Sesión 10

La práctica corresponde al:
Tema 5: Estructuras Arbóreas

Peso en la nota final:
0%

Objetivos

El objetivo principal de esta práctica es proporcionar a los estudiantes una comprensión profunda de los conceptos de complejidad algorítmica y la comparación de algoritmos, aplicándolos en el contexto de las colecciones abstractas en Java. A través de esta práctica, los estudiantes desarrollarán habilidades críticas para analizar y optimizar el rendimiento de los algoritmos, lo cual es esencial en el campo de la informática y la ingeniería de software.

1. **Comparación entre elementos genéricos**: Comprobar cómo se puede establecer un mecanismo para la comparación de datos en una colección, siendo esta definida como genérica, es decir, sin conocer el tipo de datos que se almacenará en la misma por adelantado.
2. **Comprender la Complejidad Algorítmica**: Los estudiantes aprenderán a evaluar la eficiencia de los algoritmos en términos de complejidad temporal y espacial. Esto incluye la capacidad de identificar y calcular la complejidad de diferentes algoritmos, así como entender cómo estos cálculos afectan el rendimiento del software en situaciones del mundo real.
3. **Comparar Algoritmos**: A través de la implementación de dos métodos diferentes para realizar una misma operación, los estudiantes compararán la eficiencia de estos enfoques. Este ejercicio les permitirá ver de primera mano cómo diferentes estrategias pueden tener un impacto significativo en el rendimiento, especialmente a medida que aumenta el tamaño de los datos.

Conceptos Teóricos

El alumno trabajará como conceptos teóricos:

- **Métodos de Búsqueda y Ordenación**
  - Interfaces y Colecciones en Java
    - Interfaz `Collection`, `List`, `Iterator`
    - Interfaces `Comparable` y `Comparator`
- **Complejidad Temporal y Espacial de Algoritmos**:
  - **Complejidad Temporal**: Se refiere al tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada. Es crucial para evaluar la eficiencia de un algoritmo, especialmente cuando se trabaja con grandes volúmenes de datos.
  - **Complejidad Espacial**: Se refiere a la cantidad de memoria que un algoritmo necesita durante su ejecución. Entender la complejidad espacial es importante para optimizar el uso de recursos en sistemas con memoria limitada.
- **Medición del Rendimiento**: Técnicas para medir el tiempo de ejecución y el uso de memoria de un algoritmo. Los estudiantes aprenderán a utilizar herramientas y métodos para realizar estas mediciones de manera precisa.

Trabajo del Alumno

Se pide al alumno revisar el trabajo que realizó para completar la Práctica 01 de la asignatura con los nuevos conocimientos que ha adquirido en el Tema 5, concretamente sobre las Colas de Prioridad.
Se plantea al alumno la solución del mismo problema, empleando una estructura de datos más apropiada, así como la comprobación empírica de una mejor eficiencia y de la complejidad teórica esperada en un montículo.

1. Programa

El programa de esta práctica se divide en varias etapas, cada una con objetivos específicos y tareas detalladas que los estudiantes deben completar.

1. **Implementación de la clase MaxElementCollection**

    a. **Definición de la clase**

Los estudiantes deben definir la clase `MaxElementCollection` que extiende la clase Abstract`Collection<E>`. Esta clase debe contener una lista interna (`List<E>`) para almacenar los elementos.

Se puede emplear directamente la clase que se definió en la Práctica 01 como base para la solución de este ejercicio.

    b. **Método para la localización del Elemento Mayor**

En la Práctica 01 Ya se han implementado dos métodos: `findMaxElement` y `findMaxElementBySorting`. Se pide al alumno el implementar un tercer método:

```
public E findMaxElementWithPriorityQueue() {
    // TODO - Implementar para la Sesion 10.
    /* Para ello:
     *  1. Crear una PriorityQueue con el comparador adecuado (max-heap).
     *  2. Añadir todos los elementos de la lista a la PriorityQueue.
     *  3. Extraer el elemento máximo de la PriorityQueue.
     *  4. Devolver el elemento máximo.
     */
    }
```

Con esto se pretende ver si el ordenar la lista con la `PriorityQueue` es más o menos eficiente que hacerlo con el método de `sort`.

2. Implementación de la clase **MaxElementCollectionPQ**

Como segunda aproximación al problema, se propone el crear una clase similar a la definida anteriormente, pero que, en lugar de almacenar internamente su contenido en una lista, emplee una `PriorityQueue`. De esta forma, también se pueden almacenar los elementos en una estructura lineal, pero ya están preparados para el acceso al mayor elemento. Esto permite ahorrar la copia de los elementos a la PriorityQueue, como sucede en el método descrito arriba.

a. Definición de la clase

Los estudiantes deben definir la clase `MaxElementCollectionPQ` que extiende la clase `AbstractCollection<E>`. Esta clase debe contener una **cola de prioridad** interna (`PriorityQueue<E>`) para almacenar los elementos.

Se puede copiar la clase ya definida en el punto anterior y modificar el contenido para apoyarse en la cola de prioridad internamente.

b. Métodos para la localización del Elemento Mayor

Se pide al alumno comparar otras dos variantes de trabajo. La primera es iterar directamente sobre la cola de prioridad, mientras que la segunda es emplear las operaciones de una cola de prioridad para localizar el mayor elemento incluido en ésta:

- **findMaxElement**

```
  public E findMaxElement() {
      // TODO. Completar este método.
      /* Para ello:
       * 1. Iterar por la cola de elementos.
       * 2. Comparar cada elemento con el maximo encontrado
hasta ahora.
       * 3. Devolver el maximo encontrado.
       */
  }
```

- **findMaxElementInPriorityQueue**

```
 public E findMaxElementInPriorityQueue(){
      // TODO. Completar este método.
      // Para ello: Emplear directamente los métodos de acceso
que permiten localizar este elemento en una cola de prioridad.
(Consultar documentación).
  }
```

2. Informe

Tras realizar la implementación de los métodos, se pide completar un informe en el que se incluya por cada uno de los nuevos métodos:

- descripción textual del funcionamiento del mismo
- razonamiento de su complejidad algorítmica (O grande)

Este análisis teórico se deberá completar con un análisis empírico de funcionamiento, para comprobar si se cumplen los razonamientos teóricos. Para ello, se pide a los alumnos que creen una clase de prueba que permita medir el tiempo en que se completa cada uno de los métodos, según vaya variando el número de elementos que se encuentran almacenados en la colección. Se pueden apoyar en la clase que generaron en la Práctica 01 para incluir las pruebas de los nuevos métodos creados en ésta sesión.

Para ello se facilita la clase de apoyo `GeneradorEnteros`, que permite generar de forma automática, los elementos con los que poder rellenar la colección, especificando simplemente el número de elementos que se desean almacenar.

Se puede ver varios ejemplos de problemas que incluyen test similares en los ejemplos facilitados como ayuda en la asignatura. Se sugiere al alumno emplear ésta como modelo para completar la suya. Es interesante destacar que, en el caso del ejemplo, los métodos a los que se llaman están en la misma clase que realiza la prueba por cuestiones de simplificación. En el caso de la práctica, es más recomendable separar estos métodos en otra clase.

El resultado de este análisis es la obtención de un listado de tiempos de proceso que se puede representar en una o varias gráficas (se pueden copiar fácilmente a un software de hoja de cálculo, por ejemplo) y que permita comprobar de forma visual la tendencia del algoritmo.

Condiciones de Entrega

- La práctica se realizará en grupos de 1 o 2 personas.
- La entrega se hará SOLAMENTE por medio de la plataforma UBUVirtual. Cada miembro del grupo deberá subir su propia copia de la solución. No se admitirán soluciones fuera de plazo o por otros medios de entrega (e-mail, mensajería interna, …).
- La entrega incluirá la información de quienes son los autores de la misma, tanto en los ficheros de código fuente como en los documentos asociados.
- Cada entrega consistirá en un fichero comprimido (formato .zip), con la estructura de nombre
  "`Apellidos1Nombre1_Apellidos2Nombre2`"
- Qué deberá incluir el fichero comprimido:
  - Código fuente de la solución (dentro de la estructura de paquetes necesaria)
  - Documentación en formato Javadoc
  - Informe en formato PDF (en aquellas prácticas que se solicite).
- No hace falta entregar ficheros binarios

Criterios de Evaluación

Dada la naturaleza y objetivo de la práctica, la documentación, el análisis de la complejidad de las aproximaciones y el análisis los resultados obtenidos con distintos conjuntos de datos es el núcleo de la misma. Por este motivo se valorarán todos los criterios enumerados a continuación, pero el primer punto suma el 80% de la valoración total.

- **Documentación, análisis teórico de la complejidad de los algoritmos y el análisis de los resultados numéricos.**
  - Análisis teórico de la complejidad, incluyendo su correspondiente razonamiento.
  - Análisis aplicado de esta complejidad por medio de la medición de tiempos, incluyendo programación del test, recogida de resultados y representación gráfica de los mismos.

    - Comprobación de la correspondencia de ambos análisis.

- Corrección del funcionamiento.

- Corrección del código. Ausencia de *warnings* u otros elementos no deseables como variables no utilizadas, código que no se emplea, etc.

- Documentación de los ficheros fuente. Las explicaciones y razonamientos que se incluyan a modo de comentarios en el código que entrega el alumno. Se solicita que el alumno entregue adicionalmente la documentación generada en formato javadoc, junto con los ficheros de código fuente.