Sumar a un elemento de una lista ordenada

Supongamos una lista de números enteros ordenada de manera ascendente. Este ejercicio consiste en implementar un método `add_to()` que modifique uno de sus elementos sumándole una cantidad determinada, y después mueva el nodo correspondiente de modo que la lista siga estando ordenada. Para ello modificaremos la clase `ListLinkedDouble`, que implementa el TAD Lista mediante listas doblemente enlazadas circulares con nodo fantasma:

```
class ListLinkedDouble {
public:
  // ...
private:
  struct Node {
    int value;
    Node *next;
    Node *prev;
  };
  Node *head;     // Nodo fantasma
  int num_elems;
};
```

Procede del siguiente modo:

1. Añade a la clase `ListLinkedDouble` dos métodos privados estáticos:

   ```
   static void detach(Node *node);
   static void attach(Node *node, Node *before);
   ```

   El método `detach()` recibe un nodo perteneciente a la lista enlazada y lo desacopla de ella, pero manteniéndolo en memoria. Es decir, tras desacoplar el nodo no se realiza `delete` sobre el mismo. Puedes suponer que `node` no es el nodo fantasma de la lista.

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

2. Implementa el siguiente método en la clase `ListLinkedDouble`:

   ```
   void add_to(int index, int m);
   ```

   Suponiendo que los elementos de la lista `this` están ordenados de manera ascendente, y que la lista tiene N elementos, el método `add_to()` recibe un índice `index` tal que $0 \leq$ `index` $< N$. También recibe un número entero `m`. El método suma `m` al valor del nodo situado en la posición `index` y reubica dicho nodo para que la lista siga estando ordenada. Los índices de la lista comienzan a numerarse desde `0`.

   Importante: En la implementación de este método no pueden crearse, directa o indirectamente, nuevos nodos mediante `new` ni eliminarse mediante `delete`. Tampoco se permite copiar valores de un nodo a otro. No obstante, sí puedes (y deberías) utilizar los métodos `attach()` y `detach()` del apartado anterior.

3. Indica el coste en tiempo, en el caso peor, de los métodos `attach()`, `detach()` y `add_to()`.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba ocupa dos líneas. La primera de ellas contiene tres números enteros N, i, m, tales que $0 \leq i < N \leq 5000$ y $-40\,000 \leq m \leq 40\,000$. El número N denota la longitud de la lista, i es un índice dentro de la misma y m es el valor que se sumará al elemento que ocupa la posición i-ésima. La segunda línea contiene N números enteros, comprendidos entre $-50\,000$ y $50\,000$, que son los elementos de la lista sobre la que se aplicará el método `add_to()`.

La entrada finaliza con una línea que contiene `0 0 0`, que no se procesa.

Salida

Para cada caso de prueba se escribirá una línea con el contenido de la lista tras llamar al método `add_to`. Para ello se utiliza la notación vista en clase. Puedes utilizar la sobrecarga del operador << para `ListLinkedDouble`.

Entrada de ejemplo

```
10 4 18
-5 2 4 6 10 14 20 25 26 30
10 3 -10
-5 2 4 6 10 14 20 25 26 30
10 0 0
-5 2 4 6 10 14 20 25 26 30
0 0 0
```

Salida de ejemplo

```
[-5, 2, 4, 6, 14, 20, 25, 26, 28, 30]
[-5, -4, 2, 4, 10, 14, 20, 25, 26, 30]
[-5, 2, 4, 6, 10, 14, 20, 25, 26, 30]
```

Créditos

Autor: Manuel Montenegro.