

ESTRUCTURA DE DATOS Y ALGORITMOS

# PRÁCTICA FINAL

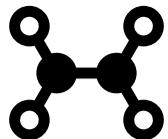
ANDREI GARCÍA CUADRA

REBECA ARIÑO OLIVARES

GRUPO 37



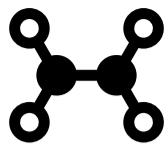
<b>Preámbulo.....</b>	<b>4</b>
<b>Fase 1 .....</b>	<b>5</b>
<b>1. Métodos: .....</b>	<b>5</b>
1.1. main(String[]): void.....	5
1.2. union(StudentsList, StudentsList): StudentsList ↑iManageNetworkList .....	5
1.3. getCampusCity(StudentsList, int): StudentsList ↑iManageNetworkList .....	6
1.4. orderBy(StudentsList, int): StudentsList ↑iManageNetworkList.....	7
1.5. sortedInsert(StudentsList, Student, int): void.....	7
1.6. locateByCity(StudentsList, String): StudentsList ↑iManageNetworkList .....	8
1.7. getStudentsByDateInterval(StudentsList, LocalDate, LocalDate): StudentsList ↑iManageNetworkList.....	9
<b>Fase 2 .....</b>	<b>10</b>
<b>1. Métodos: .....</b>	<b>10</b>
1.1. main(String[]): void.....	10
1.2. copySocialnetwork(Studentstree, studentslist): void↑iManageNetworktree .....	10
1.3. getorderedlist(Studentstree): studentslist↑iManageNetworktree.....	10
1.4. recursiveorder(bstNode, studentslist): void .....	11
1.5. deletebynumberofblocks(Studentstree, int): void↑iManageNetworktree .....	12
1.6. deletebynumberofblocks(bstnode, int): LinkedList<student>↑iManageNetworktree ... 12	
1.7. deletebynumberofblocks(bstnode, int, linkedlist<student>): void↑iManageNetworktree.....	13
1.8. traverseLevels(bstnode): void .....	13
1.8. viewtree(bstnode): List<List<BSTNode>> .....	13
<b>2. Preguntas .....</b>	<b>14</b>
1. Los miembros de la red social también pueden ser buscados por su fecha de inicio de sesión. ¿Es la StudentsTree class una estructura de datos eficiente para este tipo de buscadores?. Razona tu respuesta. Si se considera que no es eficiente, describir una estructura de datos más eficiente para dar soporte a este tipo de buscadores.....	14
<b>Fase 3 .....</b>	<b>15</b>
<b>1. Métodos: .....</b>	<b>15</b>
1.1. main(String[]): void.....	15



1.2. ManageNetworkGraph(String[]): void.....	15
1.3. show(): void .....	15
1.4. addStudent(String): void .....	16
1.4. arefriends(String, String): void .....	16
1.4. getdirectfriends(String): linkedlist<string> .....	17
1.5. suggestedfriends(String): linkedlist<string> .....	17
1.6. getadjacents(int): int[].....	18
1.7. getindex(string): int.....	18
1.8. checkvertex(int): String .....	18
1.9. addedge(int): String .....	19
<b>2. Preguntas .....</b>	<b>20</b>
1. ¿Qué tipo de representación de grafo es la más adecuada para un número tan grande de usuarios posibles? Explica tu respuesta. .....	20
<b>Logger .....</b>	<b>21</b>
<b>Comentarios finales .....</b>	<b>21</b>
<b>Bibliografía .....</b>	<b>22</b>

# PREÁMBULO

- I. A fin de mejorar el trabajo colaborativo en el proyecto, se ha abierto un repositorio de tipo **GIT** en la plataforma BitBucket. En este repositorio se han abierto **tickets** con las cosas pendientes así como subido el **código** por ambas partes, abriendo o cerrando dicho ticket. <https://d3sd1@bitbucket.org/desdiown/practica-final-eda.git>
- II. No se han creado branch especiales más que la rama original master, no obstante, se ha incluido un **README** con información relativa al proyecto, permitiendo el acceso público al mismo desde una fecha anterior a estos días para evitar plagios previos.
- III. La versión de Java utilizada para el proyecto ha sido **Java 12**, la última y más reciente estable.
- IV. La versión de **JUnit** utilizada ha sido la **versión 4.13** junto a **Hamcrest 2.1**.
- V. El **IDE** utilizado ha sido *IntelliJ Idea 2019* por todos los integrantes del proyecto.
- VI. **Los JUnit otorgados no han sido ampliados.** No obstante, **se han realizado pruebas extra en el main** de cada fase para comprobar que otro tipo de errores (si existieran) no se han producido.
- VII. **Todos los métodos contienen líneas de debugging** para clarificar la ejecución del código. Esto se ha llevado a cabo **utilizando una clase Logging**, la cual ha sido creada específicamente para el proyecto y va en el paquete. **Permite el análisis de las funciones por tiempo de ejecución**, visualización de dichos tiempos, y muestra el **output con colores** para clarificar de qué tipo de salida se trata. También **indica el fichero y método** en cuestión para tener un debugging claro y conciso del algoritmo.
- VIII. **Los análisis big-oh se han realizado utilizando el propio código.** Consideramos pues que crear aquí dichas tablas es una redundancia no necesaria, ya que dicha información es aportada al realizar los tests y está muy clarificada. No obstante, **se ha indicado de que tipo es cada algoritmo en esta memoria**.
- IX. Se han añadido **imágenes** para poder **ajustar el texto** a cada página de forma correcta sin que existan demasiados **huecos vacíos**.



# FASE 1

## 1. MÉTODOS:

### 1.1. **MAIN**(**STRING[]**): **VOID**

Análisis big-oh: *No procede.*

Se ha utilizado como un **método auxiliar** para realizar pruebas auxiliares cuando ha sido requerido. También se han probado nuevos casos de uso *in-edition*, los cuales se han modificado, y a fin de mantener la estructura del archivo y no saturarlo, se han eliminado dichas pruebas (ó simplificado) para centrar el fichero en los demás métodos del algoritmo. Las pruebas han sido locales para no modificar los tests de JUnit impuestos en la práctica, y mantener las **buenas prácticas** de programación.

### 1.2. **UNION**(**STUDENTSLIST**, **STUDENTSLIST**): **STUDENTSLIST** ↑ **I MANAGENETWORKLIST**

Análisis big-oh: *Complejidad lineal  $O(n)$ . Mejor caso: listas nulas. Peor caso: Listas largas.*

Tal y como narra la práctica, este método **une dos listas de alumnos**. Para ello, comprueba que las **listas no sean nulas** individualmente (ya que si una lista no es nula, debemos añadir sus alumnos a la lista final, sin que influya la otra). Se ha predicho algo de redundancia en el código, ya que se ejecuta dos veces exactamente el mismo código. Esto se podría haber optimizado haciendo una función auxiliar que recorriese el array, e insertara al final el último elemento de la lista dada. **Se ha hecho con redundancia para evitar mayor complejidad algorítmica** y simplificación del análisis big-oh.

$O(1) = O(\text{yeah})$
$O(\log n) = O(\text{nice})$
$O(n) = O(\text{ok})$
$O(n^2) = O(\text{my})$
$O(2^n) = O(\text{no})$
$O(n!) = O(\text{mg!})$

### 1.3. GETCAMPUSCITY(STUDENTSLIST, INT): STUDENTSLIST ↑IMANAGENETWORKLIST

Análisis big-oh: Complejidad lineal  $O(n)$ . Mejor caso: opción no válida o lista nula. Peor caso: Lista larga y con muchas inserciones.

Este método debe **verificar un parámetro de opción**, y en función de él, operar sobre la lista (también pasada por argumento). Las opciones se han recorrido con un **switch** (y no con else-if) para **clarificar** el código y fomentar la **escalabilidad** ante futuras opciones (si procedieran) fueran añadidas de forma simple y concisa. Las opciones iniciales son la devolución de los alumnos en el **mismo campus** (opción 1) o de **distinto campus** (opción 2). Si no es una opción válida, el método mostrará un texto por pantalla como un *logger* informativo. Posteriormente se verifica que la lista introducida, así como el header de la misma, no sean **nulos**, ya que en dicho caso devolveríamos la lista vacía, reportando el error por consola. Si todo ha ido bien hasta aquí, iteraremos la lista introducida por argumento, utilizando **equalsIgnoreCase** para poder **comparar el tipo enum** (mayúsculas) al **campus** de una manera efectiva y prevenir falsos casos (ya que no influyen las mayúsculas/minúsculas en este caso). Una vez hecho esto, comprueba si se necesita que sean del mismo campus o no (opción del argumento). Si se cumplen las condiciones (ya sea que se necesita el mismo campus y sí son del mismo campus o que no son del mismo campus y se necesitan los de distinto campus) se añaden a la lista final.

Nota: Se ha añadido la **variable auxiliar student** (dentro del bucle) para clarificar la comparación lógica y tener un debugging más preciso, sin sacrificar eficiencia en el código (al programar en Java, confiamos en el garbage collector. Si estuviéramos programando en C, por ejemplo, asumiríamos esto como inadmisible, ya que sí sería una diferencia crucial en el tiempo de ejecución).



#### **1.4. ORDERBY(STUDENTS LIST, INT): STUDENTS LIST**

↑IMANAGENETWORKLIST

Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** opción no válida o lista nula. **Peor caso:** Lista larga.

Este método es similar al anterior en cuanto a la entrada de parámetros. Realizamos las mismas verificaciones, solo que en esta ocasión las opciones son **ordenar la lista de forma ascendente (opción 1)** o de forma **descendente (opción 2)**. En esta ocasión denotamos que no hace falta variable auxiliar, como en el caso anterior, ya que no necesitamos líneas de comparación y está claro que *node.elem* se refiere al alumno.

En este punto, se delegaría el resto de la función al método auxiliar *sortedInsert(StudentsList, Student, int)*.

#### **1.5. SORTEDINSERT(STUDENTS LIST, STUDENT, INT): VOID**

Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** opción no válida o lista nula. **Peor caso:** Lista larga.

Este método no es recursivo y meramente sirve de **método auxiliar de ordenación al insertar**. Toma como argumentos una lista de estudiantes (la cual se utilizará para almacenar los usuarios ordenados), un estudiante nuevo a insertar y la opción (que vuelve a ser verificada) ya que el método es público y estático y puede ser llamado desde cualquier sitio, introduciendo una opción no verificada previamente y cediendo así errores de código.

Nuevamente, verifica que la lista no es nula y su header tampoco, por el mismo motivo que el anteriormente mencionado.

**Si la opción no es válida, detiene la ejecución de dicha ordenación.**

Nuevamente se vuelve a utilizar una variable auxiliar *checkStudent*, para poder legitimizar la comparación lógica. Este método comprueba, recorriendo la lista desde el header hasta el trailer, dónde encajaría el alumno (comprueba contra el anterior). La variable auxiliar *Student aux* se ha sacado fuera para optimizar el código.

## 1.6. LOCATEBYCITY(STUDENTSLIST, STRING): STUDENTSLIST ↑IMAGENetworkList

Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** lista nula. **Peor caso:** Lista larga y con muchas inserciones.

Este método, según narra el enunciado, se encarga de **encontrar** todos los **alumnos que viven en una determinada ciudad**, y toma una lista de estudiantes y el nombre (a modo de string) de aquell@s que viven en dicha ciudad.

Para ello, primero comprueba que ni la lista ni el header sean nulos.

Nuevamente se vuelve a utilizar la **variable auxiliar student** a modo de clarificación del código. Si se cumple que el estudiante no es nulo (recordemos, comprobamos que en cada iteración el nodo no es nulo, pero no comprobamos el alumno, y puede darse el caso de que un alumno sea nulo a mitad de la lista, pero deberíamos seguir recorriéndola) y que el nodo no es nulo, añadimos al final de la lista el estudiante (ya que no influye el orden).



## 1.7. GETSTUDENTSBYDATEINTERVAL(**STUDENTS LIST**, **LOCAL DATE**, **LOCAL DATE**): **STUDENTS LIST** ↑ **I MANAGE NETWORK LIST**

Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** fecha de inicio o fin nulas o lista nula. **Peor caso:** Lista larga y con muchas inserciones debido a casos positivos.

Para este método, gustaría clarificar que se ha utilizado la clase LocalDate. No obstante, **no es compatible con Java 7** y para versiones posteriores se utilizan otro tipo de objetos. Apréciese la intención de habernos **ajustado al enunciado**, pero recordando que **nos hubiera gustado utilizar otro TAD ofrecido por Java**.

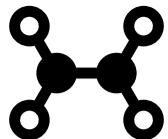
Nuevamente, obtenemos una lista, además de una fecha de inicio y una final. Para este método debemos devolver todos los alumnos entre un intervalo de fechas.

Para comenzar el algoritmo, validamos que la lista no sea nula (ni su header) y que las fechas tampoco son nulas. Si no se cumple alguna de las anteriores condiciones, devolvemos la lista final vacía.



Si las validaciones han sido correctas, iteramos entre los estudiantes de la lista (esta vez sin variable auxiliar, para apreciar la diferencia de legibilidad del código respecto a su uso, y su poca pérdida de eficiencia. Recordemos que **estamos programando en Java y por ende no buscamos la eficiencia** que podríamos obtener en lenguajes de bajo nivel como C. En **Java nos**

**centramos mayormente en legitimizar el código**). Si se cumple que la fecha de inicio marcada es menor que la fecha de registro del usuario y la fecha final es menor que la fecha de registro del usuario, lo añadimos como un resultado positivo para posteriormente devolver la lista con los alumnos que cumplen las condiciones.



## FASE 2

### 1. MÉTODOS:

#### 1.1. **MAIN(STRING[]): VOID**

##### Análisis big-oh: No procede.

Se ha utilizado como un método auxiliar para realizar pruebas auxiliares cuando ha sido requerido. También se han probado nuevos casos de uso *in-edition*, los cuales se han modificado, y a fin de mantener la estructura del archivo y no saturarlo, se han eliminado dichas pruebas (ó simplificado) para centrar el fichero en los demás métodos del algoritmo. Las pruebas han sido locales para no modificar los tests de *JUnit* impuestos en la práctica, y mantener las buenas prácticas de programación.

#### 1.2. **COPYSOCIALNETWORK(STUDENTSTREE, STUDENTSLIST): VOID↑IMAGENETWORKTREE**

##### Análisis big-oh: Complejidad lineal $O(n)$ . Mejor caso: Árbol nulo o lista nula. Peor caso: Lista larga.

Este método es muy simple y eficiente. Su propósito es recorrer una lista (que toma como parámetro) y meter cada alumno en un árbol (que también recibe como parámetro). Obviamente debemos verificar que ni la lista ni el árbol sean nulos, evitando así el *NullPointerException*.

#### 1.3. **GETORDEREDLIST(STUDENTSTREE): STUDENTSLIST↑IMAGENETWORKTREE**

##### Análisis big-oh: Complejidad lineal $O(n)$ . Mejor caso: Árbol nulo. Peor caso: Árbol largo.

Este método ha sido simplificado utilizando un método auxiliar recursivo *recursiveOrder(BSTNode, StudentsList)*. Utilizamos el paso por referencia de *Java* para ceder la *StudentsList* que debemos devolver por parámetro, y el primer nodo (*tree.root*) del árbol binario, previa revisión de que el árbol no es nulo.

#### 1.4. RECURSIVE ORDER(BSTNODE, STUDENTSLIST): VOID

Análisis big-oh: Complejidad cuadrática  $O(n^2)$ . Mejor caso: Nodo, lista o estudiante nulos. Peor caso: Lista larga.

Este **método recursivo** se basa en en caso base de que el nodo no sea nulo. Es un método auxiliar creado para simplificar el problema original de devolver la **lista ordenada sin usar ningún algoritmo de ordenación de listas**. Para ello, recorreremos el árbol (previa comprobación de que la lista, el nodo no son nulos así como el propio alumno). En primer lugar, declaramos una **variable auxiliar inserted**, que se encargará de insertar al final (posición del tamaño de la lista) si no se encontró ninguna posición previa en la que insertar. También hemos de comprobar que la lista no tenga longitud 0, ya que si la tuviera, no se ejecutaría el bucle siguiente y por ende no se añadiría absolutamente nada a la lista. Si la lista tiene longitud 0, añadimos el alumno actual en la posición 0 para operar con el posteriormente (pero que sí se recorra la lista). En este momento debemos recorrer la lista, en la cual comprobamos (**sin variable auxiliar alumno**, para denotar la ilegibilidad del código respecto a su uso) que el alumno actual debe ir antes que el alumno actual y además que el alumno no esté previamente en la lista para prevenir duplicados (tenemos conciencia de que esto se verifica también en el método *insertAt()* que nos cedéis, pero no queremos dejar agujeros en comprobaciones). Si todas estas verificaciones fueron válidas, se inserta el alumno y se notifica a la variable auxiliar *inserte* para que no inserte al final de la lista posteriormente (ya fuera del bucle). Si no hubiera sido añadido, y no estuviera previamente en la lista, se insertaría al final (esto se da cuando el nombre actual no va antes que ninguno).

Una vez finalizada la inserción, proseguimos la recursión en los subárboles derecho e izquierdo, pasándole la *StudentsList* sobre la que estamos operando (haciendo uso, de nuevo, del **paso por referencia de Java**).

## 1.5. DELETEBYNUMBEROFBLOCKS(STUDENTSTREE, INT): VOID↑IMAGENETWORKTREE

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Árbol nulo o bloques menores que 0. **Peor caso:** Lista de borrado de alumnos larga y/o árbol largo.

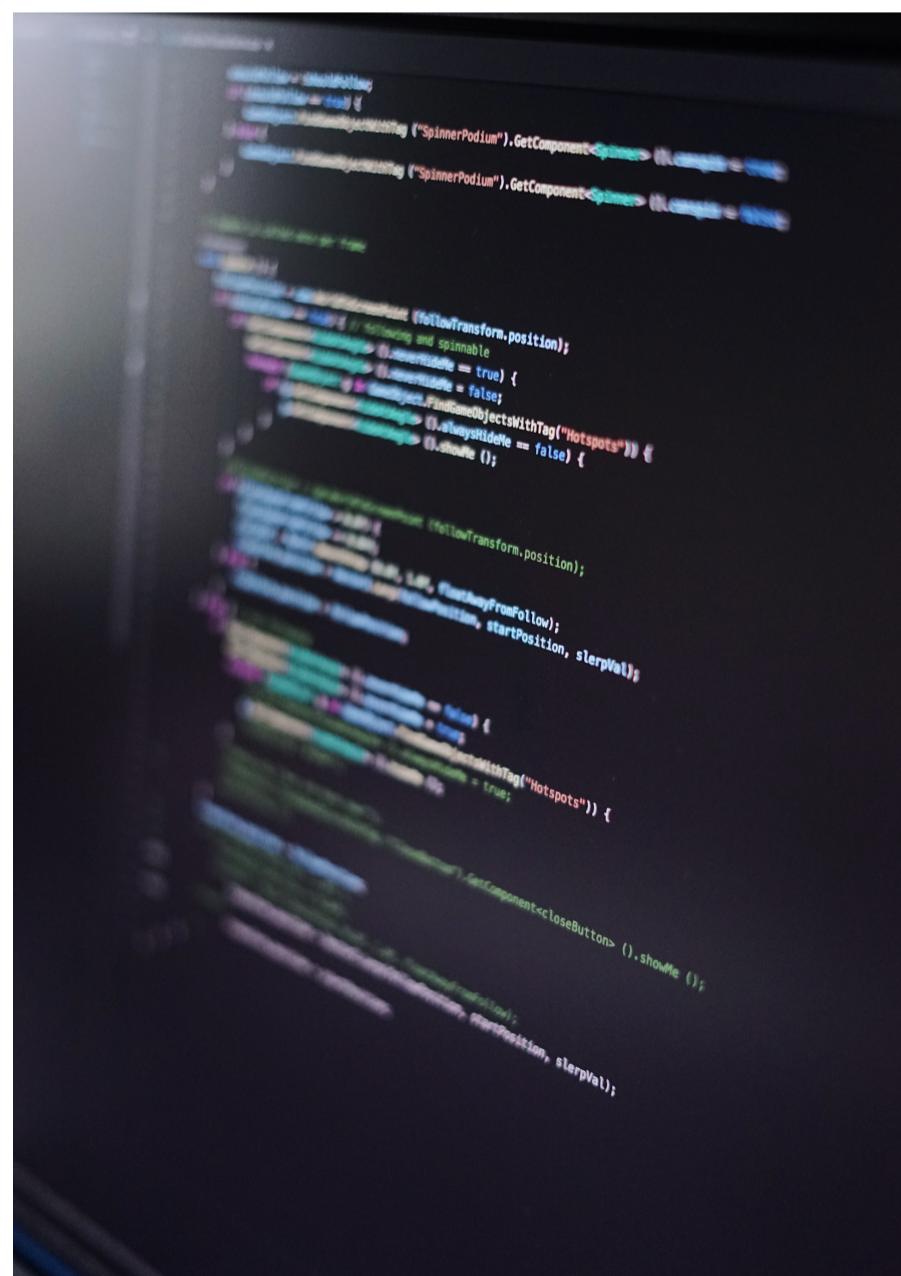
Este método ha sido subdividido en tres a modo de simplificación. Su función final es la de borrar a todos los usuarios que tengan más bloqueos que los estipulados. Este **método (1/3)** se encarga de pasar el nodo raíz del árbol y el número de bloqueos al método *wrapper* (2/3) *deleteByNumberOfBlocks(BSTNODE, INT)*.

Previamente verifica que el árbol no sea nulo y el número de bloqueos no sea negativo (no tiene sentido). Si todo esto está OK, se lo pasa al método *wrapper* (2/3) para simplificar la recursividad. Una vez hecho esto, visualiza el árbol final y elimina del árbol los estudiantes que hayan sido marcados previamente para eliminar.

## 1.6. DELETEBYNUMBEROFBLOCKS(BSTNODE, INT): LINKEDLIST<STUDENT>↑IMAGENETWORKTREE

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Árbol nulo o bloques menores que 0. **Peor caso:** Lista de borrado de alumnos larga y/o árbol largo.

Este método se encarga únicamente de sanitizar los parámetros y crear una lista vacía para devolverla en el método **recursivo**. Este método es el (2/3): *wrapper*. Primero verifica que el nodo no sea nulo y el número de bloqueos no sea negativo. De serlo devuelve una lista vacía, de lo contrario le pasa el árbol para buscar a la función recursiva (3/3) .



## **1.7. DELETEBYNUMBEROFBLOCKS(BSTNODE, INT, LINKEDLIST<STUDENT>): VOID <sup>†</sup>IMANAGENETWORKTREE**

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Árbol y/o lista nulos o bloques menores que 0. **Peor caso:** Lista de borrado de alumnos larga y/o árbol largo.

Este método es el (3/3), es el más importante y de mayor complejidad. Es un algoritmo recursivo que en primer lugar tiene un caso base de excepción. Se encarga de verificar que el nodo no es nulo, que el número de bloques a buscar no es menor que 0 y la lista no es nula. Una vez verificado, procedemos a verificar el caso base real, que consiste la verificación de los bloques del estudiante y los solicitados. Si los solicitados son menores o iguales que los del alumno (razón por la que se permite 0 como parámetro num, si no no se llegaría nunca al caso 0 si sólo se comparase como menor) el alumno se añade a la lista de alumnos para eliminar.

Posteriormente, si los nodos izquierdo y derecho del árbol binario, respectivamente, no son nulos (verificación individual) se usa la recursión para continuar explorando el árbol. Esta verificación se podría omitir pero se ha realizado para evitar falsos positivos en la verificación de la función y que así la legibilidad del algoritmo (*trace*) sea óptima.

## **1.8. TRAVERSELEVELS(BSTNODE): VOID**

Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** Árbol nulo. **Peor caso:** Árbol largo.

Este método es extra y un apoyo para método *viewtree()* y así poder ver correctamente alineados horizontalmente los nodos de un árbol. Primero verifica que root no es nulo, y posteriormente jugando con listas y colas (nativas de Java) itera entre todos los nodos, anadiéndolos a dicha lista. Posteriormente devuelve dicha lista para que sea tratada en el método *viewtree()*.

## **1.8. VIEWTREE(BSTNODE): LIST<LIST<BSTNODE>>**

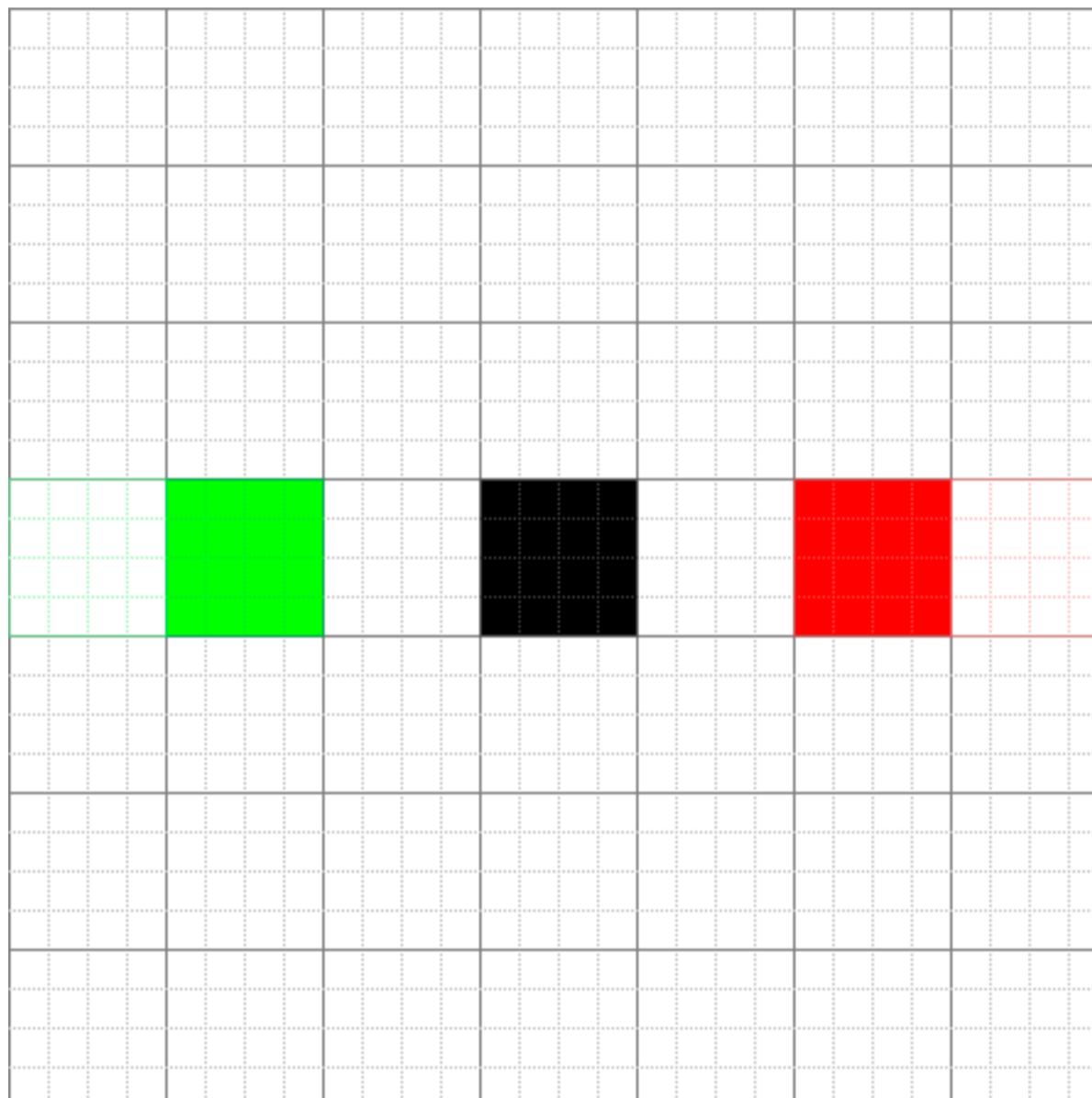
Análisis big-oh: Complejidad lineal  $O(n)$ . **Mejor caso:** Árbol nulo. **Peor caso:** Árbol largo.

Este método ha sido añadido a modo de extra para la visualización por consola del árbol binario. Tiene una complejidad básica y es muy útil. Primero comprueba que el nodo no sea nulo, y posteriormente si se valida correctamente itera entre los niveles apoyándose en el método anteriormente detallado para jugar con los *print* de Java y mostrarlo alineado horizontalmente.

## 2. PREGUNTAS

**1. LOS MIEMBROS DE LA RED SOCIAL TAMBIÉN PUEDEN SER BUSCADOS POR SU FECHA DE INICIO DE SESIÓN. ¿ES LA STUDENTSTREE CLASS UNA ESTRUCTURA DE DATOS EFICIENTE PARA ESTE TIPO DE BUSCADORES?. RAZONA TU RESPUESTA. SI SE CONSIDERA QUE NO ES EFICIENTE, DESCRIBIR UNA ESTRUCTURA DE DATOS MÁS EFICIENTE PARA DAR SOPORTE A ESTE TIPO DE BUSCADORES.**

Escogería un **TAD grafo**, ya que permite una búsqueda eficiente y además listando teniendo en cuenta el orden de relación entre ambos, con una mayor eficiencia, ya que al recorrerlo repulsivamente podríamos encontrar más caminos que con una simple búsqueda de árbol, y por consiguiente, mayor rapidez y eficiencia. El **TAD árbol** no es eficiente para este tipo de búsquedas por valor de un determinado nodo.



# FASE 3

## 1. MÉTODOS:

### 1.1. **MAIN(STRING[]): VOID**

#### Análisis big-oh: No procede.

El método **no se ha utilizado** ya que no estimamos pruebas por esta vía.

### 1.2. **MANAGENETWORKGRAPH(STRING[]): VOID**

#### Análisis big-oh: No procede.

Este método es el constructor principal (y único) de la clase. **Inicia el atributo *students* con una lista de tipo *LinkedList*.** Se ha usado este tipo de lista ya que es doblemente enlazada (colección interna de *Java*).

**Posteriormente, a dicha lista, se añaden los estudiantes** pasados por parámetro.

Tras realizar esta operación, se crea la lista de adyacencias que será utilizada como grafo. Es de tipo *LinkedList* de dos dimensiones, de este modo, se pueden conectar dos nodos de forma dimensional.

En nuestro caso, para unir los nodos, **hemos introducido una doble dimensión**, estando en la primera lista el ID del alumno y en la siguiente dimensión los amigos de dicho alumno (por *ID*).

### 1.3. **SHOW(): VOID**

#### Análisis big-oh: Complejidad lineal $O(n)$ . Mejor caso: Lista vacía. Peor caso: Lista larga.

Este método se encarga de mostrar los **estudiantes actualmente activos** en nuestra lista de almacenamiento de acceso rápido (atributo *students*).

Nótese que no comprobamos si la lista es nula ya que siempre se inicializará desde el constructor y nunca podrá ser modificada externamente a un valor nulo, por ende, no procede su revisión en este ámbito.

#### 1.4. ADDSTUDENT(**STRING**): **VOID**

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

En este caso es un método que tan sólo se encarga de **añadir el estudiante a la lista de almacenamiento**, previa revisión de que el estudiante pasado por parámetro **no es nulo y no está en la lista**, evitando así los **duplicados**.

Denotar que no se crea la lista de adyacencias debido a que la implementación planteada se basa en sistemas pares, es decir, implementar el vértice en una lista de estudiantes y la arista en esta lista de adyacencias. **Al no ser dirigida, se han de añadir dos veces**, uno para cada dirección, en dicha lista de adyacencias. **En nuestra lista de almacenamiento alumnos tan sólo una vez** (un vértice).

#### 1.4. AREFRIENDS(**STRING, STRING**): **VOID**

Análisis big-oh: Complejidad casi lineal ( $\log n$ ). **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Este método se encarga de comprobar si **dos estudiantes son amigos y si no son amigos, hacerles amigos**. Al tener una lista de adyacencias no dirigida, nos da igual comprobar quién va primero, ya que estarán los dos almacenados en dicha lista.

Primero, comprobamos que los estudiantes no son nulos y sí se encuentran en la lista (si no, el método no tiene sentido). Una vez hecho esto, comprobamos que no sean amigos ya (**evitamos duplicidades en la lista de adyacencias**). Esto lo realiza apoyándose en el método `getAdjacents()`, iterando sobre su resultado y verificando que el `estudianteB` no se encuentre en dicha lista.

Si todas las validaciones fueron satisfactorias, añadimos la arista a nuestra lista de adyacencias.

#### **1.4. GETDIRECTFRIENDS(STRING): LINKEDLIST<STRING>**

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Este método se encarga de **recuperar todos los amigos que están conectados directamente al estudiante en cuestión**. Para ello, toma por parámetro el nombre del estudiante.

Para las validaciones, primero hemos de comprobar que el estudiante no sea nulo, y posteriormente el índice del estudiante. Si no se encuentra en la lista de estudiantes, no podemos recuperar su lista de amigos. De lo contrario, nos apoyamos en el método `getAdjacents()` para recuperar todas las adyacencias del vértice (estudiante) actual.

Al devolvernos este método los índices, debemos recuperar el nombre de cada alumno. Para ello hacemos una integración simple, y de paso, comprobamos que el índice es correcto, notificándolo por consola.

Posteriormente retornamos la lista en formato String.

#### **1.5. SUGGESTEDFRIENDS(STRING): LINKEDLIST<STRING>**

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Este método devuelve los amigos de mis amigos.

En primer lugar, comprobamos que el estudiante dado no es nulo y que ya está añadido a la lista. Esto podría omitirse ya que se vuelve a comprobar posteriormente en `getDirectFriends()`, el método en el que se apoya esta función... Pero hemos preferido hacerlo así.

Una vez coge los amigos directos del usuario dado, utilizando el recorrido en profundidad, filtramos los resultados que no sean ni el usuario actual ni los amigos directos.

## 1.6. GETADJACENTS(INT): INT[]

Análisis big-oh: Complejidad casi lineal  $O(n \log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Esta clase se encarga de devolver los vértices conectados con aristas al estudiante dado por parámetro.

Simplemente, recoge el índice del alumno en cuestión y devuelve la sublist (lista ed amigos).

Una vez finalizado el proceso, convierte la lista a array. **Nótese el uso de ArrayList en lugar de un Array estático**, ya que no sabemos con qué operativa de datos trabajaremos en cuanto a los amigos de studentIdx, pero sí sabemos que tenemos que devolver un array estático para optimizar el output. Por ello, se ha usado el método `stream()`, el mapeo a `int`, y se ha convertido a `Array`.

También se podría haber hecho utilizando un bucle, pero nos quedamos con este **método oneliner**, que centra la atención en el resto del método y elimina complejidad al algoritmo.

## 1.7. GETINDEX(STRING): INT

Análisis big-oh: Complejidad lineal  $O(\log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Este método se apoya en el método `indexOf()` de la lista de estudiantes para **devolver el índice de un estudiante** determinado. Si no se encuentra, o la validación del alumno (éste es nulo), devuelve -1.

## 1.8. CHECKVERTEX(INT): STRING

Análisis big-oh: Complejidad lineal  $O(\log n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

En este caso, se comprueba si el vértice dado por parámetro (índice) está contenido en el grafo. De no estar contenido (el índice es negativo o superior al tamaño de la lista dinámica) devuelve null junto a un mensaje, de lo contrario, **el nombre del alumno de dicho vértice**.

## 1.9. ADDEDGE(INT): STRING

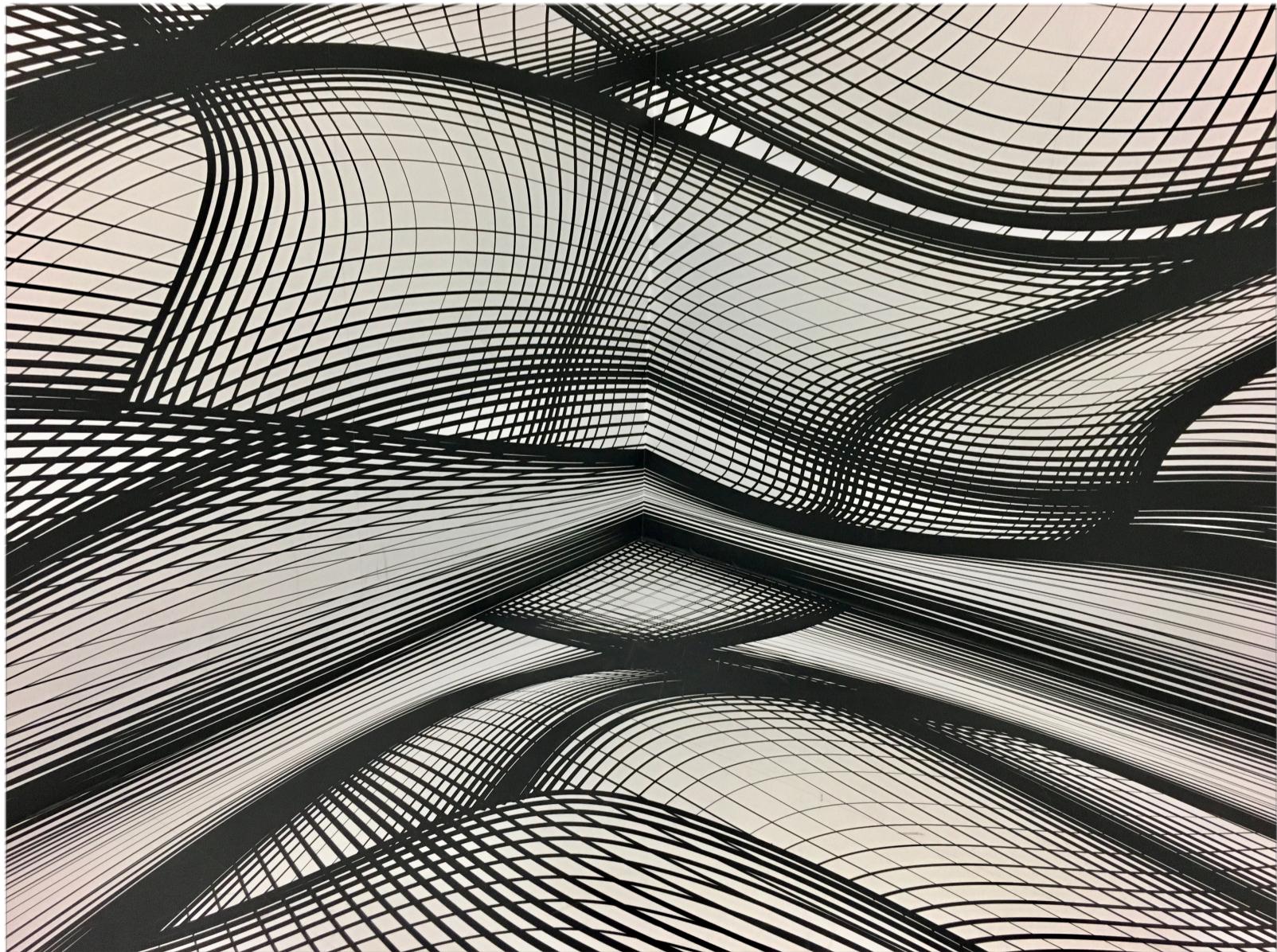
Análisis big-oh: Complejidad lineal  $O(\log_n)$ . **Mejor caso:** Listas vacías. **Peor caso:** Listas largas.

Simplemente, añade un vértice. Para ello, comprueba que ni el vértice A ni el B están fuera del rango del grafo, de estarlo, devolvería un mensaje de precaución.

Si la validación fue correcta, añade una lista de adyacencias, pero si el parámetro **directed** es negativo (el vértice es simétrico o bilateral) **añade lo mismo pero en la relación inversa**.

Para ello, primero **comprueba si existe el vértice dado**. Si no existe, lo crea e iniciativa los amigos vacío (con una *LinkedList*). Una vez nos cercioramos de que existe, añadimos la relación.

Si el grafo es dirigido, hacemos lo mismo pero a la inversa. **Se podría haber usado la misma función recursivamente**, con los índices de los alumnos en inverso y diciendo que no lo queremos dirigido. Lo hemos omitido ya que inducía a confusión, a expensas de un pequeño margen de redundancia en el código.



## 2. PREGUNTAS

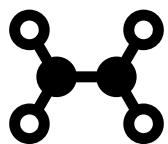
**1. ¿QUÉ TIPO DE REPRESENTACIÓN DE GRAFO ES LA MÁS ADECUADA PARA UN NÚMERO TAN GRANDE DE USUARIOS POSIBLES? EXPLICA TU RESPUESTA.**

Escogería un **grafo ponderado no dirigido acíclico** con una implementación por listas, ya que no tenemos clara la longitud exacta de la cantidad de usuarios, nos interesa relacionarlos por pesos (mayor o menor grado de afinidad), no nos interesa dirigirlo ya que el grado de afinidad en este caso afecta bilateralmente y no nos interesa que un usuario tenga afinidad con si mismo, por lo que quitamos los ciclos del grafo.

Aunque posteriormente se mencione que no afecta el grado de relación entre los alumnos, se ha elegido esta.

Nos gustaría reseñar que hemos rellenado la implementación TAD del grafo, ya que estaba vacía en el ejemplo y la consideramos una implementación completamente necesaria.

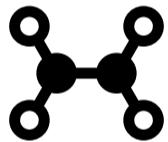




## LOGGER

Se ha creado un logger con el fin de **facilitar el análisis big-oh de los algoritmos**, de la eficiencia, y de el trace algorítmico para poder visualizar el comportamiento y del mismo modo aprender más de esta práctica. Han alterado ínfimamente el código, no ha mejorado su eficiencia directamente, pero sí nos ha permitido mejorar el código que considerábamos menos ágil. Consta de **funciones estáticas** para no tener que crear un nuevo objeto cada vez que se requiera de hacer su uso. Estos métodos son bastante descriptivos por su nombre: **info**, **warning** y **error**. Hay un método privado que es el que realmente se encarga de mandarlo a la consola. Se puede apreciar que dichos comentarios tienen **colores** asociados al tipo de mensaje. Esto se ha logrado utilizando **códigos ANSI** para que sean interpretados por la consola, siendo estos invisibles al output, y tan sólo coloreando las líneas correspondientes.

Para mantener un **formato común** se ha optado por mostrar la hora, minuto y segundos actuales, además de los milisegundos para evaluar los algoritmos; seguido del tipo de log y del mensaje asociado. Tras esto, se muestra el fichero que ha ejecutado dicho log con la línea correspondiente. Esto último se ha logrado utilizando la clase *StackTraceElement* de Java.



## COMENTARIOS FINALES

Nos ha gustado realizar la práctica y ha sido de gran utilidad para afianzar conceptos. No obstante, consideramos que pese a la gran herramienta de *JUnit*, se podrían utilizar tecnologías más modernas para la realización de la corrección, con sugerencias de mejora al código, ya que *JUnit* sólo confirma si está OK o no. A veces ha sido pesado trabajar con ello. Los ficheros tienen un *copyright* determinado para que se evite el plagio durante el transcurso de la práctica ya que el repositorio está abierto.

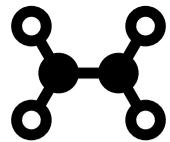
Nos gustaría recibir feedback directo acerca de la práctica. Si es menester, os cedemos abajo nuestros correos ante cualquier comentario, sugerencia, o crítica constructiva.

Esta memoria viene acompañada de una presentación, la cual proyectaremos en clase.

Andrei García Cuadra: [100405803@alumnos.uc3m.es](mailto:100405803@alumnos.uc3m.es)

Rebeca Ariño Olivares: [100363365@alumnos.uc3m.es](mailto:100363365@alumnos.uc3m.es)

*Muchas gracias por su atención.*



# BIBLIOGRAFÍA

- I. Google
- II. Bitbucket
- III. Git
- IV. Java
- V. Baeldung
- VI. GeeksForGeeks
- VII. Universidad Carlos III de Madrid
- VIII. Oracle