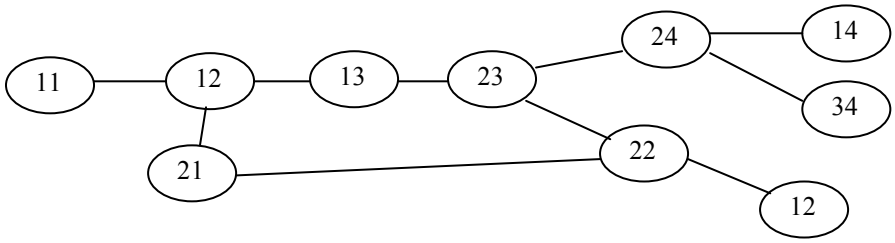
```
public class Hauteskunde {

   SimpleLinkedList<Partidu>    partiduak;   // partidu guztiak
   SimpleLinkedList<Kandidatu>  kandidatuak; // kandidatu guztiak
   SimpleLinkedList<Barruti>    barrutiak;   // barruti guztiak
   SimpleLinkedList<Hiri>       hiriak;      // hiri guztiak
   SimpleLinkedList<Probintzia> probintziak; // probintzia guztiak

   public kandidaturakGehitu(SimpleLinkedList<Kandidatura> kz)
   // Aurre: partiduen, kandidatuen, barrutien, hirien eta
   //        probintzien zerrendak hutsik daude
   // Post: kandidaturak gehitu dira, datu guztiak eguneratuz
}
```

3. Labirintoa (1,5 puntu)

Ondoko irudiak labirinto bat adierazten du.



Labirintoa adierazteko grafo bat erabil daiteke. Ondoko irudiak goiko labirintoaren zati bat adierazten du:



Adabegi bakoitzak lauki bat adierazten du, eta string baten bidez emango da (adibidez, “23” adabegiak bigarren lerroko eta hirugarren zutabeko laukia adierazten du). Arku bat egongo da bi adabegiren artean ondoz ondoko laukiak baldin badira, eta batetik bestera pasatzea badagoenean.

Algoritmo bat egin nahi dugu, labirintoko bi lauki emanda, bi elementu horiek konektatzen dituen bide minimoa emango diguna. Algoritmoak hau bueltatuko du:

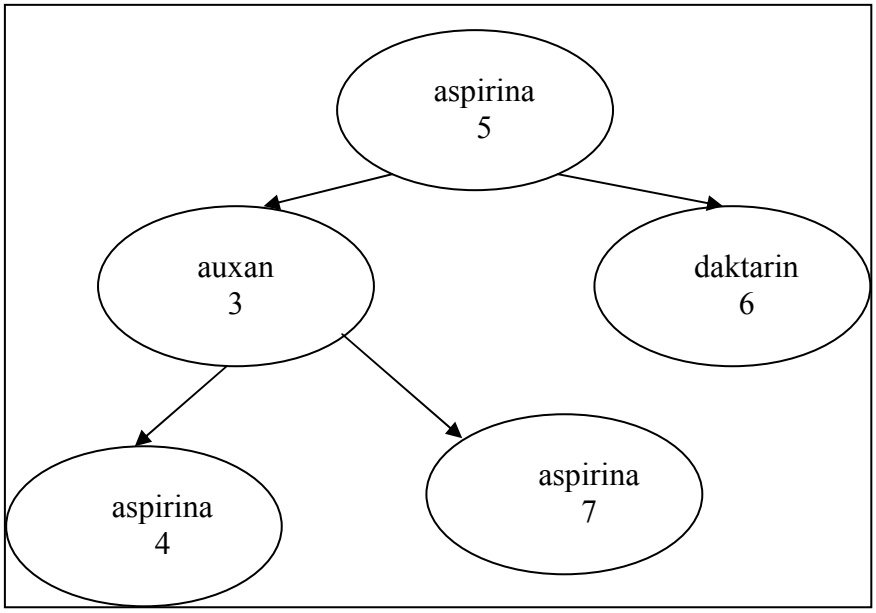
- Zerrenda hutsa, adabegien arteko konexiorik ez dagoenean
- Luzera minimoko bide bat baino gehiago balego, orduan edozein bueltatu daiteke

Adibidez, bilatuBidea(“11”, “35”) deaiak <11, 12, 13, 23, 24, 34, 35> zerrenda bueltatuko luke (ikus daiteke badaudela luzera horretako edo gehiagoko beste bide batzuk).

```
public class GraphAL<T> implements GraphADT<T>
{
   protected final int DEFAULT_CAPACITY = 100;
   protected int numVertices;   // number of vertices in the graph
   protected LinkedList<Integer>[] adjList;   // adjacency list
   protected T[] vertices;   // values of vertices

}

public class GrafoLabirinto extends GraphAL<String> {
   public SimpleLinkedList<String> bilatuBidea(String hasiera, String bukaera)
}
```

4. Zuhaitzaren bihurketa (1,5 puntu)

Produktuen salmentak dituen zuhaitz bitarra emanda (adabegi bakoitzean (produktua, saldutako unitateak) moduko bikotea), zuhaitz horretako elementuak hash taula batera pasako dituen azpiprograma nahi dugu, produktu bakoitzeko elementu bakarra emanez, produktu horren salmenta guztiekin.



Adibideko zuhaitza emanda, lortuko den hash taulak hau izango luke:

0		
1	daktarin	6
2		
3	aspirina	16
4	auxan	4
5		

```
public class BinaryTreeNode<T> {
       protected T content;
       protected BinaryTreeNode<T> left;
       protected BinaryTreeNode<T> right;
}

public class BinaryTree<T> {
   protected int count;
   protected BinaryTreeNode<T> root;
}

public class Produktu {
   int salmentak;
   String izena;
}

public class ProduktuenZuhaitza extends BinaryTree<Produktu> {

   public HashMap<String, Integer> zuhaitzaHTBihurtu()
}
```

```
import java.util.LinkedList;

public class ListadePalabras extends DoubleLinkedList<String> { // herencia

          public LinkedList<String> obtenerContexto(String pal, Integer n) {
          // pre: la palabra “pal” se encuentra en la lista
          //      n >= 0
          // post: devuelve una lista que contiene la palabra buscada junto con
          //      las “n” palabras anteriores y posteriores a “pal”
          //      En caso de que la palabra “pal” aparezca varias veces, se
          //      tendrá en cuenta únicamente la primera aparición
          //      En caso de que no haya n palabras delante o detrás de “pal”
          //      se devolverán solamente las que haya

                 // primero buscar el elemento
                 DoubleNode<String> act = first;
                 while (!act.data.equals(pal)){
                        act = act.next;
                 }

                 LinkedList<String> res = new LinkedList<String>();
                 res.add(pal);

                 // añadir los anteriores
                 Integer cont = 0;
                 DoubleNode<String> aux = act.prev;
                 while ((cont < n) && (aux != null)) {
                        res.addFirst(aux.data);
                        aux = aux.prev;
                 }

                 // añadir los posteriores
                 cont = 0;
                 aux = act.next;
                 while ((cont < n) && (aux != null)) {
                        res.addLast(aux.data);
                        aux = aux.next;
                 }
                 return res;
          }
       }
```

```
public class Eleccion {

   SimpleLinkedList<Partido> partidos;     // todos los partidos
   SimpleLinkedList<Candidato> candidatos; // todos los candidatos
   SimpleLinkedList<Distrito> distritos;   // todos los distritos
   SimpleLinkedList<Municipio> municipios; // todos los municipios
   SimpleLinkedList<Provincia> provincias; // todas las provincias

   public añadirCandidaturas(SimpleLinkedList<Candidatura> lc) {
   // Pre: las listas de partidos, candidatos, distritos,
   //      municipios y provincias se encuentran vacías
   // Post: se han añadido las candidaturas dadas, actualizando
   //       todos los datos

     lc.goFirst();
     while (lc.hasNext()) {
       Candidatura elem = lc.get();
       lc.goNext();

       // actualizar o añadir el partido
       Partido part = new Partido(elem.partido);
       if !partidos.find(part) { partidos.insertLast(part); }

       // buscar o añadir el candidato
       Candidato cand = new Candidato(elem.apelCandidato);
       if (!candidatos.find(cand)) {
         candidatos.insertLast(cand);
       }
       else {
         cand = candidatos.get();
       };

       // actualizar o añadir el distrito
       Distrito dist = new Distrito(elem.idDistrito);
       if !distritos.find(dist) {
         dist.losCandidatos.insert(cand);
         distritos.insertLast(dist);
       }
       else {
         dist = distritos.get();
         dist.losCandidatos.insertLast(cand);
       }

       // actualizar o añadir el municipio
       Municipio munip = new Municipio(elem.municipio);
       if !municipios.find(munip) {
         munip.losDistritos.insert(dist);
         municipios.insertLast(munip);
       }
       else {
         munip = municipios.get();
         munip.losDistritos.insertLast(dist);
       }

       // actualizar o añadir la provincia
       Provincia prov = new Provincia(elem.prov);
       if !provincias.find(prov) {
         prov.lasMunips.insert(munip);
         provincias.insertLast(prov);
       }
       else {
         prov = provincias.get();
         prov.lasMunips.insertLast(munip);
       }
     } // while
   } // añadirCandidaturas

}
```

Solución ej. 3 (es el mismo problema de la práctica 3: estanRelacionados):

```
public class GraphAL<T> implements GraphADT<T>
{  protected final int DEFAULT_CAPACITY = 100;
   protected int numVertices;   // number of vertices in the graph
   protected LinkedList<Integer>[] adjList;   // adjacency list
   protected T[] vertices;   // values of vertices
}

package jss2;
import java.util.Iterator;
import java.util.LinkedList;

public class GrafoLaberinto extends GraphAL<String> {

          public LinkedList<String> buscarCamino(String inicio, String fin) {

            LinkedList<String> res = new LinkedList<String>();
            boolean[] visitado = new boolean[numVertices];
            Integer[] anterior = new Integer[numVertices];
            LinkedQueue<Integer> c = new LinkedQueue<Integer>(); // lista vacía

            Integer indFin = getIndex(fin);
            Integer indInicio = getIndex(inicio);

            for (int i = 0; i < TAMANO; i++) {
              visitado[i] = false;
              anterior[i] = -1;
            }

            c.enqueue(indInicio); // meter el primer nodo en la cola
            visitado[indInicio] = true;
            boolean enc = false;

            while ((!c.isEmpty()) && (!enc)) {
              Integer u = c.first();
              c.dequeue();
              if (u == indInicio) { enc = true;};

              Iterator<Integer> it = adjList[u].iterator();
              while (it.hasNext()) {
                Integer w = it.next();

                if (!visitado[w]) {
                   visitado[w] = true;
                   anterior[w] = u;
                   c.enqueue(w);
                }
              } // while

            if (enc) {
               // obtener el camino recorriendo los enlaces en orden inverso
               // (desde el fin hasta el inicio)
              Integer ind = indFin;
              while (ind != indInicio) {
                res.addFirst(vertices[ind]);
                ind = anterior[ind];
              }
              res.addFirst(vertices[indInicio]);
             } // if
            } // while

            return res;
          } // buscarCamino


       }
```

```
public class BinaryTreeNode<T> {
       protected T content;
       protected BinaryTreeNode<T> left;
       protected BinaryTreeNode<T> right;
}
public class BinaryTree<T> {
   protected int count;
   protected BinaryTreeNode<T> root;
}

public class Medicina {
   int ventas;
   String nombre;
}

public class ArbolMedicinas extends LinkedBinaryTree<Medicina> {

  public HashMap<String, Integer> convertirArbolEnTH() {
         // Post: el resultado es una table hash con los elementos del árbol,
         //    donde cada elemento aparece una sola vez (el dato asociado a
         //    cada producto es el total de ventas de ese producto)
        HashMap<String, Integer> th = new HashMap<String, Integer>(1000);

        convertirArbolEnTH(th, root);
        return th;
  }

  private void convertirArbolEnTH(HashMap<String, Integer> th,
                                  BinaryTreeNode<Medicina> nodo) {
        if (nodo != null) {
              if (th.containsKey(nodo.element.nombre)) {
                    Integer valViejo = th.get(nodo.element.nombre);
                    valViejo = valViejo + nodo.element.ventas;
                    th.put(nodo.element.nombre, valViejo);
                        // actualizar t. hash
              }
              else { th.put(nodo.element.nombre, nodo.element.ventas);}
                         // añadir nuevo elemento a t. hash
              convertirArbolEnTH(th, nodo.left);
              convertirArbolEnTH(th, nodo.right);
        }

  }

}
```