# Técnicas de Ordenación Aplicadas a Listas Doblemente Enlazadas.

Alejandro Gómez Quiñones.

Thomas Camilo Vanegas Acevedo.

Universidad Pontificia Bolivariana.

Facultad de Ingeniería.

Estructuras de Datos y Algoritmos.

Jorge Mario Londoño Peláez.

Medellín, Colombia.

Abril 15 de 2024.

## Identificación de los estudiantes.

ID_01: 000287437

ID_02: 000150096

Hash_**Par**:

3c77cdad4919651949cc9d9559afb435001645b72f19aa179805b3a2090a34ca

## Estimación del tiempo requerido para el método split().

```
39      private DoublyLinkedList[] split() {
40          DoublyLinkedList[] lists = new DoublyLinkedList[2];
41          lists[0] = new DoublyLinkedList();
42          lists[1] = new DoublyLinkedList();
43
44          Node slow = head;
45          Node fast = head;
46          while (fast != null && fast.next != null) {
47              lists[0].insert(slow.data);
48              slow = slow.next;
49              fast = fast.next.next;
50          }
51
52          while (slow != null) {
53              lists[1].insert(slow.data);
54              slow = slow.next;
55          }
56          return lists;
57      }
```

El presente método divide la lista doblemente enlazada original y retorna dos listas (sub listas) doblemente enlazadas.

En primer lugar, la inicialización o creación de las dos sub listas enlazadas, es decir (new DoublyLinkedList) toma un tiempo constante, en consecuencia, toma O(1) en notación Big-O.

En segundo lugar, la primer estructura iterativa while, iterará hasta que la referencia (puntero) fast llega al final de la lista (fast == null) o que (fast.next == null). En el análisis del peor de los casos, esto ocurrirá cuando el puntero fast recorra toda la lista. Asimismo, es necesario destacar que el puntero slow avanza una posición por cada iteración mientras que el puntero fast avanza dos posiciones por cada iteración.

Así pues, el primer bucle, suponiendo que n es el número de elementos que existen en la lista, se tendrá una complejidad en notación Big-O de O(n), es decir, lineal.

En tercer lugar, la segunda estructura iterativa while, iterará sobre los elementos restantes de la lista, es decir, n/2 elementos y los inserta en lists[1]. Al igual que el bucle anterior, se simplifica la complejidad en una Big-0 lineal, O(n).

En conclusión realizando la suma de las complejidades obtenidas, se tiene que O(1) + O(n) + O(n) = **O(n)**, es decir todo el método Split tiene una complejidad lineal.

**Estimación del tiempo requerido para el método merge(ListaDoble l1, ListaDoble l2).**

```
59      private static DoublyLinkedList merge(DoublyLinkedList list1, DoublyLinkedList list2) {
60          DoublyLinkedList mergedList = new DoublyLinkedList();
61
62          Node current1 = list1.head;
63          Node current2 = list2.head;
64
65          while (current1 != null && current2 != null) {
66              if (current1.data.compareTo(current2.data) <= 0) {
67                  mergedList.insert(current1.data);
68                  current1 = current1.next;
69              } else {
70                  mergedList.insert(current2.data);
71                  current2 = current2.next;
72              }
73          }
74
75          while (current1 != null) {
76              mergedList.insert(current1.data);
77              current1 = current1.next;
78          }
79
80          while (current2 != null) {
81              mergedList.insert(current2.data);
82              current2 = current2.next;
83          }
84
85          return mergedList;
86      }
```

El presente método merge fusiona (ordenación por fusión) dos listas doblemente enlazadas ya ordenadas y retorna una sola lista doblemente enlazada ordenada.

En primer lugar, la creación de la instancia mergedList toma un tiempo constante, simbolizado en Big-O como O(1).

En segundo lugar, la primera estructura iterativa while, recorre ambas listas al mismo tiempo hasta que al menos uno de las referencias o punteros (current1 o current2) llegue al final. En cada iteración del ciclo implica comparar los elementos actuales de ambas listas y agregar el elemento más pequeño a la mergedList (uso del método compareTo). Por otro lado, la cantidad total de iteraciones es igual al número total de elementos en

ambas listas, en consecuencia, el ciclo while tiene una complejidad en Big-O de O(n) siendo n = suma de las longitudes de ambas listas (longitud de list1 + longitud list2), las cuales se puede expresar en otras constantes, sin embargo, se simplifica usando n como representante de la sumatoria de las dos longitudes.

En tercer lugar, la segunda y tercer estructuras iterativas while, administran los casos en los que una de las listas es mas larga que la otra, es decir, que (longitud de list1 > longitud de list2) o que (longitud de list2 > longitud de list1). Cada uno de estos ciclos iterará la lista doblemente enlazada restante y agregará sus elementos a la mergedList. Cada uno de estos ciclos tiene una complejidad representada en Big-O como O(m), siendo m= longitud de la lista restante.

En cuarto lugar, la inserción en la mergedList dentro de cada ciclo tiene un tiempo constante, ya que insertar un elemento al final de una lista doblemente enlazada toma en Big-O O(1).

En conclusión, la complejidad del método merge() es de **O(n + m)**, donde n es la longitud de la primer lista doblemente enlazada y m la longitud de la segunda lista doblemente enlazada.

**Estimación del tiempo requerido para el método mergesort(ListaDoble list).**

```
88      public static DoublyLinkedList mergesort(DoublyLinkedList list) {
89          if (list.head == list.tail) {
90              return list;
91          }
92
93          DoublyLinkedList[] lists = list.split();
94          DoublyLinkedList firstHalf = mergesort(lists[0]);
95          DoublyLinkedList secondHalf = mergesort(lists[1]);
96
97          return merge(firstHalf, secondHalf);
98      }
99  }
```

En primer lugar, si la lista tiene un solo elemento (list.head == list.tail), se retorna directamente la lista, es decir, obtenemos el análisis del caso base del algoritmo. Su complejidad representada en Big-O es constante, es decir, O(1).

En segundo lugar, el método Split() divide la lista doblemente enlazada en dos mitades, y teniendo en cuenta el análisis previo, esta operación tiene un complejidad de O(n), siendo n= longitud de la lista doblemente enlazada original.

En tercer lugar, el método mergesort se invoca recursivamente para ordenar las dos mitades de la lista. Cada llamada recursiva se realiza en mitades de tamaño cada vez menor (n/2 recursivamente), dividiendo así la lista doblemente enlazada a la mitad en cada paso. Así pues, la profundidad de la recursión es lg(n), donde n es la longitud de la lista doblemente enlazada original y lg(n) es el logaritmo en base 2 de n.

En cuarto lugar, luego de que las invocaciones anteriores ordenaron las sub listas doblemente enlazadas de manera recursiva, se fusionan utilizando el método merge, cuya complejidad analizada anteriormente resulto en  O(n +m), donde n y m son las longitudes de las sub listas (list1 y list2).

En conclusión, se obtiene que la complejidad y la estimación de tiempo del algoritmo de ordenación merge sort en un contexto de listas doblemente enlazadas es de **O(n * lg(n))**, donde n = longitud de la lista doblemente enlazada original y lg(n) = logaritmo en base 2 de n.