Ejercicio 1

2 puntos

Desparizar una lista consiste en eliminar sus elementos pares (dejando únicamente los impares), añadiendo esos pares en el mismo orden al final de otra lista.

En concreto, dada 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,

```
template <typename T> class ListLinkedDouble {
private:
  struct Node {
    T value;
    Node *next;
    Node *prev;
  };

  Node *head;     // Nodo fantasma
  int num_elems;  // Nmero de elementos de la lista
public:
  // ...
};
```

debes añadir un nuevo método `void desparizar(ListLinkedDouble &aux)`, que despariza la lista `this`, quedándose esta última con los elementos impares, y moviendo los nodos con valores pares al final de la lista `aux` pasada como parámetro. Por ejemplo, dada la lista `l = [20, 4, 5, 7, 10, 3]`, y la lista `aux = []`, tras la llamada `l.desparizar(aux)` la lista `l` tiene los valores `[5, 7, 3]` y la lista `aux` los valores `[20, 4, 10]`.

La lista `aux` puede no estar vacía. Si en el ejemplo anterior `aux = [11, 67]`, después de ejecutarse `l.desparizar(aux)`, tendremos `aux = [11, 67, 20, 4, 10]`

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 a `this.num_elems`.

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 es la descripción de la lista a desparizar, mientras que la segunda es la lista `aux`. Cada lista se representa mediante una secuencia con sus elementos (números enteros distintos de cero), finalizando 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 `desparizar` y otra con el contenido de la lista `aux` tras esa misma llamada.

Entrada de ejemplo

```
4
1 2 3 4 5 6 7 8 9 10 0
0
2 1 0
22 0
4 6 8 0
1 0
0
2 0
```

Salida de ejemplo

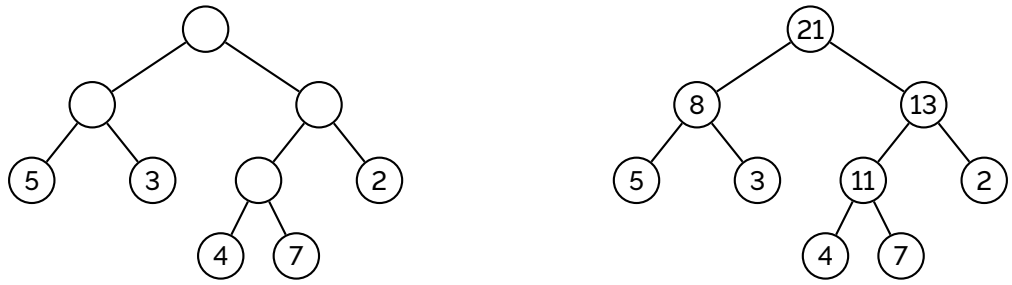
```
1 3 5 7 9
2 4 6 8 10
1
22 2

1 4 6 8

2
```

Ejercicio 2

2 puntos

Dado un árbol binario con números enteros en los nodos, de los cuales solamente nos interesan los valores en las hojas, queremos implementar una función `acumula` que recibe ese árbol y devuelve otro que tiene la misma estructura que el primero pero donde los valores de los nodos internos se han sustituido por la suma de las hojas que tienen como descendientes (es decir, un nodo interno *X* se sustituye por la suma de las hojas del subárbol que tiene a *X* como raíz).

Por ejemplo, si aplicáramos la operación al árbol de la izquierda (donde solamente aparecen los valores de las hojas) obtendríamos el árbol de la derecha.



Se pide:

1. Implementar una función

   ```
   BinTree<int> acumula(const BinTree<int> &arbol);
   ```

   que realice la operación mencionada arriba y devuelva el árbol resultante.

2. Justificar el coste en tiempo de la función `acumula` en función del número de nodos del árbol de entrada.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en la descripción de un árbol binario de enteros, donde los nodos internos tendrán siempre el valor `0` y las hojas podrán tener cualquier valor entre 1 y 500. El árbol vacío se representa con un '`.`' y un árbol no vacío con raíz `R`, hijo izquierdo `i` e hijo derecho `dr` se representa como `(iz R dr)`.

Salida

Para cada caso de prueba, se escribirá una línea con la representación del árbol obtenido tras aplicar la operación `acumula` al árbol de la entrada.

Entrada de ejemplo

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

Salida de ejemplo

```
.
(. 7 (. 7 .))
(((. 3 .) 3 .) 7 (. 4 .))
(((. 5 .) 8 (. 3 .)) 21 (((. 4 .) 11 (. 7 .)) 13 (. 2 .)))
```

Ejercicio 3

3 puntos

Bienvenidos al restaurante *La caracena*, donde el cocinero es tan meticuloso que elabora los pedidos en el mismo orden en el que le llegan. ¡Ningún pedido se cuela! El restaurante dispone de una gran cantidad de mesas, cada una de ellas identificada por un número. Los comensales de las mesas pueden pedir platos a lo largo de la noche. La cocina va recibiendo los pedidos de todas las mesas y va sirviendo los platos, siempre comenzando por el pedido más antiguo que esté pendiente de servir.

Se pide implementar un TAD `Restaurante` que implemente las siguientes operaciones:

- `void nueva_mesa(int num)`

  Registra que un grupo de comensales ha llegado a la mesa `num`. A partir de ese momento, se podrán realizar pedidos desde esa mesa. Si la mesa `num` ya tenía comensales, se lanza una excepción `domain_error` con el mensaje `Mesa ocupada`.

- `void nuevo_pedido(int mesa, const string &plato)`

  Indica a la cocina que la `mesa` cuyo identificador se pasa como primer parámetro ha pedido el `plato` pasado como segundo parámetro. Si la `mesa` no tiene comensales, se lanza una excepción `domain_error` con el mensaje `Mesa vacia`. Ten en cuenta que una mesa puede pedir varias veces el mismo plato a lo largo de la noche.

- `void cancelar_pedido(int mesa, const string &plato)`

  Cancela el último pedido (esto es, el más reciente) del `plato` indicado por parte de la `mesa` pasada como primer parámetro. Solo pueden cancelarse los pedidos que estén en cocina, esto es, pedidos que no hayan sido servidos aún. Si la mesa había pedido varias veces el plato indicado, se cancela solamente el pedido más reciente, pudiéndose cancelar el resto de pedidos mediante sucesivas llamadas a `cancelar_pedido`. Si la `mesa` no tiene comensales, se lanza una excepción `domain_error` con el mensaje `Mesa vacia`. Si la cocina no tiene ningún pedido pendiente del `plato` indicado para esa `mesa`, se lanzará un excepción `domain_error` con el mensaje `Producto no pedido por la mesa`.

- `pair<int, string> servir()`

  Obtiene el pedido más antiguo que esté pendiente de servir, y lo registra como servido en la mesa correspondiente. Devuelve un `pair` con el identificador de la mesa y el plato correspondientes al pedido. Si la cocina no tiene pedidos pendientes de servir, se lanza una excepción `domain_error` con el mensaje `No hay pedidos pendientes`.

- `vector<string> que_falta(int mesa) const`

  Devuelve una lista con aquellos platos pedidos por la `mesa` pasada como parámetro que aún no han sido servidos. La lista resultante debe estar ordenada de manera ascendente según el nombre de los platos, utilizando el orden lexicográfico. Si la mesa tiene varios pedidos pendientes de un mismo plato, el plato solamente debe aparecer una vez en la lista devuelta. Si la mesa no tiene comensales, se lanza una excepción `domain_error` con el mensaje `Mesa vacia`.

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.

Salida

Las operaciones `nueva_mesa`, `nuevo_pedido`, `cancelar_pedido` no imprimen nada, salvo en caso de error. Con respecto al resto de operaciones:

- `servir` imprime una línea con el plato servido y el número de mesa que lo pidió, ambos separados por un espacio.
- `que_falta X` debe imprimir una línea con el mensaje `En la mesa X falta:` (donde `X` es el número de mesa pasado como parámetro). Después deben imprimirse los elementos de la lista devuelta por la operación, cada uno en una línea y precedido por dos espacios.

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

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

Entrada de ejemplo

```
nueva_mesa 1
nuevo_pedido 1 bravas
nueva_mesa 7
nuevo_pedido 7 gazpacho
nuevo_pedido 7 croquetas
servir
servir
que_falta 7
que_falta 1
nuevo_pedido 7 ensalada
nuevo_pedido 7 croquetas
que_falta 7
cancelar_pedido 7 croquetas
que_falta 7
cancelar_pedido 7 croquetas
que_falta 7
servir
servir
FIN
nueva_mesa 1
nueva_mesa 1
nuevo_pedido 2 croquetas
nuevo_pedido 1 croquetas
cancelar_pedido 1 bravas
servir
servir
cancelar_pedido 1 croquetas
FIN
```

Salida de ejemplo

```
bravas 1
gazpacho 7
En la mesa 7 falta:
  croquetas
En la mesa 1 falta:
En la mesa 7 falta:
  croquetas
  ensalada
En la mesa 7 falta:
  croquetas
  ensalada
En la mesa 7 falta:
  ensalada
ensalada 7
ERROR: No hay pedidos pendientes
---
ERROR: Mesa ocupada
ERROR: Mesa vacia
ERROR: Producto no pedido por la mesa
croquetas 1
ERROR: No hay pedidos pendientes
ERROR: Producto no pedido por la mesa
---
```