Mezcla de dos listas ordenadas

Estructuras de Datos
Facultad de Informática - UCM

Partimos de la siguiente clase que define el TAD de las listas de enteros, implementada mediante listas enlazadas simples:

```
class ListLinkedSingle {
public:
  ListLinkedSingle();
  ~ListLinkedSingle();
  ListLinkedSingle(const ListLinkedSingle &other);

  void push_front(const int &elem);
  void push_back(const int &elem);
  void pop_front();
  void pop_back();

  int size() const;
  bool empty() const;
  const int & front() const;
  int & front();
  const int & back() const;
  int & back();
  const int & at(int index) const;
  int & at(int index);
  void display() const;

private:
  // ...
};
```

El presente ejercicio tiene como objetivo *mezclar* los elementos de dos listas ordenadas en una sola, de modo que la lista resultado también esté ordenada. Por ejemplo, a partir de las listas $l_1 = [2, 7, 10, 14]$ y $l_2 = [1, 9, 12, 14, 20]$ queremos obtener la lista $[1, 2, 7, 9, 10, 12, 14, 14, 20]$.

Realizaremos dos versiones de este ejercicio:

- La primera versión utiliza únicamente métodos de la interfaz pública de `ListLinkedSingle`, sin acceder a atributos privados y sin manipular directamente nodos. Esta versión se implementa como una función aparte, ajena a los métodos de la clase

`ListLinkedSingle`, y la lista devuelta es una lista independiente de las listas pasadas como parámetro, en el sentido en que no comparte nodos con ninguna de ellas.

- La segunda versión accede directamente a los nodos de ambas listas y manipula los punteros `next` para que los nodos acaben formando una única lista ordenada de manera ascendente. Esta versión se implementará como un método de la clase `ListLinkedSingle`. De este modo, si `l1` y `l2` son instancias de la clase `ListLinkedSingle`, la llamada `l1.merge(l2)` mezclará ambas listas, de modo que los nodos de `l2` pasarán a formar parte de `l1`, quedando `l2` vacía.

Indica el coste en tiempo, en el caso peor, de cada una de ambas versiones.

Créditos

Adaptación del problema *Mezclar listas enlazadas ordenadas* de Alberto Verdejo.

Solución

Primera versión

```
ListLinkedSingle merge_lists(const ListLinkedSingle &l1, const
    ListLinkedSingle &l2) {
  // Hacemos copias de ambas listas, porque tenemos que extraer elementos
  // desde el inicio de cada una de ellas.
  ListLinkedSingle l1c = l1;
  ListLinkedSingle l2c = l2;

  ListLinkedSingle result;

  // Mientras no se nos haya terminado una de las listas
  while (!l1c.empty() && !l2c.empty()) {
    // Comparamos los elementos iniciales. Si el primer elemento
    // de l1c es menor que el primer elemento de l2c, colocamos el
    // elemento de l1c antes. Si no, colocamos del de l2c
    if (l1c.front() < l2c.front()) {
      result.push_back(l1c.front());
      l1c.pop_front();
    } else {
      result.push_back(l2c.front());
      l2c.pop_front();
    }
  }

  // Salimos del bucle anterior cuando una de las listas haya quedado
  // vacia. Ahora tenemos que anyadir al resultado los restantes elementos
  // de la lista que no haya quedado vacia
  if (!l1c.empty()) {
    while(!l1c.empty()) {
      result.push_back(l1c.front());
      l1c.pop_front();
    }
  } else {
    while(!l2c.empty()) {
      result.push_back(l2c.front());
      l2c.pop_front();
    }
  }
```

```
  return result;
}
```

Coste: Si N es la longitud de la lista `this` y M es la longitud de la lista `other`, el primer bucle realiza como mucho $N+M$ iteraciones. En cada una de ellas realiza una llamada a `push_back`, que tiene coste lineal con respecto a la lista `result`. Como esta última lista puede llegar a tener longitud $N + M$, el coste del primer bucle es $\mathcal{O}((N + M)^2)$. Los restantes bucles tienen el mismo coste en el caso peor. Por tanto, el coste de `merge` en este caso es de $\mathcal{O}((N + M)^2)$.

Segunda versión

```
void ListLinkedSingle::merge(ListLinkedSingle &l2) {
  // prev = Ultimo nodo de la lista ya mezclada
  // cur = Primer nodo de la lista this que aun no ha sido mezclado
  // other = Primer nodo de la lista l2 que aun no ha sido mezclado
  Node *prev, *cur = head, *other = l2.head;

  if (head == nullptr) {
    // Si la lista this es vacia, enlazamos this.head con l2, de modo
    // que todos los nodos de l2 pasan a formar parte de this.
    head = l2.head;
    // La lista l2 se queda sin nodos.
    l2.head = nullptr;
  } else if (l2.head == nullptr) {
    // Si la lista l2 es vacia, no hacemos nada. La lista this se queda
    // con sus nodos (que ya estaban ordenados inicialmente), y la lista
    // l2 se sigue quedando vacia
  } else {
    // Si ambas listas no son vacias, inicializamos prev, cur y other
    if (head->value <= l2.head->value) {
      prev = head;
      cur = head->next;
      other = l2.head;
    } else {
      prev = l2.head;
      cur = head;
      other = l2.head->next;
      head = l2.head;
    }

    // Bucle que va construyendo la lista
```

```
    while (cur != nullptr && other != nullptr) {
      if (cur->value <= other->value) {
        prev->next = cur;
        prev = cur;
        cur = cur->next;
      } else {
        prev->next = other;
        prev = other;
        other = other->next;
      }
    }

    // Cuando salimos del bucle anterior, uno de los dos
    // punteros (cur u other) es nulo. Tenemos que enganchar
    // prev con aquel que NO sea nulo.
    if (cur == nullptr) {
      prev->next = other;
    } else {
      prev->next = cur;
    }
    // La lista l2 se queda sin nodos, porque todos han pasado
    // a formar parte de this
    l2.head = nullptr;
  }
}
```

Coste: Si N es la longitud de la lista `this` y M es la longitud de la lista `other`, el primer bucle realiza como mucho $N + M$ iteraciones. Cada una de ellas tiene coste constante. Por tanto, el coste de `merge` en este caso es de $\mathcal{O}(N + M)$ o, equivalentemente, $\mathcal{O}(\text{máx}\{N, M\})$. Es decir, la lista más larga es la que determina el coste del algoritmo.