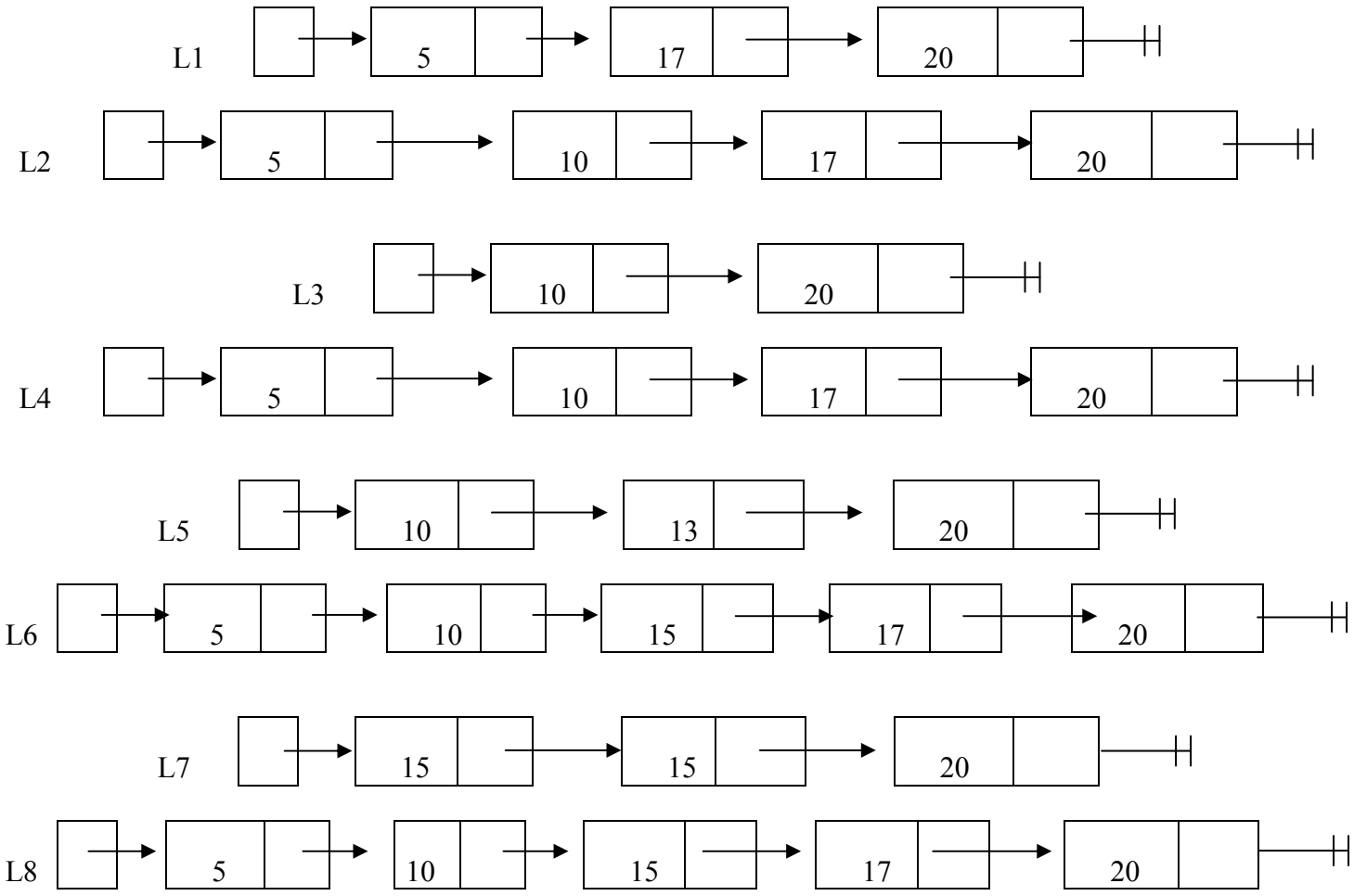
ESTRUCTURAS DE DATOS Y ALGORITMOS

JUNIO 2012

1. Sublista (2,5 puntos)

Dadas 2 listas ordenadas ascendentemente, queremos hacer un subprograma que diga si los elementos de la primera lista están contenidos en la segunda lista, manteniendo el orden.

Por ejemplo, L1 es sublista de L2, y L3 lo es de L4. Sin embargo, L5 no es sublista de L6, ni L7 de L8.



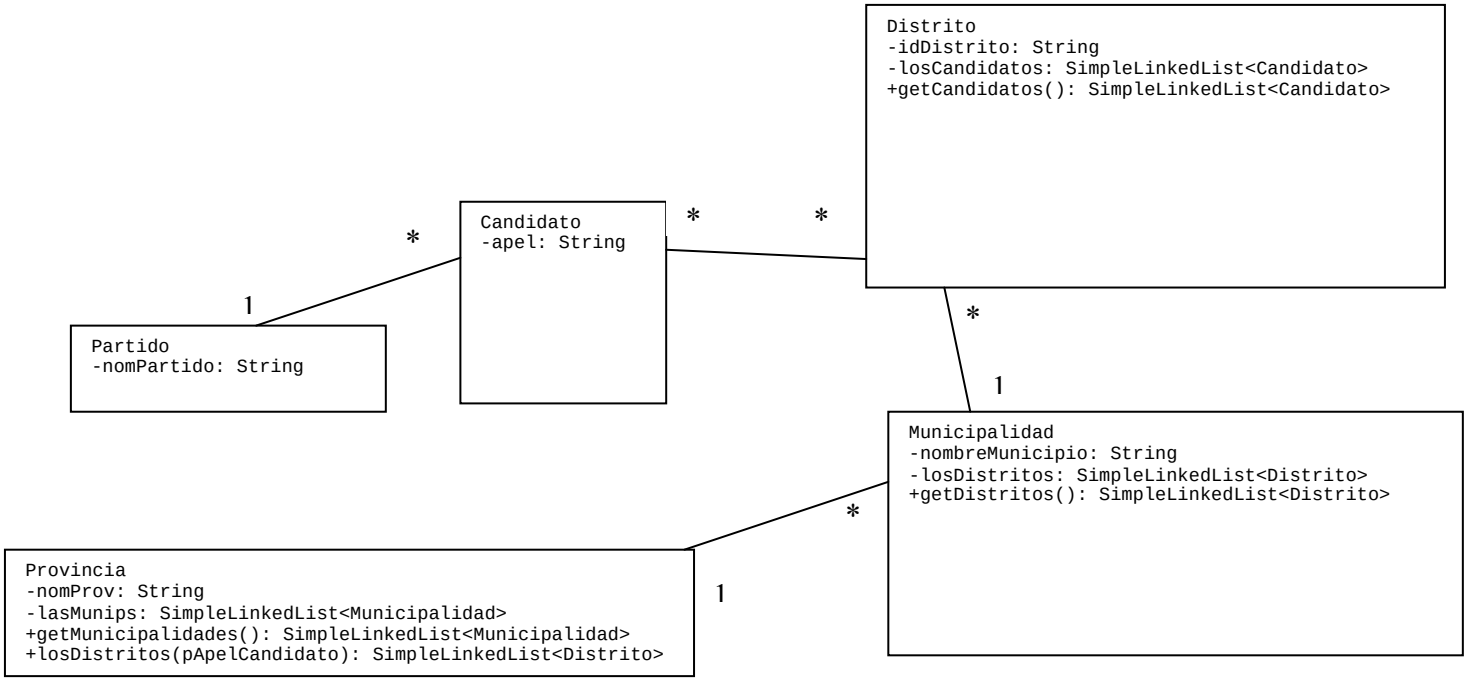
Se deberá calcular el coste del algoritmo resultante, que se valorará (es decir, el coste del algoritmo es importante, cuanto más eficiente, mejor). Cada lista solo puede ser recorrida una sola vez.

Para obtener la solución no se podrán usar los métodos de la clase SimpleLinkedList (excepto los que nosotros implementemos).

---

2. Elecciones (2,5 puntos)

Dado el siguiente diagrama de clases:



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

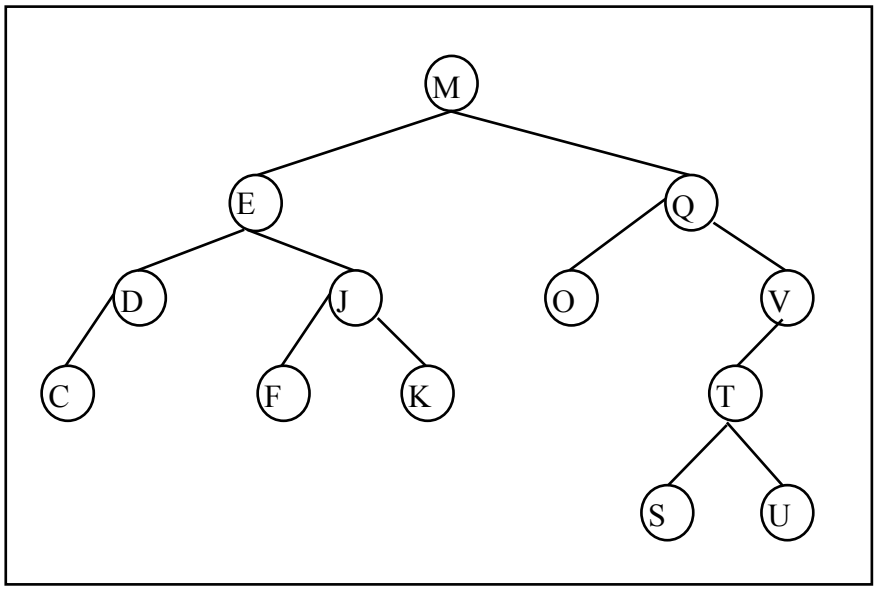
```
public class Eleccion {

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

   public HashMap<Municipio, SimpleLinkedList<Candidato>>

                                 calcularTablaHashMunicipios(Provincia p)
   // Pre: la provincia p existe
   // Post: el resultado es una tabla hash cuya clave es el nombre de los
   //      municipios de esa provincia, y cuyo contenido es la lista de los
   //      candidatos que se presentan a ese municipio
}
```

Por ejemplo, la llamada `calcularTablaHashMunicipios(new Provincia("Gipuzkoa"))` produciría una tabla hash con los siguientes valores:

0		
1	Tolosa	<pepe, jon, ana, alberto>
2		
3	Lasarte	<julen, pedro>
4	Deba	<luis, eneko, amaia, rosa>
5		

3. Cálculo de eficiencia (2,5 puntos)

Considera un Árbol Binario de Búsqueda donde cada nodo contiene un String. Se quiere calcular el número medio de elementos a examinar en las búsquedas, y para ello se calculará empíricamente el número de elementos examinados al realizar una búsqueda para cada elemento de una lista.



Se quiere implementar la siguiente función:

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

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

public class MiArbol extends BinarySearchTree<String> { // Herencia

   public double numMedioDeElementos(SimpleLinkedList<String> l)
   // Pre: los elementos de la lista l están en el árbol
   // Post: el resultado es el número medio de elementos del árbol examinados
   //       en las búsquedas de los elementos de la lista l
}
```

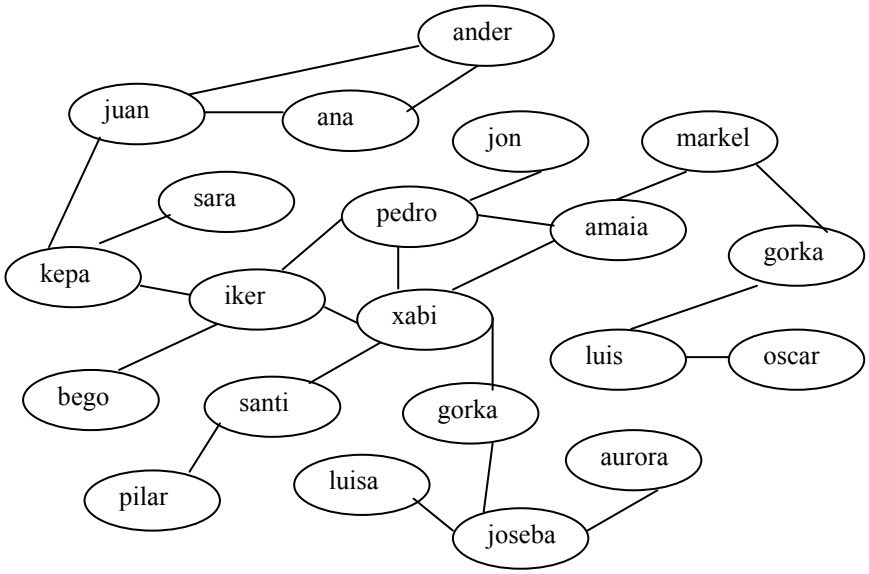
Por ejemplo, si la lista de entrada contuviera <T, M, E, S> el resultado sería:

- 4 elementos examinados para buscar T
- 1 elemento examinado para buscar M
- 2 elementos examinados para buscar E
- 5 elementos examinados para buscar S

Y el resultado final sería 3 = (4 + 1 + 2 + 5) / 4

4. Herencia (2,5 puntos)

Tenemos el siguiente grafo no dirigido, que representa relaciones de parentesco entre personas:



Cuando una persona fallece, se quiere repartir su herencia de la siguiente manera: dado un valor entero N, se repartirá el dinero a partes iguales entre todos los parientes de grado N o menor.

Por ejemplo, la llamada repartir(100000, 2, pedro) devolvería 11111, que es el resultado de dividir 100000 entre las personas que se encuentran a distancia igual o inferior a 2 de Pedro.

En el grafo, a distancia 1 de pedro tenemos a (jon, amaia, xabi, iker) y a distancia 2 se encuentran (markel, gorka, santi, kepa, bego). El resultado será 100.000 / 9, es decir, 11.111.

```
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 int index(T t) {
   // pre: el elemento t se encuentra en el grafo
   // post devuelve el índice del array "vertices" correspondiente a t
}

public class GrafoPersonas extends GraphAL<Persona> {

   public int repartir(int cantidad, int n, Persona p)
   // pre:  n indica el grado de parentesco máximo de las personas a las que
   //       se repartirá la herencia
   // post: el resultado es la cantidad que tocará a cada uno de los
   //      perceptores de la herencia, que se calcula dividiendo la cantidad a
   //      repartir a partes iguales entre las personas que se encuentran a
   //      una distancia menor igual a n de la persona p.

}
```

Lista

```
public interface ISimpleLinkedList<T> {
	public void goFirst();
	public void goNext();
	public void goPrevious();
	public void goLast();
	public boolean hasNext();
	public boolean find(T elem);
	public void insert(T elem)
	public void insertFirst(T elem);
	public void insertLast(T elem);
	public T remove(int index);
		// Pre: al menos hay "index" elementos
	public void remove();
	public void remove(T elem);
	public T get();
	public T get(int index);
		// Pre: al menos hay "index" elementos
	public boolean isEmpty();
}
```

Pila

```
public interface StackADT<T> {
	public void push(T elem)
	public T pop()
	public T peek()
	public boolean isEmpty()
	public int size()
}
```

Ilara / Cola

```
public interface QueueADT<T> {
	public void insert(T elem)
	public T remove()
	public T first()
	public boolean isEmpty()
	public int size()
}
```

Hash taula / Tabla hash

```
public interface Map<K, V> {
	public V put(K key, V value)
	public V get(K key)
	public T remove(K key)
	public boolean containsKey(K key)
	public int size()
}
```