PRÁCTICA 11

Estructura de datos

Grupo 201 GII

María Guzmán Valdezate

Guillermo López de Arechavaleta Zapatero

Contenido

[Introducción 3](#_Toc199446617)

[Descripción y Análisis de Métodos 4](#_Toc199446618)

[MaxElementCollection 4](#_Toc199446619)

[MaxElementCollectionPQ 5](#_Toc199446620)

[Experimento y Resultados 6](#_Toc199446621)

[Objetivo 6](#_Toc199446622)

[Resultados 6](#_Toc199446623)

[Conclusiones 7](#_Toc199446624)

[Análisis del experimento 7](#_Toc199446625)

## Introducción

Esta práctica tiene como objetivo la implementación y el análisis comparativo de distintos métodos para obtener el mayor elemento en una colección genérica en Java. Partiendo de la clase MaxElementCollection, desarrollada en prácticas anteriores, se han incorporado nuevos métodos que emplean estructuras de datos eficientes como las colas de prioridad (PriorityQueue).

El propósito es aplicar los conocimientos adquiridos sobre algoritmos, estructuras arbóreas, complejidad temporal y espacial, interfaces de colección en Java, y mecanismos de comparación de datos genéricos.

Además, para comprobar cómo era el rendimiento de los códigos hemos modificado la clase Experimento de la Practica 2, haciendo que compruebe todos los nuevos métodos.

## Descripción y Análisis de Métodos

### MaxElementCollection

**findMaxElement()**

**Descripción:**Recorre toda la lista de elementos comparando cada uno con el máximo actual utilizando compareTo().

**Complejidad Algorítmica:**

* O(n) – se recorre la lista de elementos una sola vez.

**findMaxElementBySorting()**

**Descripción:**  
Ordena la lista con Collections.sort() y devuelve el último elemento, que es el mayor.

**Complejidad Algorítmica:**

* O(n log n) – debido al algoritmo de ordenación que utiliza el método sort(), ya que divide el array por la mitad recursivamente hasta que tengas listas de tamaño 1 y luego va combinando las listas en orden, comparando los elementos de dos en dos. Al dividir la lista es O(log n), y luego tiene que recorrer todos los elementos para compararlos, haciendo que sea O(n). Por la regla de la multiplicación queda en O(n log n).

**findMaxElementWithPriorityQueue()**

**Descripción:**Crea una PriorityQueue con orden inverso (max-heap), inserta todos los elementos, y devuelve el primero.

**Complejidad Algorítmica:**

* O(n) – tiene que construir la cola de prioridad, para hacerlo tiene que añadir todos los elementos de la lista de elementos. El proceso de sacar el primer elemento es de O(1) porque solo mira el elemento que está en la cima de la cola.

**Observación:**  
Aunque su complejidad es similar a findMaxElement(), conlleva un coste adicional por la creación y copia de la estructura de datos.

### MaxElementCollectionPQ

**findMaxElement()**

**Descripción:**Itera directamente sobre la cola de prioridad (que no garantiza orden durante la iteración) para encontrar el mayor elemento.

**Complejidad Algorítmica:**

* O(n) – recorre todos los elementos de la cola.

**findMaxElementInPriorityQueue()**

Descripción:  
Accede directamente al mayor elemento usando peek() en una cola max-heap.

**Complejidad Algorítmica:**

* O(1) – el método peek() solo mira el elemento que está en la cima de la cola de prioridad.

**Observación:**Es el método más eficiente si la colección se mantiene en todo momento en una estructura tipo heap.

## Experimento y Resultados

### Objetivo

Evaluar empíricamente la eficiencia de los distintos métodos implementados para obtener el mayor elemento de una colección genérica, en términos de tiempo de ejecución.

### Resultados

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tamaño de la lista** | **Tiempo Sorting (lista):** | **Tiempo Iteración (lista):** | **Tiempo PriorityQueue (lista):** | **Tiempo Iteración (PQ):** | **Tiempo peek()** |
| 10 | 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 1 | 0 | 2 | 0 | 0 |
| 1000000 | 408 | 34 | 284 | 23 | 0 |
| 2000000 | 756 | 35 | 616 | 21 | 0 |
| 2500000 | 958 | 58 | 821 | 11 | 0 |
| 5000000 | 2091 | 122 | 1801 | 28 | 0 |
| 6000000 | 2927 | 161 | 2229 | 35 | 0 |
| 6500000 | 3135 | 183 | 2168 | 28 | 0 |
| 7000000 | 3538 | 203 | 2630 | 42 | 0 |

## Conclusiones

Esta práctica ha permitido comparar diversas aproximaciones para localizar el elemento máximo en colecciones genéricas:

* El uso directo de comparaciones (findMaxElement) es simple y eficiente para listas pequeñas o cuando no se requiere mantener orden.
* La ordenación completa (findMaxElementBySorting) solo es útil si también se necesita una colección ordenada.
* Las colas de prioridad (PriorityQueue) ofrecen una alternativa eficiente si se requiere acceso frecuente al elemento máximo.

**Aprendizajes clave:**

* La estructura de datos elegida influye directamente en la eficiencia del algoritmo.
* La complejidad teórica puede no reflejar el rendimiento real si no se consideran los costes de creación de estructuras temporales.
* El análisis empírico es necesario para complementar y validar el análisis teórico.

### Análisis del experimento

* findMaxElement() fue muy eficiente para colecciones pequeñas, pero crece linealmente con el tamaño.
* findMaxElementBySorting() resultó ser el más costoso debido a la ordenación completa.
* findMaxElementWithPriorityQueue() mejora el acceso, pero paga el coste de construir el heap.
* MaxElementCollectionPQ.findMaxElementInPriorityQueue() fue el más eficiente si la colección ya está en formato heap, pues el acceso es inmediato (O(1)).