Algoritmos y Estructuras de Datos - DC - UBA

Segundo cuatrimestre de 2025

Trabajo práctico 2: Diseño e implementación de estructuras

Normativa

Revisión de mitad de TP: viernes 07 de Noviembre, en el horario de la clase.
Límite de entrega: domingo 16 de Noviembre a las 23:59.
Defensa del TP: viernes 28 de Noviembre, en el horario de la clase.
Límite de re-entrega: domingo 14 de Diciembre a las 23:59.
Forma de entrega: Subir el archivo `EdR.java` junto con *todos* los archivos de otras clases que hagan dentro de un único ZIP al campus.

Enunciado

Los Iroquai no comen vidrio, saben perfectamente que las bellas ideas platónicas del mundo de la especificación deben ser bajadas a la Tierra (o más precisamente a GPT5, su planeta de origen). Es por eso que algo cambia en su espíritu cuando tienen un objetivo claro por delante, se vuelven pragmáticos y toman decisiones que se ajustan a la imperfecta realidad.

En este TP se espera que realicen un diseño del TAD EdR que especificaron en el TP1. Es importante aclarar que hay algunas diferencias en las operaciones.

Aclaraciones

- Los/as estudiantes se identifican con un número entero no negativo (que denominamos "id") que corresponde al orden en que van ingresando al aula. El primero es el 0.
- Los/as estudiantes van ocupando las posiciones del aula en orden desde el primer asiento hasta el último dejando un lugar libre de por medio en cada fila.
- **Criterio de detección de copia**:
  - Un/a estudiante sospechoso de haberse copiado es alguien que tiene cada respuesta de su examen igual a al menos el 25 % de todos los estudiantes (sin contarlo a él/ella). **Nota**: sólo hay que considerar las respuestas de los ejercicios que resolvió o se copió; las respuestas en blanco no hay que considerarlas.
  - Cada respuesta se verifica por separado.
  - Por ejemplo, supongamos que hay 4 estudiantes y que el examen tiene 2 items, si el estudiante 1 tiene su primera respuesta igual a los estudiantes [2,3] y su segunda respuesta igual a los estudiantes [3,4], entonces es sospechoso de haberse copiado.
  - **Importante**: notar que no se tiene en cuenta la proximidad en el aula entre los estudiantes.
- **Sobre el puntaje del examen**
  - El puntaje es la parte entera del porcentaje de respuestas correctas.
- Las operaciones **corregir** y **chequearCopias** requieren que todos los estudiantes hayan entregado.
- La operación **corregir** requiere que previamente se haya realizado la operación **chequearCopias**.

Operaciones a implementar

Las operaciones se deben implementar respetando las complejidades temporales indicadas. Usaremos las siguientes variables para expresar las complejidades:

- E: cantidad total de estudiantes
- R: cantidad total de respuestas del examen

1. `nuevoEdR(in ladoAula: ℤ, in cantidadEstudiantes: ℤ, in examenCanonico: seq<ℤ>): EdR` $O(E * R)$
   Inicializa el sistema EdR.

2. `copiarse(inout edr: EdR, in estudiante: ℤ)` $O(R + log(E))$
   El/la estudiante se copia del vecino que más respuestas completadas tenga que él/ella no tenga; se copia solamente la primera de esas respuestas.

3. `resolver(inout edr: EdR, in estudiante: ℤ, in nroEjercicio: ℤ, in respuestaEjercicio: ℤ)` $O(log(E))$
   El/la estudiante resuelve un ejercicio.

4. `consultarDarkWeb(inout edr: EdR, in k: ℤ, in examenDW: seq<ℤ>)` $O(k\ (R + log(E))$
   Los/as k estudiantes que tengan el peor puntaje (hasta el momento) reemplazan completamente su examen por el examenDW. **Nota**: en caso de empate en el puntaje, se desempata por menor id de estudiante.

5. `notas(in edr: EdR)`: $seq < \mathbb{Z} >$ $O(E)$
   Devuelve una secuencia de las notas de todos los estudiantes ordenada por id.

6. `entregar(inout edr: EdR, in estudiante: ℤ)` $O(1)$
   El/la estudiante entrega su examen.

7. `chequearCopias(in edr: EdR)`: $seq < \mathbb{Z} >$ $O(E * R)$
   Devuelve la lista de los estudiantes sospechosos de haberse copiado ordenada por id de estudiante.

8. `corregir(in edr: EdR)`: $seq < NotaFinal >$ $O(E * log(E))$
   Devuelve las notas de los exámenes de los estudiantes que no se hayan copiado ordenada por NotaFinal.nota de forma decreciente. En caso de empate, se desempata por mayor NotaFinal.id de estudiante.

La entrega debe incluir *al menos* una clase `Java` que implemente la solución al problema. Recomendamos (y valoraremos) modularizar la solución en varias clases.

IMPORTANTE: No se manden a programar sin pensar antes una solución al problema. La idea es que piensen "en papel" las estructuras de datos que permitirán cumplir con las complejidades especificadas y las consulten con su corrector durante la clase. Una vez que su corrector les dé el OK, pueden comenzar a programar la solución.

Condiciones de entrega y aprobación

Durante la clase del día viernes 07 de Noviembre deberán hablar con su corrector asignado para la **revisión obligatoria** de mitad de TP. En esta, deben mostrarle la estructura de representación propuesta para resolver el problema. Al finalizar esta clase, deben salir con una estructura que funcione para cumplir las complejidades pedidas.

La entrega deberá incluir el archivo `EdR.java`, el cual implementará la solución al problema. Las demás clases diseñadas para la solución se deben incluir en otros archivos `.java`. También los archivos de testeo que desarrollen. Todos estos archivos deben ser subidos al campus antes de la fecha y hora límite de entrega.

Para la aprobación del trabajo práctico, la implementación debe superar todos los tests provistos y, además, debe cumplir con las complejidades temporales especificadas en la sección anterior. Se debe dejar comentado (de forma breve y concisa) la justificación de la complejidad obtenida. Solamente pueden utilizar las estructuras vistas en la teórica hasta ahora (arreglo, arreglo redimensionable, lista enlazada, ABB, AVL, heap), implementadas por ustedes mismos. Pueden usar el código de las estructuras que hicieron para los talleres. **No se puede usar ninguna clase predefinida en la biblioteca estándar de Java, con la excepción de ArrayList, String y StringBuffer**.

Si bien les proveemos una base de tests, **esperamos que agreguen sus propios tests**.

También evaluaremos:

- la claridad del código y de las justificaciones de las complejidades.
- que se respeten los principios de abstracción y encapsulamiento.
- la modularidad del código: crear nuevas clases y funciones auxiliares cuando sea necesario.
- el testeo correcto de sus implementaciones.