PRÁCTICA 1

Estructuras de datos

Grupo 201 GII

María Guzmán Valdezate
Guillermo López de Arechavaleta Zapatero

Contenido

Descripción y Análisis de Métodos

MaxElementCollection ()

Descripción: Inicializa la lista vacía y el comparador a null.

Complejidad Algorítmica: O (1)

rellenarLista ()

Descripción: Rellena la lista con otra lista pasada por parámetro.

Complejidad Algorítmica: O (1)

findMaxElement ()

Descripción: Comprueba que la lista no está vacía y busca el elemento máximo de la colección iterando por la lista sin necesidad de ordenarla previamente. Se inicializa la variable maxElement con el primer elemento de la lista y se compara con los siguientes elementos utilizando un comparador si está definido, o directamente con el método compareTo () si no lo está. Al final devuelve el elemento máximo.

Complejidad Algorítmica: La operación dominante es el bucle for, que recorre n elementos y realiza una comparación por cada uno. Es la línea con una O () mayor, por lo que, por la regla de la suma, la complejidad total del método es O(n).

findMaxElementBySorting ()

Descripción: Comprueba que la lista no esté vacía y encuentra el elemento máximo ordenando primero la lista y luego seleccionando el último elemento de la misma. Si se ha definido un comparador la lista se utiliza usándolo, si no con ordenación natural.

Complejidad Algorítmica: El método Collection.sort() divide la lista en mitades, por lo que es O (log (n)) y para cada división compara y mueve los elementos n veces, por lo que O (n). Por la regla de la multiplicación el total es O (n log(n)).

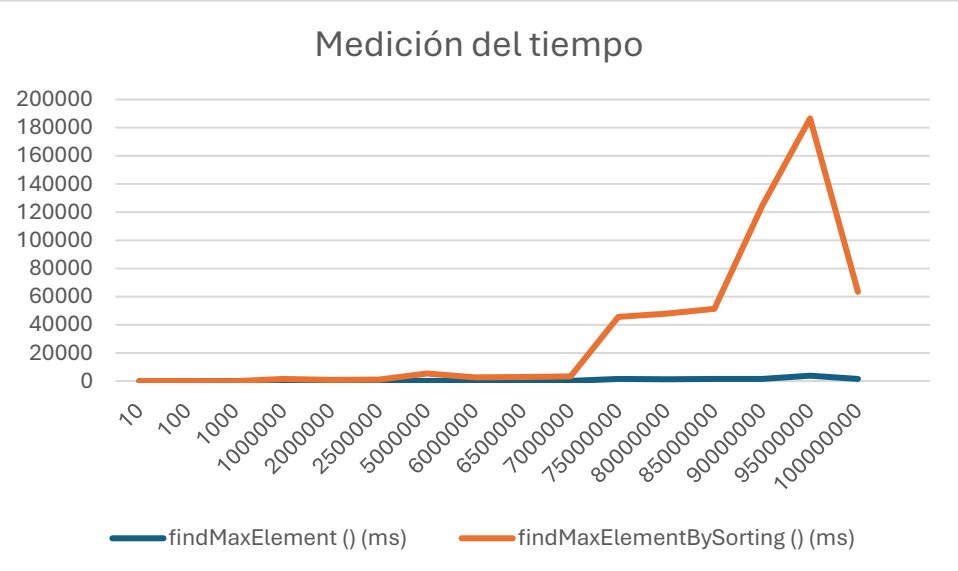
```
@SuppressWarnings({"unchecked", "rawtypes"})
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

Pruebas y gráficas

Para validar las complejidades teóricas, se ha diseñado una clase de prueba que mide los tiempos de ejecución de ambos métodos con listas de diferentes tamaños generadas aleatoriamente mediante la clase GeneradorEnteros.

Se realizan múltiples pruebas con tamaños de lista crecientes y se registran los tiempos de ejecución. A continuación, se presentan los resultados en una tabla y sus respectivas representaciones gráficas.

Tamaño de la lista	findMaxElement () (ms)	findMaxElementBySorting () (ms)
10	0	0
100	0	0
1000	0	1
1000000	60	1423
2000000	39	872
2500000	24	926
5000000	203	5324
6000000	83	2626
6500000	91	2839
7000000	103	3243
75000000	1451	45621
80000000	1202	47829
85000000	1387	51274
90000000	1388	124398
95000000	3797	186616
100000000	1550	63375



Conclusiones

findMaxElement () es más eficiente en términos de tiempo de ejecución para encontrar el máximo, con una complejidad de O(n).

findMaxElementBySorting () es menos eficiente debido a la sobrecarga del ordenamiento, con una complejidad de O (n log n).

Las pruebas empíricas confirman las complejidades teóricas, con tiempos de ejecución que crecen acorde a las previsiones matemáticas.

Este análisis permite concluir que, salvo que se requiera una lista ordenada por otros motivos, el método findMaxElement () es la opción óptima para encontrar el máximo elemento de la colección.

Además, con este trabajo hemos observado dos aspectos fundamentales:

- La importancia de la optimización y el rendimiento del código que desarrollamos, ya que, al trabajar con grandes volúmenes de datos, las diferencias en el tiempo de ejecución entre distintas implementaciones pueden ser significativas.
- La naturaleza imperfecta de la experimentación, ya que los resultados obtenidos no siempre coinciden con los esperados. Un ejemplo de esto es nuestro último experimento, en el que los tiempos registrados fueron menores que en pruebas anteriores.