C03: Partición de una lista

Estructuras de Datos
Facultad de Informática - UCM

Este ejercicio consta de una única entrega, que debe enviarse al problema *DOMjudge* con identificador `C03` antes del **jueves 20 de febrero a las 23:55**.

La entrega consiste en un único fichero `.cpp` que se subirá a *DOMjudge*. Podéis subir tantos intentos como queráis. Se tendrá en cuenta el último intento con el veredicto `CORRECT` que se haya realizado antes de la hora de entrega por parte de alguno de los miembros del grupo.

No olvidéis poner el nombre de los componentes del grupo en cada fichero `.cpp` que entregéis. Solo es necesario que uno de los componentes del grupo realice la entrega.

Las preguntas relativas a costes deben responderse como comentarios en el fichero `.cpp`.

Evaluación: Este ejercicio se puntuará de 0 a 10. Para poder obtener una calificación superior a 0 es necesario obtener un veredicto `CORRECT`.

Este ejercicio consiste en extender la clase `ListLinkedDouble`, que implementa el TAD Lista mediante una lista doblemente enlazada circular, añadiendo un nuevo método:

```
// Lista de elementos de tipo 'int'
class ListLinkedDouble {
public:
  // ...
  void partition(int pivot);  // <- nuevo metodo
private:
  struct Node { int value; Node* next; Node* prev; };
  Node *head;
  int num_elems;
}
```

El método `partition()` recibe un número entero `pivot`, y debe separar, a un lado de la lista, aquellos elementos que sean menores (o iguales) que `pivot` y, al otro lado de la lista, aquellos elementos que sean estrictamente mayores que `pivot`.

Por ejemplo, supongamos la lista `l = [5, 10, 9, 7, 4, 6]`. Tras la llamada a `l.pivot(8)`, la lista debe quedar del siguiente modo: `[5, 7, 4, 6, 10, 9]`. Es decir, al principio están todos los elementos menores o iguales a 8 seguidos de todos los elementos de la lista mayores que 8. Fíjate en el hecho de que se ha preservado el orden relativo entre los elementos que son menores o iguales a 8. Esto es, si el `5` estaba antes a la izquierda del `7` en la lista original, también debe ser así en la lista ordenada. Lo mismo ocurre con aquellos que son mayores que 8. Cuando un algoritmo de partición cumple esta propiedad, se dice que es *estable*.

Procede del siguiente modo:

1. Descarga del Campus Virtual el fichero `.cpp` de plantilla. Este fichero contiene tres métodos sin implementar; dos de ellos privados (`detach()` y `attach()`) y uno público (`partition()`).
2. Implementa el siguiente método privado:

   ```
   void detach(Node *node);
   ```

Este método recibe un nodo perteneciente a la lista enlazada y lo desacopla de ella, pero manteniéndolo en memoria (es decir, tras desacoplarlo no realiza `delete` sobre el mismo). Desacoplar un nodo significa modificar los punteros de sus vecinos para que dejen de apuntar a él. Puedes asumir que `node` *no* es el nodo fantasma de la lista.

3. Implementa el siguiente método privado:

   ```
   void attach(Node *node, Node *position);
   ```

   Este método recibe dos punteros a nodos. El primero de ellos (`node`) no pertenece a la lista. El segundo de ellos (`position`) sí pertenece a la lista. El método `attach()` debe engarzar el nodo `node` en la lista, de modo que acabe situado *antes* del nodo `position`.

   Importante: No puedes crear nuevos nodos mediante `new` en este ejercicio. Tienes que insertar `node` en la lista modificando los punteros de `node` y los de algunos de los nodos ya existentes en la lista.

4. Implementa el método `partition()` descrito al principio del enunciado. Para ello puedes utilizar los métodos `attach()` y `detach()` de los apartados anteriores.

   Importante: De nuevo, aquí no puedes utilizar `new` para crear nuevos nodos. Has de modificar los punteros de los nodos ya existentes.

5. Indica el coste en tiempo, en el caso peor, de los tres métodos implementados: `attach()`, `detach()` y `partition`.

6. Escribe un programa que lea de la entrada varios casos de prueba, cada uno consistente en una lista y un pivote, e imprima por pantalla el resultado de realizar la partición. El formato de la entrada y salida se describen al final de este enunciado. Para asegurarte de que has asignado correctamente los punteros `next` y los punteros `prev` de la cadena, se te pedirá que imprimas el resultado de la lista mediante los métodos `display()` y `display_reverse()`. Este último método imprime los elementos de la lista de derecha a izquierda, empezando por el último y terminado por el primero.

7. Entrega el resultado en el problema de *DOMjudge* con identificador `C03`.

Entrada

La entrada comienza con un `int` que indica el número de casos de prueba que vienen a continuación. Cada caso de prueba consiste en dos lineas. La primera línea contiene una lista, representada mediante una secuencia de números comprendidos entre `1` y `5000`, finalizando con un `0` que no se considera parte de la lista. La segunda línea contiene un número entero, que es el valor del parámetro `pivot` que debe utilizarse para esa lista.

Salida

Para cada caso se escribirán dos líneas con el resultado de la partición. La primera de ellas contiene los elementos de la lista resultado de izquierda a derecha, es decir, desde el primero hasta el último. El formato de esta lista es el mismo que el que produce el método `display()` explicado en teoría, pero ten en cuenta que este último método **no imprime el carácter de fin de línea tras la lista**, por lo que debes añadirlo tú tras llamar a este método. La segunda línea debe contener los elementos de la lista resultado en orden inverso, es decir, desde el último hasta el primero. Para ello puedes utilizar el método `display_reverse()`.

Entrada de ejemplo

```
7
5 10 9 7 4 6 0
8
5 10 9 7 4 6 0
6
5 10 9 7 4 6 0
3
5 10 9 7 4 6 0
20
4 0
4
4 0
6
0
9
```

Salida de ejemplo

```
[5, 7, 4, 6, 10, 9]
[9, 10, 6, 4, 7, 5]
[5, 4, 6, 10, 9, 7]
[7, 9, 10, 6, 4, 5]
[5, 10, 9, 7, 4, 6]
[6, 4, 7, 9, 10, 5]
[5, 10, 9, 7, 4, 6]
[6, 4, 7, 9, 10, 5]
[4]
[4]
[4]
[4]
[]
[]
```