Ejercicio 1. Tipos de datos lineales (2 puntos)

Desentramar una lista consiste en separar, por un lado, los elementos que están en las posiciones pares y, por otro lado, los que están en posiciones impares. Por ejemplo, tras desentramar la lista `[10, 9, 3, 8, 2, 7, 5]` se obtienen como resultado las listas `[10, 3, 2, 5]` y `[9, 8, 7]`.

Partimos de la clase `ListLinkedDouble<T>`, que implementa el TAD lista mediante listas doblemente enlazadas circulares con nodo fantasma, y un contador con el número de elementos. Queremos añadir un nuevo método `void unzip(ListLinkedDouble & dest)`, que desentrama la lista `this`, quedándose esta última con los elementos situados en posiciones pares (suponemos que las posiciones se empiezan a numerar desde el 0), y moviendo los de las posiciones impares al final de la lista `dest` pasada como parámetro. Por ejemplo, dada la lista `xs = [10, 1, 20, 2, 30, 3]` y la lista `zs = []`, tras la llamada `xs.unzip(zs)` la lista `xs` tiene los valores `[10, 20, 30]` y la lista `zs` tiene los valores `[1, 2, 3]`.

Si, en el ejemplo anterior, la lista `zs` no estuviese vacía antes de la llamada a `unzip`, se añadirían los elementos de las posiciones impares de `xs` al *final* de la lista `zs`. Por ejemplo, si inicialmente tuviésemos `zs = [5, 0]`, tras hacer `xs.unzip(zs)` tendríamos `zs = [5, 0, 1, 2, 3]`.

Importante: Para la implementación del método no pueden crearse, directa o indirectamente, nuevos nodos mediante `new` ni borrar nodos mediante `delete`; han de reutilizarse los nodos de la lista de entrada. Tampoco se permite copiar valores de un nodo a otro. El coste de la operación ha de ser lineal con respecto al número de elementos de la lista `this`.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en dos líneas: la primera de ellas denota la lista `this`, y la segunda denota la lista `dest`. Cada una de las listas se representa mediante una secuencia con sus elementos (números enteros distintos de cero). Cada secuencia finaliza con `0`, que no forma parte de la lista.

Salida

Para cada caso de prueba se imprimirán dos líneas: una con el contenido de la lista `this` tras llamar al método `unzip` y otra con el contenido de la lista `dest` tras esa misma llamada. Para imprimir las listas puedes utilizar el método `display()`, o la sobrecarga del operador « que se proporciona para listas.

Entrada de ejemplo

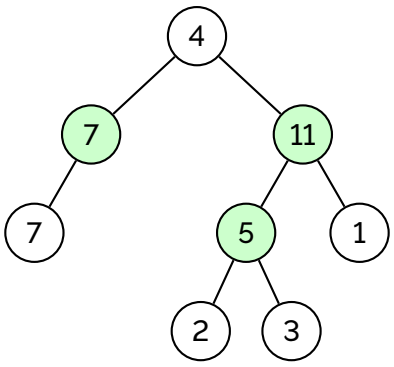
```
2
10 1 20 2 30 3 0
0
7 6 5 4 3 2 1 0
10 20 0
```

Salida de ejemplo

```
[10, 20, 30]
[1, 2, 3]
[7, 5, 3, 1]
[10, 20, 6, 4, 2]
```

Ejercicio 2. Tipos de datos arborescentes (2 puntos)

En un árbol binario de números enteros, decimos que un nodo es *cuadrado* si su valor es la suma de los valores de sus descendientes. Por ejemplo, dado el árbol mostrado en la figura, el nodo interno 7 es cuadrado, ya que la suma de sus descendientes (en este caso, solo hay uno) es 7. El nodo interno 5 también es cuadrado, ya que $5 = 2 + 3$. Por último, el nodo 11 también es cuadrado, ya que $11 = 5 + 2 + 3 + 1$. Los restantes nodos no son cuadrados.



En este ejercicio se pide:

1. Implementar una función `nodo_cuadrado_mayor_prof` con la siguiente cabecera:

   ```
   int nodo_cuadrado_mayor_prof(const BinTree<int> & tree);
   ```

   Esta función debe devolver el valor del nodo cuadrado dentro de `tree` que esté a mayor nivel de profundidad. Para el árbol mostrado anteriormente, el valor devuelto debe ser `5`. En caso de que varios nodos cuadrados se encuentren en un mismo nivel de profundidad, tendrá prioridad el que se encuentre más a la izquierda. Si el árbol no contiene ningún nodo cuadrado, la función debe devolver `-1`.

2. Indicar el coste, en el caso peor, de la función anterior. El coste debe estar expresado en función del número de nodos del árbol de entrada.

Entrada

La entrada comienza con un número que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario mediante la notación vista en clase. El árbol vacío se representa mediante `.` y el árbol no vacío mediante `(iz x dr)`, siendo `x` la raíz, e `iz` y `dr` las representaciones de ambos hijos.

Salida

Para cada árbol se escribirá el valor devuelto por la función `nodo_cuadrado_mayor_prof`.

Entrada de ejemplo

```
3
(((. 7 .) 7 .) 4 (((. 2 .) 5 (. 3 .)) 11 (. 1 .)))
((. 0 .) 4 ((. 2 .) 3 (. 1 .)))
((. 1 .) 3 .)
```

Salida de ejemplo

```
5
0
-1
```

Ejercicio 3. Aplicaciones de tipos abstractos de datos (3 puntos)

La dirección del Hospital Central de Fanfanisflán quiere informatizar su consultorio médico por medio de un sistema que permita realizar al menos las siguientes operaciones:

- `altaMedico(medico)`: da de alta en el sistema a un nuevo médico (identificado por un `string`). Si el médico ya estaba dado de alta lanza una excepción `domain_error` con el mensaje `Medico existente`.
- `pedirConsulta(paciente, medico)`: registra que un paciente (identificado por un `string`) se pone a la espera para ser atendido por un médico. Si el médico no está dado de alta se lanza una excepción `domain_error` con el mensaje `Medico inexistente`. Un paciente puede estar esperando simultáneamente a varios médicos, pero es un error que un paciente pida consulta a un médico al que ya está esperando, lo que se indicará lanzando una excepción `domain_error` con el mensaje `Paciente ya esperando`.
- `siguientePaciente(medico)`: devuelve el paciente a quien le toca el turno, por ser el que más tiempo lleva esperando, para ser atendido por un médico, el cual debe estar dado de alta (si no se lanzará una excepción `domain_error` con el mensaje `Medico inexistente`) y debe tener algún paciente que le haya pedido consulta (si no se lanzará una excepción `domain_error` con el mensaje `Medico sin pacientes`).
- `atenderConsulta(medico)`: elimina de la lista de espera al paciente a quien le toca el turno, por ser el que más tiempo lleva esperando, para ser atendido por un médico, el cual debe estar dado de alta (si no se lanzará una excepción `domain_error` con el mensaje `Medico inexistente`) y debe tener algún paciente que le haya pedido consulta (si no se lanzará una excepción `domain_error` con el mensaje `Medico sin pacientes`).
- `abandonarConsulta(paciente)`: registra el hecho de que un paciente se ha cansado de esperar y abandona la consulta, dejando de estar en las listas de espera de los médicos para los que hubiera pedido consulta y aún no le hubieran atendido; si el paciente no estaba esperando a ningún médico se lanzará una excepción `domain_error` con el mensaje `Paciente inexistente`). La operación devuelve una lista ordenada alfabéticamente con los nombres de los médicos que dejan de tener que atender a este paciente.

La implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD, implementar las operaciones y justificar la complejidad resultante.

Los métodos del TAD no deben mostrar nada por pantalla. El manejo de la entrada y salida de datos se realizará en funciones externas al TAD.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas, en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra `FIN` en una línea indica el final de cada caso.

Los nombres de los médicos y los pacientes son cadenas de caracteres sin espacios en blanco.

Salida

Para cada caso de prueba se escribirán los datos que se piden. Las operaciones que generan salida son:

- `siguientePaciente`, que debe escribir una línea con el nombre del siguiente paciente que será atendido por un médico.
- `abandonarConsulta`, que debe escribir una línea con el mensaje `Medicos abandonados:` seguido de los nombres de los médicos que han perdido a un paciente, en orden alfabético y separados por blancos.

Cada caso termina con una línea con tres guiones (`---`).

Si una operación produce un error, entonces se escribirá una línea con `ERROR:`, seguido del error que devuelve la operación, y no se escribirá nada más para esa operación.

Entrada de ejemplo

```
altaMedico DrSanchez
altaMedico DraFuentes
pedirConsulta Pedro DraFuentes
pedirConsulta Luis DrSanchez
pedirConsulta Pedro DrSanchez
siguientePaciente DrSanchez
abandonarConsulta Pedro
siguientePaciente DraFuentes
FIN
atenderConsulta DrDolittle
altaMedico DrDolittle
altaMedico DrDolittle
pedirConsulta E.Murphy DrDolittle
pedirConsulta E.Murphy DrDolittle
siguientePaciente DrDolittle
FIN
```

Salida de ejemplo

```
Luis
Medicos abandonados: DrSanchez DraFuentes
ERROR: Medico sin pacientes
---
ERROR: Medico inexistente
ERROR: Medico existente
ERROR: Paciente ya esperando
E.Murphy
---
```