Memoria

Práctica de estrategias de programación y estructuras de datos

Curso 2015-2016

Héctor Manuel Luaces Novo | hector@luaces-novo.es

1 TABLA DE CONTENIDO

2 Preguntas previas – Academia simple

2.1 La clase AcademiaS implementa la interfaz AcademiaIF que, a su vez, extiende la interfaz CollectionIF<DoctorIF>. De los Tipos Abstractos de Datos estudiados en la asignatura, ¿cuáles se pueden utilizar para representar las relaciones de dirección de tesis doctorales? ¿Cómo deberían utilizarse esos TAD?

De las clases estudiadas podemos afirmar lo siguiente:

- Los *árboles binarios* no sirven para representar estas relaciones ya que un doctor puede tener **N > 2** estudiantes.
- *Pilas* y *Colas* están descartadas porque exigen un orden que haría que las operaciones de la academia fuesen poco eficientes.
- Los *conjuntos* no nos permitirían implementar de forma eficiente dichas relaciones al solo exponer operaciones de conjuntos como la intersección o unión.

Sabiendo lo anterior, podemos afirmar que de las TAD que se han estudiado en la asignatura, aquellas que podemos usar para representar las relaciones los son los árboles *N-arios* y las *listas*.

A la hora de implementar la opción con listas, una posibilidad es la de tener dos listas independientes, una con todos los doctores de la biblioteca (podría estar ordenada por ID) y otra que almacene objetos auxiliares de tipo *relación* que guarde los estudiantes de un tutor.

Alternativamente podríamos delegar en el propio doctor el conocimiento sobre su supervisor y sus hijos, lo que podría ayudar en las operaciones de recorrido, aunque causaría un incremento en el uso de memoria.

Una estructura arborescente sería implementada de forma en la que cada nodo represente un tutor y los hijos de dicho nodo sean otros árboles que representen a sus estudiantes, profundizando de forma recursiva para seguir representando sus descendientes académicos.

2.2 Entre los TAD considerados en la pregunta anterior, escoja razonadamente uno y explique cómo funcionaría cada uno de los métodos detallados en la interfaz DoctorIF y el método getSupervisor() de la clase DoctorS. ¿Cuál sería su coste asintótico temporal en el caso peor? (Sería un buen ejercicio hacerlo para cada uno de los TAD considerados en 1.1)

El estudiante ha optado por utilizar los árboles *N-arios* para implementar esta versión de la práctica.

Las razones pasan porque los árboles implícitamente ya tienen la relación de "*tutor tiene **N>=0** estudiantes*" -y por ende, de "*estudiante tiene **0<=K<=1** supervisores*"- con lo que no será necesaria implementar ninguna lógica adicional y bastará recorrer el ábol en una dirección u otra para obtener la información que requiere la interfaz de la academia.

Con la excepción de *getSupervisor()*, todos los métodos de la clase *DoctorIF* implementados en *DoctorS* son *frontends* de los métodos homónimos implementados en *AcademiaS*.

Esto se implementó así porque el estudiante creyó que tenía mas cohesión que fuese la academia la que realmente supiese los detalles entre las relaciones, mientras que *DoctorS* se limitase a preguntar a su academia.

El coste asintótico temporal de los métodos solicitados es el siguiente. Para todos los ejemplos se asumirá que ***N*** representa al número de doctores en la academia.

- **getSupervisor()**: es un método *getter* que devuelve una variable almacenada en cada objeto. Su coste es constante ***O(1)*** y es independiente del tamaño del problema.
- **getStudents()**: este método recorre todo el árbol de la academia para encontrar la rama que corresponde al doctor. Al encontrarla devuelve el número de hijos que ésta tiene. Se trata de una función recursiva, pero solo pasa una vez por cada nodo y no hace operaciones extra. Sabiendo esto, podemos confirmar que el coste en caso peor es lineal ***O(N)*** ya que el coste de los casos base y las operaciones no recusivas del método son de coste inferior.
- **getDescendants():** este método utiliza la misma implementación recursiva que *getStudents()*, solo que profundiza en cada doctor un número de generaciones igual a ***K*** para insertar en una lista los descendientes necesarios. Sabiendo que solo se recorrerá cada nodo una única vez y que no hay otras operaciones recursivas o casos base que sean de coste superior, podemos afirmar que el coste del método es lineal ***O(K*N).***
- **getAncestors():** éste método llama a *getSupervisor()* un número de veces igual a las generaciones pasadas como parámetro ***K.*** Esto implica que el coste será lineal **respecto a K** y **N** ***O(K*N)***.
- **getSiblings():** este método consulta el supervisor del doctor, que es una operación *O(1)* y busca la rama del doctor supervisor, devolviendo sus hijos. Para buscar al doctor

supervisor es necesario recorrer los nodos del árbol (de la misma forma que en los métodos anteriores) lo que tiene un coste de **O(N)**.

2.3 Ateniéndonos a este escenario, para cada uno de los cuatro métodos descritos en la interfaz DoctorIF, ¿pueden aparecer repeticiones? Justifique su respuesta.

No pueden existir repeticiones en ninguno de los métodos ya que los métodos recursivos implementados solo recorren el árbol una única vez, añadiendo a una lista que será devuelta como parámetro las raíces de las ramas correspondientes.

Basándonos en que solo pasamos una vez por cada rama y en que no hay dos ramas que puedan compartir un mismo descendiente, confirmamos que no puede haber elementos repetidos en las colecciones devueltas por los métodos.

3 Preguntas previas – Academia completa

3.1 Explique cómo cambian las relaciones de supervisión en este segundo escenario. A la hora de representar el problema, ¿puede aplicarse directamente el mismo TAD utilizado en el primer escenario? ¿Por qué?

En este caso un doctor puede tener varios supervisores, en lugar de uno solo.

Al ser así, puede darse el caso de que haya doctores de una generación que tengan a otros de generaciones mucho mayores bajo su supervisión, lo que implica que varios alumnos de generaciones completamente distintas puedan compartir uno o más supervisores.

Esto altera significativamente el escenario inicial, en el que una estructura jerárquica simple representaba las relaciones entre todos los doctores.

En este caso el árbol *K-Ario* de la primera práctica no podría ser usado ya que los procesos de recorrido -al tener varios hijos de generaciones completamente heterogéneas- podrían devolver resultados inesperados o no ser capaces de terminar, además de que en éste escenario si pueden darse duplicados y la *AcademiaS* no tenía ningún control sobre los mismos.

3.2 Razone, sin realizar ningún tipo de implementación, cómo funcionarían en este escenario los métodos de la interfaz DoctorIF y el método getSupervisors() de la clase DoctorC suponiendo que la clase AcademiaC se implemente utilizando una lista de objetos de la clase DoctorC

Si únicamente se implementase una lista de objetos *DoctorC* y se asume que no hay otra estructura auxiliar para implementar las relaciones, sería necesario que cada objeto doctor conociese tanto a sus estudiantes como a sus supervisores.

De esta forma podría recorrerse la lista de doctores para obtener la información necesaria en los diferentes métodos, si bien sería necesario recorrerla varias veces.

Otra alternativa sería tener una segunda lista que almacenase objetos auxiliares de tipo *Tesis* que guardasen la relación entre un grupo de supervisores y un doctor estudiante.

3.3 Basándose en la respuesta a la pregunta anterior, supongamos que se añade a la clase DoctorC un nuevo atributo supervisors con la colección de los doctores que dirigieron su tesis doctoral (y, en consecuencia, el método getSupervisors() se convierte en un getter de dicho atributo). ¿Cómo afectaría esto al espacio necesario para almacenar el Árbol Genealógico de la Academia en memoria? ¿Y cómo afectaría a la programación y al tiempo de ejecución de los métodos?

La memoria empleada por la academia se elevaría sensiblemente ya que cada doctor individual podría tener como supervisores un número arbitrario de otros doctores, cuyas referencias pasarían a estar almacenadas no en una única lista de la academia, si no también en cada nodo individual.

Sin embargo, haría que el tiempo de ejecución de las operaciones de *getSupervisors()* y *getStudents()* pase a ser lineal, ya que únicamente se necesitaría devolver una variable almacenada en el propio objeto.

La programación de dichos métodos sería trivial y los métodos recursivos *getAncestors()* y *getDescendants()* podrían aprovecharse de ellos para iterar sobre las distintas generaciones.

3.4 Supongamos que la lista de objetos de la clase DoctorC referida en la pregunta 2.2 se encuentra siempre ordenada según el atributo id. ¿Cómo se podría aprovechar en la implementación de los métodos para mejorar el coste temporal de las operaciones de consulta? ¿Cómo afectaría a la operación de modificación?

Las operaciones de consulta podrían ejecutar una búsqueda binaria sobre la lista de doctores, consiguiendo así tener una eficiencia de búsqueda de ***O(log n)*** en contrapartida de una eficiencia lineal.

El coste de operación de modificación se vería ligeramente perjudicado ya que cada inserción implicaría una recolocación (**no** reordenación) de los elementos de la lista para garantizar que éstos estuviesen ordenados.

3.5 Supongamos que levantamos la restricción de que en una Academia sólo exista un Doctor sin supervisores dentro de la Academia y permitimos que haya varios Doctores en esa situación. ¿Habría que hacer cambios en la implementación? ¿Cuáles y por qué?

En ese escenario hipotético el único cambio que sería necesario realizar es el de adaptar el método *getFounder()* para que devolviese una colección de doctores y no un único doctor, ya que ahora podría haber más de un doctor fundador.

El resto de métodos no sería necesario que se retocasen, ya que en este modelo de academia cada doctor puede contener un número arbitrario de supervisores o estudiantes sin que esto sea relevante para ninguna operación, ya que todas las operaciones de recorrido podrían trabajar con cualquier número de descendientes o antecesores, siendo incluso 0.

3.6 Ateniéndonos a este escenario, para cada uno de los cuatro métodos descritos en la interfaz DoctorIF y para el método getSupervisors() de la clase DoctorC, ¿pueden aparecer repeticiones? Justifique su respuesta.

- **getSupervisors() y getStudents(): no** pueden aparecer duplicados ya que la relación de *supervisión* implica únicamente que una lista de doctores va a tutelar al alumno y la propia interfaz *AcademiaIF* asevera mediante una precondición (etiqueta *@pre*) en el método *addSupervision* que no existiría una relación previa entre ambos doctores.
- **getAncestors() y getDescendants(): si** pueden aparecer duplicados, ya que al recorrer varias generaciones de doctores con un número arbitrario de estudiantes y/o

supervisores puede darse el caso de que dos nodos tengan uno o más descendientes comunes.

4 Análisis del coste empírico de la práctica

A continuación, mostraremos los gráficos que ilustrarán el crecimiento de los diferentes métodos de los doctores de la aplicación.

Cabe destacar lo siguiente respecto a la medición realizada:

- Se han utilizado las herramientas y datos proporcionados por el equipo docente para llevar a cabo las pruebas.
- La medición se lleva a cabo en milisegundos, si bien hay que tener en cuenta que el resultado de la medición es la suma de varias iteraciones sobre cada método individual.
- Todas las pruebas de crecimiento no constante se han realizado con **100.000** iteraciones para garantizar un coste calculado preciso.
- Algunas pruebas de crecimiento no constante se han realizado con **1000000** iteraciones para poder medir el tiempo. Las pruebas se han señalado en cada caso.

Por último, todas las pruebas se han hecho con árboles *K-arios* de una profundidad igual a ***1<=P<=5*** y un número de hijos igual a ***1<=H<=5***. Dichos árboles eran los proporcionados por el equipo docente y en las gráficas se ilustrarán los costes de los métodos para cada árbol.

5 Gráficas para el escenario simplificado

5.1 getDescendants()

La

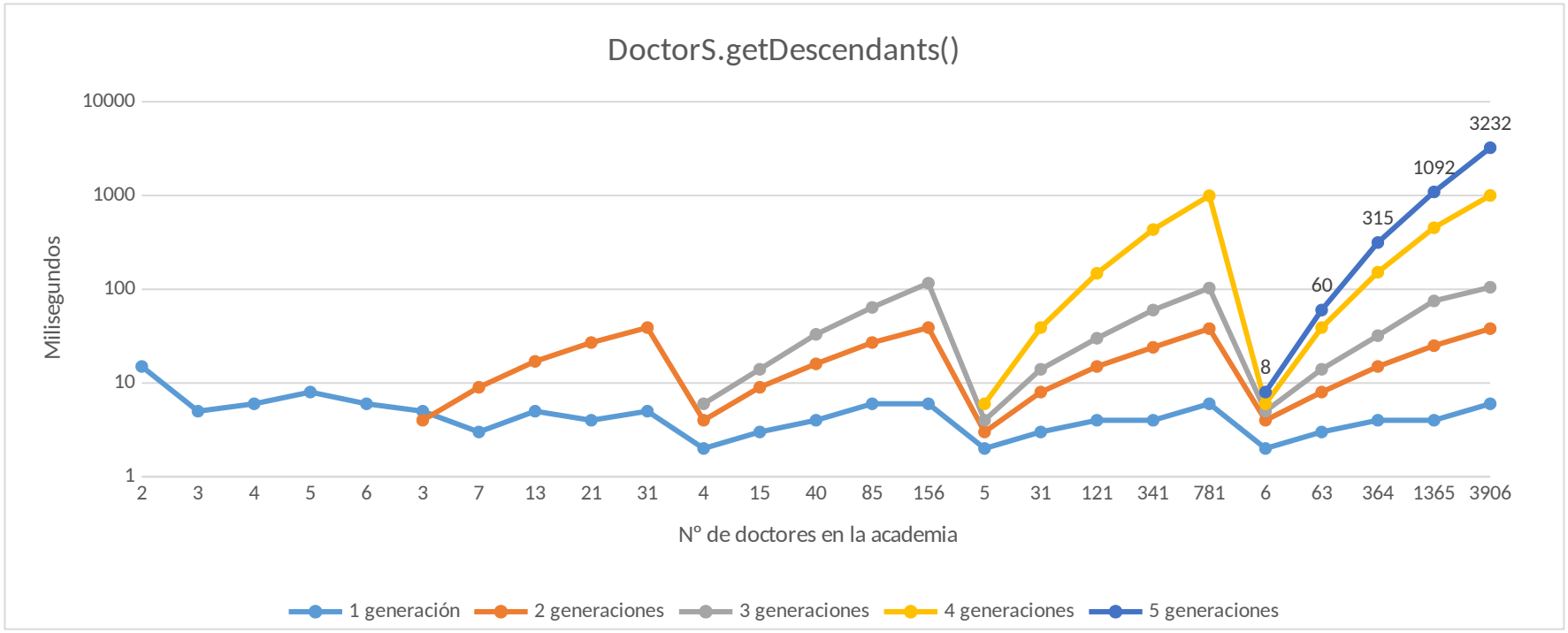


Ilustración 1

gráfica muestra un acusado crecimiento lineal que se agudiza a medida que se recorren más generaciones. Cabe destacar que cada uno de los tramos muestra una tendencia parecida para cada generación y que cada tramo **siempre** recorre toda la profundidad del árbol.

La gráfica coincide con el análisis que se hizo anteriormente que indicaba que el crecimiento del método sería lineal.

5.2 getSupervisor

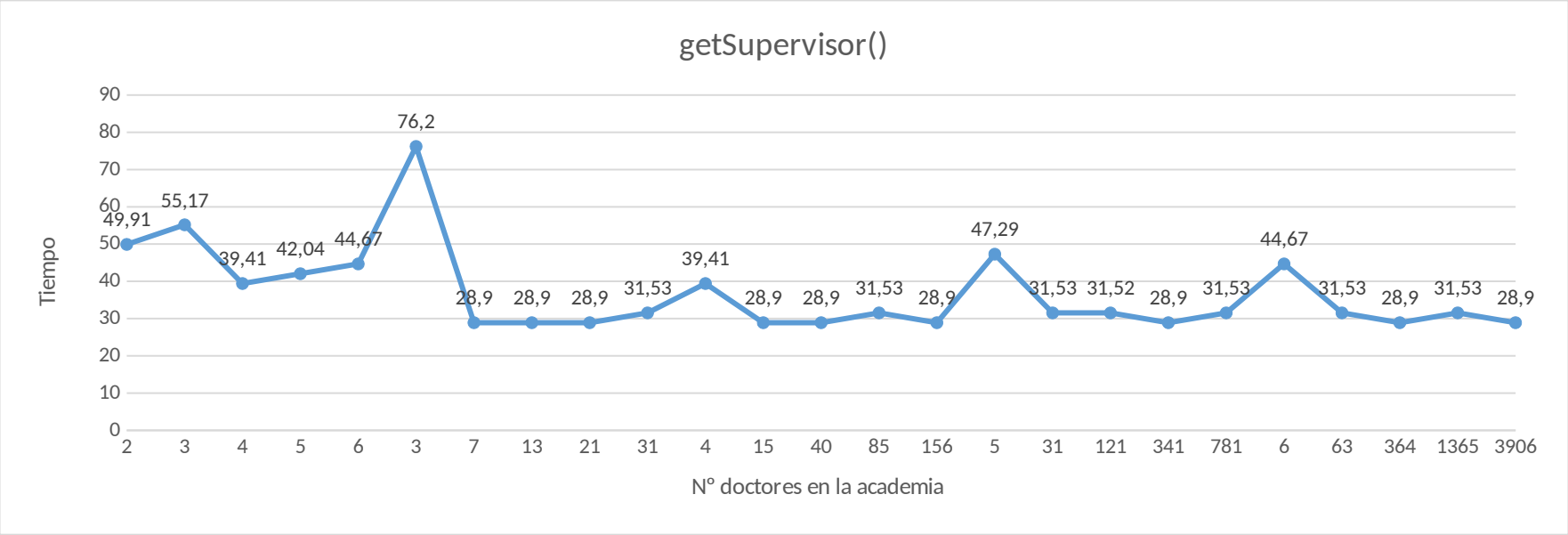


Ilustración 2

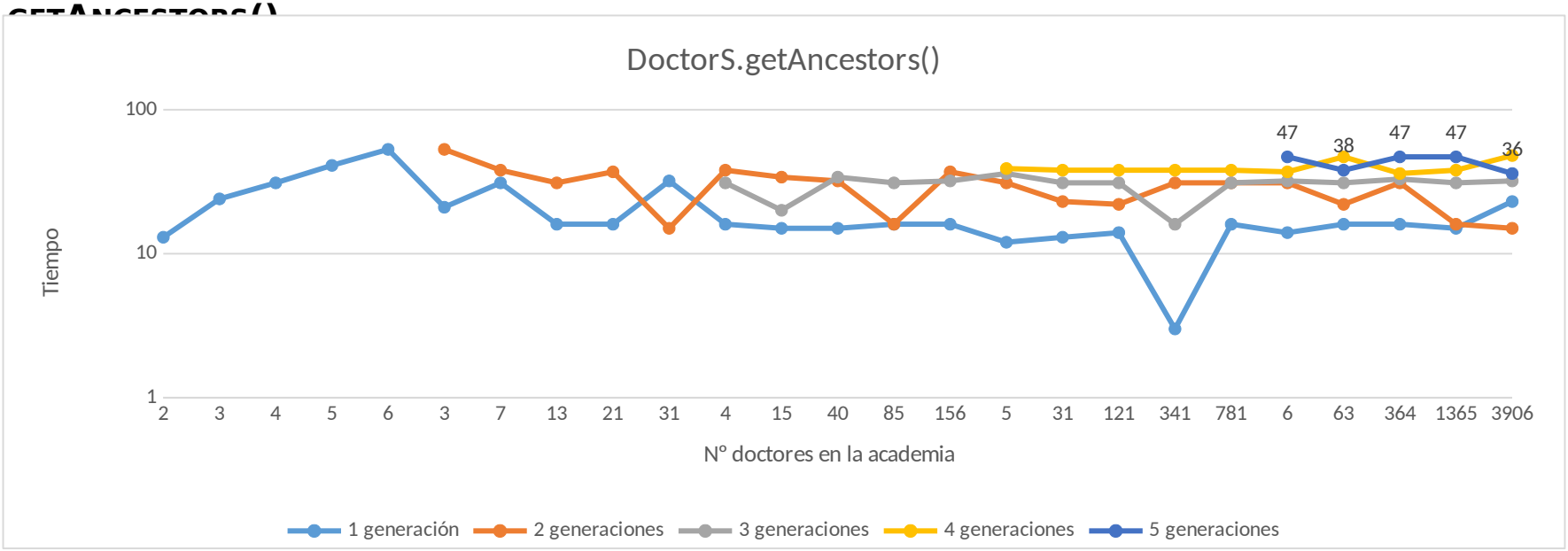
getSupervisor() tiene un crecimiento constante respecto al número de doctores de la academia ***N***.

Esto es así debido a que cada doctor guarda una referencia a su supervisor, lo que convierte la operación de obtener su supervisor en una operación trivial a costa de sacrificar memoria para guardar la referencia.

Para medir los tiempos de éste método se han hecho 1.000.000 iteraciones.

5.3 getAncestors()

La



gráfica muestra un crecimiento constante respecto al número de doctores ***N*** en la academia.

Ilustración 3

Esto es debido a que *DoctorS* guarda en una variable a su supervisor, lo que implica que para obtener las ***K*** generaciones de un doctor basta con repetir ***K*** veces una operación de coste lineal respecto a ***N***.

Esto coincide con la premisa del alumno, que indicaba que el coste era lineal **respecto a K.**

5.4

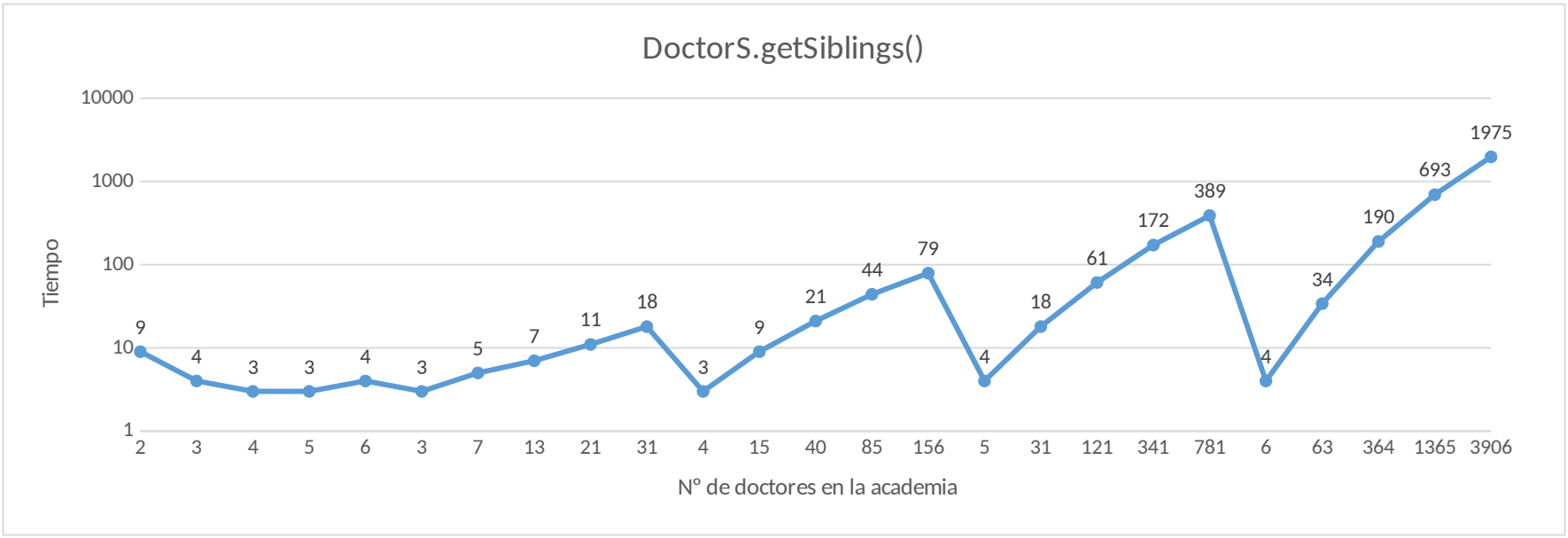


Ilustración 4

La gráfica muestra un crecimiento lineal respecto al número de doctores de la academia ***N***.

Cada vez que se buscan los hermanos es necesario recuperar el supervisor, que tiene un coste *O(1)*, localizar su rama en la academia (coste O(N)) y finalmente devolver sus hijos O(1), resultando en un coste lineal **O(N)**, como indicó el alumno en el ejercicio anterior.

6 Gráfica para el escenario completo

6.1 getDescendants()

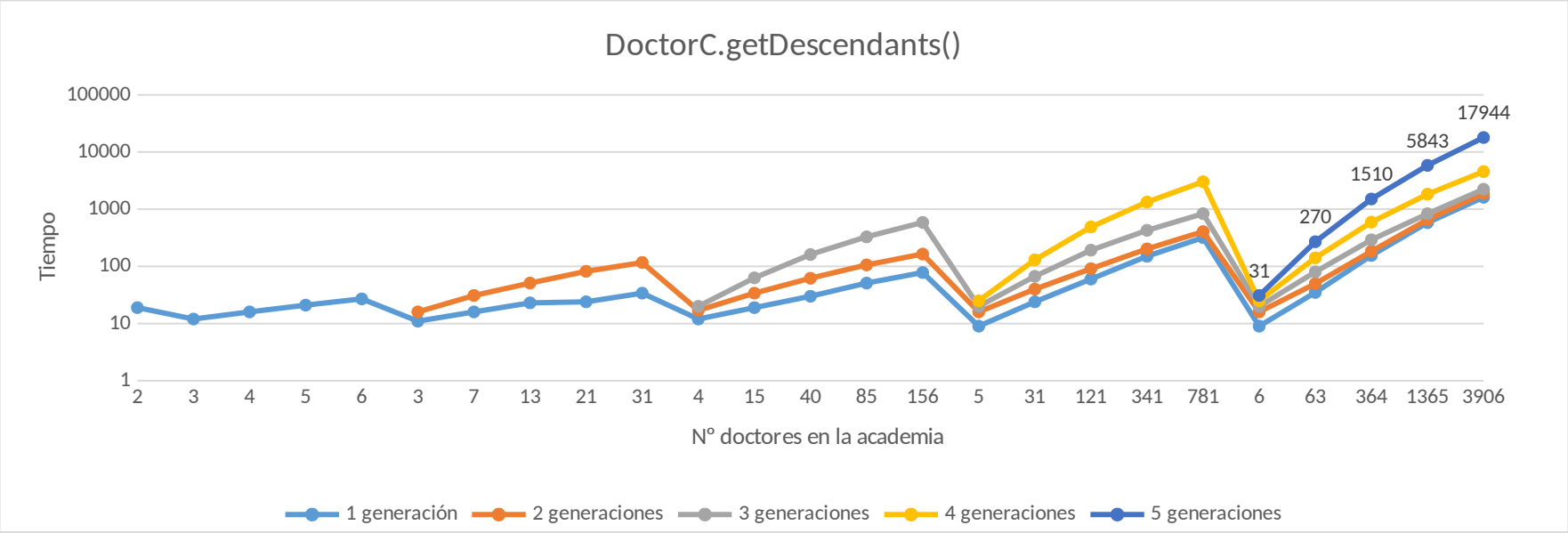


Ilustración 5

La gráfica muestra un incremento lineal **O(N)** respecto al número de doctores de la academia. Al igual que en el escenario simplificado las generaciones del árbol hace que el coste se incremente sensiblemente.

6.2 getSiblings()

Éste

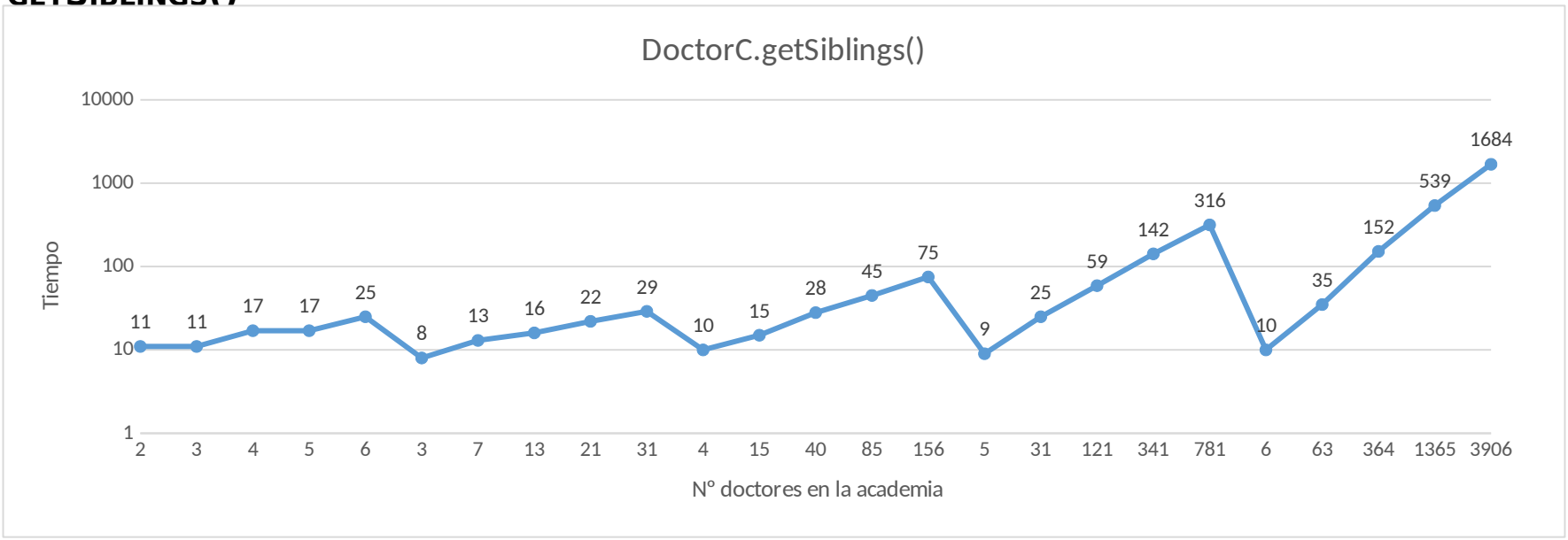


Ilustración 6

método obtiene los supervisores del *DoctorC*, que están almacenados en el propio objeto (coste constante O(1)).

Acto seguido, se recorren todos los supervisores y se comprueban sus alumnos, añadiéndolos a una lista de acceso e inserción aleatoria con eficiencia de O(1).

El peor caso implicaría que los supervisores de un doctor tienen a todos los alumnos bajo su supervisión, lo que conllevaría un coste **lineal** respecto a **N**.

Esta eficiencia coindice con lo analizado en los ejercicios anteriores, donde el alumno confirmaba que el hecho de que un doctor almacenase sus estudiantes y/o supervisores mejoraría la eficiencia a costa del consumo de memoria.

6.3

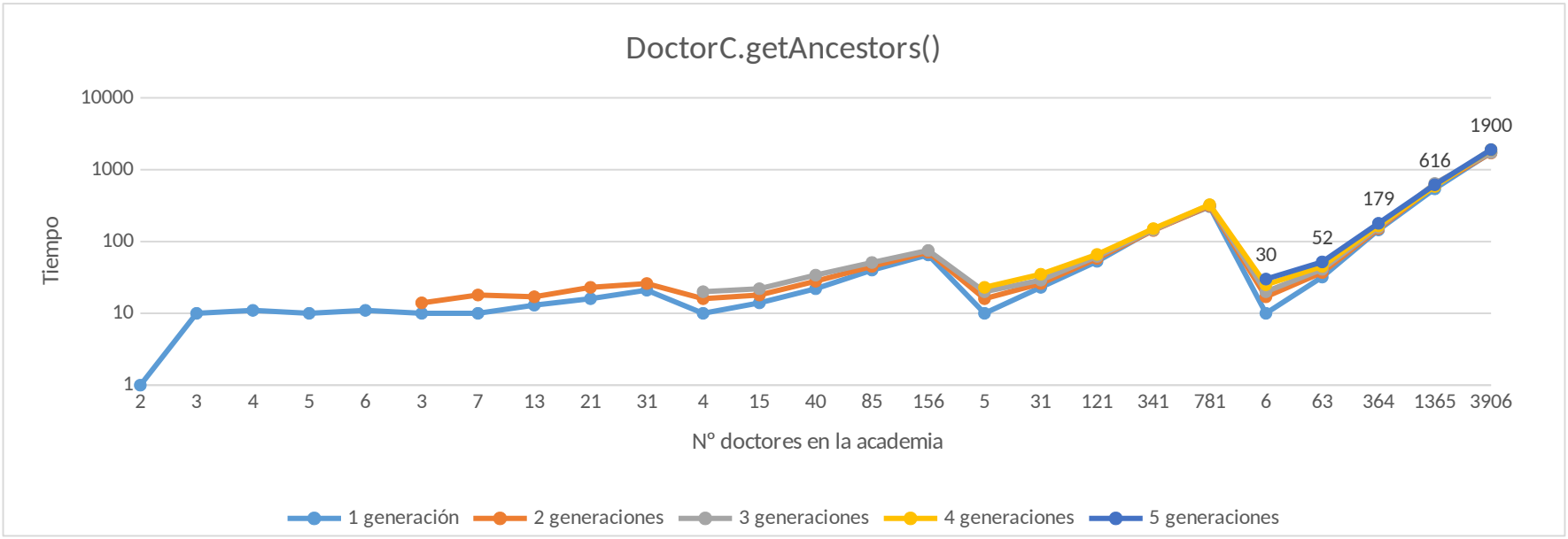


Ilustración 7

La gráfica muestra un crecimiento lineal respecto al número de doctores **N**. Esto se debe a que la operación de *getSupervisores()* tiene una eficiencia constante O(1) y que el número de generaciones es lo que hace que se llame más veces, iterando por más doctores de la academia y causando que el coste se incremente.

6.4

La

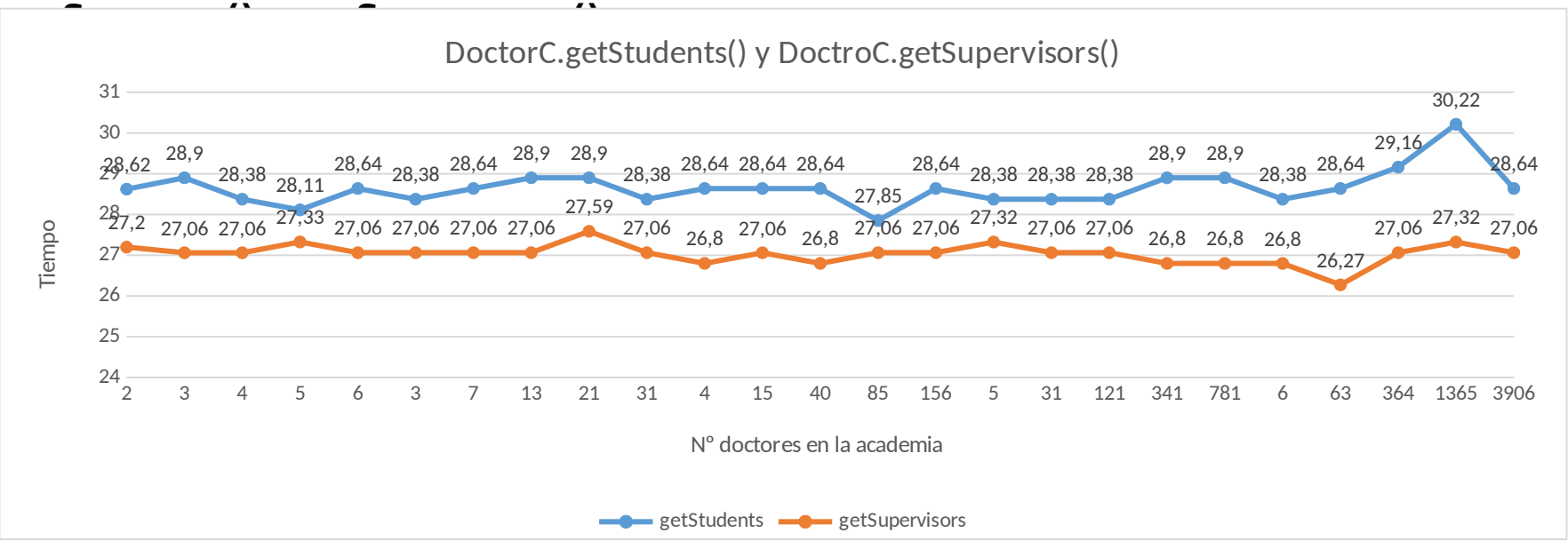


Ilustración 8

gráfica muestra un crecimiento constante ***O(1)*** respecto al número de doctores en la academia.

Esto es así porque el alumno optó por hacer que cada doctor conociese tanto los alumnos como los supervisores que posee, haciendo que muchas de las operaciones anteriormente ilustradas fuesen mucho más eficientes.

Esto combinado con una lista de acceso y escritura aleatoria que garantiza una eficiencia de O(1) (*ListaDoctores*) hace que los métodos en ningún momento pasen a tener un coste exponencial.

7 Código fuente de la herramienta

7.1 AcademiaS

```
package es.uned.lsi.eped.pract2016;

import es.uned.lsi.eped.DataStructures.DoctorIF;
import es.uned.lsi.eped.DataStructures.AcademiaIF;
import es.uned.lsi.eped.DataStructures.IteratorIF;
import es.uned.lsi.eped.DataStructures.Tree;
import es.uned.lsi.eped.DataStructures.TreeIF;
import es.uned.lsi.eped.DataStructures.List;
import es.uned.lsi.eped.DataStructures.CollectionIF;

/**
 * Academia sencilla basado en un árbol N-ario para implementar
las relaciones
 * supervisor->estudiante.
 *
 * Se apoya en el hecho de que cada doctor conoce directamente su
supervisor.
 *
 * @author Héctor Luaces NOvo
 */
public class AcademiaS implements AcademiaIF {
    private TreeIF<DoctorIF> doctores;

    public AcademiaS() {
        this.doctores = new Tree<>();
    }

    @Override
    public int size() {
        return this.doctores.size();
    }

    @Override
    public boolean isEmpty() {
        return this.doctores.isEmpty();
    }

    @Override
    public boolean contains(DoctorIF e) {
        return this.doctores.contains(e);
    }

    @Override
    public void clear() {
        this.doctores.clear();
    }

    @Override
    public IteratorIF<DoctorIF> iterator() {
```

```
        return this.doctores.iterator(TreeIF.PREORDER);
    }

    public TreeIF<DoctorIF> getDoctores() {
        return this.doctores;
    }

    /**
     * Dado el ID de un doctor y un árbol (que se asume es una
rama de la
     * academia) devuelve la rama que coincide con el doctor.
     *
     * Es un método recursivo auxiliar.
     *
     * @param id Id del doctor a averiguar
     * @param arbol Árbol en el que estamos buscando actualmente
     * @return Rama del doctor o null si no existe
     */
    private TreeIF<DoctorIF> getRamaDoctor(int id,
TreeIF<DoctorIF> arbol) {
        if ( arbol.isEmpty() )
            return null;

        if ( arbol.getRoot().getID() == id )
            return arbol;

        IteratorIF <TreeIF<DoctorIF>> it =
arbol.getChildren().iterator();

        while(it.hasNext()) {
            TreeIF<DoctorIF> rama_hijo = it.getNext();
            rama_hijo = getRamaDoctor(id, rama_hijo);

            if ( rama_hijo != null )
                return rama_hijo;
        }

        return null;
    }

    /**
     * Método recursivo que, dado un ID de doctor, devuelve la
rama del árbol
     * de la academia que lo tiene como raíz.
     *
     * @param id ID del doctor a buscar
     * @return Rama del árbol de la academia que tiene como raíz a
dicho doctor
     */
    private TreeIF<DoctorIF> getRamaDoctor(int id) {
        return this.getRamaDoctor(id, this.doctores);
    }

    @Override
    public DoctorIF getFounder() {
        return this.doctores.getRoot();
    }
```

```
    @Override
    public DoctorIF getDoctor(int id) {
        IteratorIF<DoctorIF> it = this.iterator();

        while (it.hasNext()) {
            DoctorIF d = it.getNext();

            if ( d.getID() == id )
                return d;
        }

        return null;
    }


    @Override
    public void addDoctor(DoctorIF newDoctor, DoctorIF supervisor)
{
        if ( getFounder() == null ) {
            this.doctores.setRoot(newDoctor);
            return;
        }
        TreeIF<DoctorIF> rama = getRamaDoctor(supervisor.getID());
        TreeIF<DoctorIF> hijo = new Tree<DoctorIF>(newDoctor);
        ((DoctorS) newDoctor).setSupervisor(supervisor);
        rama.addChild(1, hijo);
    }

    @Override
    public void addSupervision(DoctorIF student, DoctorIF
supervisor) {
        TreeIF<DoctorIF> rama =
this.getRamaDoctor(supervisor.getID());

        if ( rama.contains(student) )
            return;

        rama.addChild(1, new Tree<>(student));
        ((DoctorS) student).setSupervisor(supervisor);
    }

    /**
     * Busca en el árbol de la academia el doctor que pasamos como
parámetro y
     * devuelve la rama de su supervisor.
     *
     * @param id ID del doctor para el que queremos conocer la
rama de su super
     * visor
     * @return La rama del doctor supervisor o null
     */
    private TreeIF<DoctorIF> findRamaSupervisor(int id) {
        DoctorS doc = (DoctorS) getDoctor(id);

        if ( doc.getSupervisor() == null )
            return null;
```

```
        return getRamaDoctor(doc.getSupervisor().getID());
    }

    /**
     * Devuelve el doctor supervisor que corresponde con un ID de
doctor.
     *
     * @param id ID para el que queremos conocer el supervisor.
     * @return Doctor supervisor para el doctor pasado como
parámetro.
     */
    public DoctorIF getSupervisor(int id) {
        DoctorS doc = (DoctorS) getDoctor(id);

        return doc.getSupervisor();
    }

    /**
     * Método recursivo auxiliar que desciende por los hijos de un
doctor para
     * devolver un número de descendientes en función de las
generaciones
     * pasadas como parámetro.
     *
     * @param generations Generaciones de descendientes que
queremos.
     * @param arbol Árbol (rama) por la que vamos a profundizar.
     * @param ret Lista con los doctores que hemos encontrado
hasta ahora
     * @return La lista de los descendientes para las generaciones
pasadas
     * como parámetro.
     */
    private CollectionIF<DoctorIF> getDescendants(
        int generations, TreeIF<DoctorIF> arbol, List<DoctorIF>
ret
    ) {
        if ( arbol.isEmpty() || generations < 1 )
            return ret;

        IteratorIF<TreeIF<DoctorIF>> it =
arbol.getChildren().iterator();

        while(it.hasNext()) {
            TreeIF<DoctorIF> arbol_hijo = it.getNext();

            getDescendants(generations - 1, arbol_hijo, ret);
            ret.insert(arbol_hijo.getRoot(), 1);
        }

        return ret;
    }

    /**
     * Método recursivo que recorre todo el árbol para buscar las
descendientes
```

```
     * de un doctor.
     *
     * @param id_doctor ID del doctor para el que queremos conocer
los descendientes.
     * @param generations Generaciones de descendientes que
queremos conocer.
     * @return La lista con los descendientes académicos del
doctor
     */
    public CollectionIF<DoctorIF> getDescendants(int id_doctor,
int generations) {
        return this.getDescendants(generations,
getRamaDoctor(id_doctor), new List<DoctorIF>());
    }

    /**
     * Devuelve una lista con los hermanos de un doctor pasado
como parémtro.
     *
     * @param id_doctor el ID del doctor para el que queremos
conocer los hemanos
     * @return colección con los hermanos del doctor
     */
    public CollectionIF<DoctorIF> getSiblings(int id_doctor) {
        TreeIF<DoctorIF> rama_supervisor =
this.findRamaSupervisor(id_doctor);
        List<DoctorIF> lista_hermanos = new List<DoctorIF>();

        if ( rama_supervisor == null )
            return lista_hermanos;


        IteratorIF<TreeIF<DoctorIF>> it =
rama_supervisor.getChildren().iterator();

        while(it.hasNext()) {
            DoctorIF d = it.getNext().getRoot();

            if ( d.getID() == id_doctor )
                continue;

            lista_hermanos.insert(d, 1);
        }

        return lista_hermanos;
    }

    /**
     * Método recursivo auxiliar que asciende por los supervisores
de un doctor para
     * devolver un número de ancestros en función de las
generaciones
     * pasadas como parámetro.
     *
     * @param doc Doctor para el que queremos conocer los
ancestros
     * @param generations Generaciones de ancestros que queremos.
```

```
     * @return La colección de los ancestrospara las generaciones
pasadas
     * como parámetro.
     */
    public CollectionIF<DoctorIF> getAncestors(DoctorS doc, int
generations) {
        List<DoctorIF> ret = new List<>();

        doc = (DoctorS) doc.getSupervisor();

        while(generations-- > 0 && doc != null) {
            ret.insert(doc, 1);
            doc = (DoctorS) doc.getSupervisor();
        }

        return ret;
    }

    void setFounder(DoctorS founder) {
        this.doctores.setRoot(founder);
    }
}
```

7.2 DoctorS

```
package es.uned.lsi.eped.pract2016;

import es.uned.lsi.eped.DataStructures.DoctorIF;
import es.uned.lsi.eped.DataStructures.AcademiaIF;
import es.uned.lsi.eped.DataStructures.CollectionIF;

/**
 * Doctor para una academia sencilla. Guarda internamente una
referencia
 * a su supervisor para garantizar un acceso O(1) al mismo.
 *
 * @author Héctor Luaces Novo
 */
public class DoctorS implements DoctorIF {
    private int id;
    private AcademiaS academia;
    private DoctorIF supervisor;
    public DoctorS() {
    }

    public DoctorS(int id) {
        this.id = id;
    }

    public DoctorS(int id, AcademiaIF academia) {
        this.id = id;
        this.academia = (AcademiaS) academia;
    }

    public DoctorS(int id, AcademiaIF academia, DoctorIF
supervisor) {
        this.id = id;
```

```
        this.academia = (AcademiaS) academia;
        this.supervisor = supervisor;
    }

    public DoctorIF getSupervisor() {
        return this.supervisor;
    }

    @Override
    public CollectionIF<DoctorIF> getAncestors(int generations) {
        return this.academia.getAncestors(this, generations);
    }

    @Override
    public CollectionIF<DoctorIF> getStudents() {
        return this.getDescendants(1);
    }

    @Override
    public CollectionIF<DoctorIF> getDescendants(int generations)
{
        return this.academia.getDescendants(id, generations);
    }

    @Override
    public CollectionIF<DoctorIF> getSiblings() {
        return this.academia.getSiblings(id);
    }

    public AcademiaS getAcademia() {
        return academia;
    }

    @Override
    public int getID() {
        return this.id;
    }

    void setAcademia(AcademiaS a) {
        this.academia = (AcademiaS) a;
    }

    public void setSupervisor(DoctorIF supervisor) {
        this.supervisor = supervisor;
    }
}
```

7.3 ListaDoctores

```
package es.uned.lsi.eped.pract2016;

import es.uned.lsi.eped.DataStructures.DoctorIF;
import es.uned.lsi.eped.DataStructures.IteratorIF;
import es.uned.lsi.eped.DataStructures.List;
import es.uned.lsi.eped.DataStructures.ListIF;

/**
```

```
 * Lista de doctores: una ListaIF que usa internamente un array
para garantizar
 * un acceso O(1) al elemento N de la lista. El array crecerá en
un 50% cada vez
 * que no haya espacio para almacenar un nuevo espacio, lo que
significa que
 * una inserción costará O(1) o O(N) en función de si hay o no
espacio. Con esto
 * garantizamos que el coste es de O(1) amortizado, puesto que
para cada K veces
 * que hemos tenido que redimensionar el array los habremos usado
eficientemente
 * por lo menos K/2 veces.
 *
 * Actualmente la lista asume que las posiciones de los doctores
en el array
 * interno se corresponden por su ID.
 *
 * Puesto que el ejercicio añade los doctores en forma secuencia,
no se hace
 * ningún esfuerzo en ordenar los datos y se asume que al insertar
éstos
 * se ordenarán solos.
 *
 * La lista garantiza un acceso O(1) al doctor de un ID dado,
asumiendo que
 * nada haya alterado el orden.
 *
 * Incluye también un iterador eficiente para recorrer la lista de
doctores.
 *
 *
 * @author Héctor Luaces Novo
 * @param <T> Tipo de doctor que guardará la lista
 */
public class ListaDoctores<T extends DoctorIF> implements
ListIF<T> {
    private Object[] doctores;
    private int size;

    private class IteratorDoctores implements IteratorIF<T> {
        int posicion = 0;
        @Override

        public T getNext() {
            posicion++;

            return (T) doctores[posicion - 1];
        }

        @Override
        public boolean hasNext() {
            return size > posicion;
        }

        @Override
        public void reset() {
```

```
            posicion = 0;
        }
    }

    public ListaDoctores(int capacidad) {
        this.doctores = new Object[capacidad];
    }

    private boolean isFull() {
        return size == doctores.length;
    }

    private void grow() {
        Object[] aux = this.doctores;

        this.doctores = new Object[(int) (size * 1.5f)];

        for(int i = 0; i < aux.length; i++) {
            doctores[i] = aux[i];
        }
    }

    private void desplazarDerecha(int desde) {
        for(int i = size; i > desde; i--) {
            doctores[i] = doctores[i-1] ;
        }

        doctores[desde] = null;
    }

    private void desplazarIzquierda(int desde) {
        for(int i = desde; i < size; i++) {
            doctores[i] = doctores[i+1] ;
        }

        doctores[size - 1] = null;
    }

    @Override
    public T get(int id) {
        if ( size < id - 1 )
            return null;

        return (T) this.doctores[id -1];
    }

    @Override
    public void set(int pos, T e) {
        if ( pos >= doctores.length )
            return;

        if ( doctores[pos] == null )
            size++;

        doctores[pos] = e;
    }
```

```
    @Override
    public void insert(T elem, int pos) {
        if ( this.isFull() )
            this.grow();

        if ( pos == 0 )
            pos = size;
        else if ( doctores[pos] != null )
            desplazarDerecha(pos);

        doctores[pos] = elem;
        size++;
    }

    @Override
    public void remove(int pos) {
        if ( pos > size - 1 )
            return;

        if ( doctores[pos] == null )
            return;

        desplazarIzquierda(pos);
        size--;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public boolean contains(T e) {
        if ( size >= e.getID() )
            return false;

        return doctores[e.getID() - 1] != null;
    }

    @Override
    public void clear() {
        size = 0;
        this.doctores = new Object[doctores.length];
    }

    @Override
    public IteratorIF iterator() {
        return new IteratorDoctores();
    }

    public ListIF<T> toList() {
        ListIF<T> ret = new List<>();
```

```
        for(int i = 0; i < doctores.length; i++ ) {
            if ( doctores[i] == null )
                continue;

            ret.insert((T) doctores[i], 1);
        }

        return ret;
    }

    private void debug() {
        for(int i = 0; i < doctores.length; i++) {
            T doc = (T) doctores[i];

            if ( doc == null )
                continue;

            System.out.println(" - " + i + ": " + doc.getID());
        }
    }

    public static void main(String args[]) {
        ListaDoctores<DoctorS> l = new ListaDoctores(5);

        l.insert(new DoctorS(1), 0);
        l.insert(new DoctorS(2), 0);
        l.insert(new DoctorS(3), 0);
        l.insert(new DoctorS(4), 0);

        l.debug();
        l.insert(new DoctorS(5), 0);
        l.insert(new DoctorS(6), 0);
        l.insert(new DoctorS(7), 0);

        l.debug();
        l.insert(new DoctorS(8), 1);
        l.debug();
        l.remove(7);
        l.debug();
        l.remove(0);
        l.debug();
        l.remove(1);
        l.debug();

        IteratorIF<DoctorS> it = l.iterator();

        System.out.println("Iterador");
        while(it.hasNext()) {
            DoctorS doc = it.getNext();

            System.out.println(doc.getID());
        }
    }
}
```

7.4 AcademiaC

```
package es.uned.lsi.eped.pract2016;

import es.uned.lsi.eped.DataStructures.DoctorIF;
import es.uned.lsi.eped.DataStructures.AcademiaIF;
import es.uned.lsi.eped.DataStructures.IteratorIF;

/**
 * Academia completa basada en una ListaDoctores (lista) y una
serie de nodos
 * (doctores) que conocen la información de sus antecesores y
sucesores directos.
 *
 * Esto último, unido a la ListaDoctores ayuda a mantener el
crecimiento
 * asintótico en un lugar asumible.
 *
 * @author Héctor Luaces Novo
 */
public class AcademiaC implements AcademiaIF {
    /**
     * La academiaC usará una ListaDoctores para guardar una
referencia de
     * todos los doctores y garantizar una eficiciencia de o(1) en
el acceso.
     */
    private ListaDoctores<DoctorC> doctores;
    /**
     * El fundador de la academia
     */
    private DoctorC founder;

    /**
     * Constructores para satisfacer el juego de pruebas y
empírico del equipo
     * docente.
     */
    public AcademiaC() {
        doctores = new ListaDoctores(20);
    }

    public AcademiaC(DoctorC fundador) {
        this();

        this.founder = fundador;
    }

    @Override
    public DoctorIF getFounder() {
        return founder;
    }

    @Override
    public DoctorIF getDoctor(int id) {
        return this.doctores.get(id);
```

```
    }

    @Override
    public int size() {
        return this.doctores.size();
    }

    @Override
    public void addDoctor(DoctorIF newDoctor, DoctorIF supervisor)
{
        this.doctores.insert((DoctorC) newDoctor, 0);

        // Tras añadir el doctor es necesario actualizar los
objetos de las
        // relaciones
        this.addSupervision(newDoctor, supervisor);
    }

    @Override
    public void addSupervision(DoctorIF student, DoctorIF
supervisor) {
        if ( supervisor == null )
            return;

        // Añadimos estudiante y supervisor a sendos doctores
        ( (DoctorC) student).addSupervisor(supervisor);
        ( (DoctorC) supervisor).addStudent(student);
    }

    @Override
    public boolean isEmpty() {
        return this.doctores.isEmpty();
    }

    @Override
    public boolean contains(DoctorIF e) {
        return this.doctores.contains((DoctorC) e);
    }

    @Override
    public void clear() {
        this.doctores.clear();
    }

    @Override
    public IteratorIF<DoctorIF> iterator() {
        return this.doctores.iterator();
    }

    /**
     * Establece un nuevo fundador
     * @param doc Doctor a establecer como fundador
     */
    public void setFounder(DoctorIF doc) {
        founder = (DoctorC) doc;
        addDoctor(doc, null);
    }
```

```
}
```

7.5 DoctorC

```
package es.uned.lsi.eped.pract2016;

import es.uned.lsi.eped.DataStructures.DoctorIF;
import es.uned.lsi.eped.DataStructures.AcademiaIF;
import es.uned.lsi.eped.DataStructures.CollectionIF;
import es.uned.lsi.eped.DataStructures.IteratorIF;
import es.uned.lsi.eped.DataStructures.List;
import es.uned.lsi.eped.DataStructures.ListIF;

/**
 * Doctor para la academia completa.
 *
 * Un doctor de esta academia conoce sus sucesores y antecesores
directos
 * para garantizar una eficiencia de O(1) a la hora de
consultarlos.
 *
 * @author Héctor Luaces Novo
 */
public class DoctorC implements DoctorIF {
    private int id;
    private AcademiaC academia;
    private CollectionIF<DoctorIF> estudiantes;
    private CollectionIF<DoctorIF> supervisores;

    /**
     * Constructores para satisfacer el juego de pruebas del
equipo docente
     * y su implementación de IO
     */
    public DoctorC() {
        this.estudiantes = new List<>();
        this.supervisores = new List<>();
    }

    public DoctorC(int id) {
        this();

        this.id = id;
    }

    public DoctorC(AcademiaIF academia) {
        this();

        this.academia = (AcademiaC) academia;
    }

    public DoctorC(int id, AcademiaIF academia) {
        this();

        this.id = id;
        this.academia = (AcademiaC) academia;
    }
```

```
    public DoctorC(int id, AcademiaIF academia,
CollectionIF<DoctorIF> supervisores) {
        this();

        this.id = id;
        this.academia = (AcademiaC) academia;
        this.supervisores = supervisores;
    }

    @Override
    public CollectionIF<DoctorIF> getStudents() {
        return this.estudiantes;
    }

    @Override
    public int getID() {
        return this.id;
    }


    public CollectionIF<DoctorIF> getSupervisors() {
        return this.supervisores;
    }

    /**
     * Método privado recursivo usado para encontrar los
antecesores de un doctor.
     *
     * Usa una ListaDoctores con espacio suficiente para albergar
a toda la
     * academia y controlar así los duplicados al mismo tiempo que
minifica
     * el acceso.
     *
     * @param generations Número de generaciones que queremos
obtener.
     * @param doc Doctor para el que queremos obtener las
generaciones.
     * @param ret Lista de antecesores que lleva el método
recursivo hasta ahora
     * @return La lista de los antecesores de dichas generaciones
para el doctor
     * seleccionado.
     */
    private ListaDoctores<DoctorIF> getAncestors(int generations,
DoctorC doc, ListaDoctores<DoctorIF> ret) {
        if ( generations < 1 || doc.getSupervisors().size() == 0)
            return ret;

        IteratorIF<DoctorIF> it = doc.getSupervisors().iterator();

        while(it.hasNext()) {
            DoctorC sup = (DoctorC) it.getNext();

            if ( ! ret.contains(sup) )
                ret.set(sup.getID() - 1, sup);
```

```
            getAncestors(generations -1, sup, ret);
        }

        return ret;
    }

    @Override
    public CollectionIF<DoctorIF> getAncestors(int generations) {
        return getAncestors(
            generations, this, new
ListaDoctores<DoctorIF>(academia.size())
        ).toList();
    }


    /**
     * Método privado recursivo usado para encontrar los
descendientes de un doctor.
     *
     * Usa una ListaDoctores con espacio suficiente para albergar
a toda la
     * academia y controlar así los duplicados al mismo tiempo que
minifica
     * el acceso.
     *
     * @param generations Número de generaciones que queremos
obtener.
     * @param doc Doctor para el que queremos obtener las
generaciones.
     * @param ret Lista de descendientes que lleva el método
recursivo hasta ahora
     * @return La lista de los descendientes de dichas
generaciones para el doctor
     * seleccionado.
     */
    private ListaDoctores<DoctorIF> getDescendants(int
generations, DoctorC doc, ListaDoctores<DoctorIF> ret) {
        if ( generations < 1 || doc.getStudents().size() == 0)
            return ret;

        IteratorIF<DoctorIF> it = doc.getStudents().iterator();

        while(it.hasNext()) {
            DoctorC sup = (DoctorC) it.getNext();

            if ( ! ret.contains(sup) )
                ret.set(sup.getID() - 1, sup);

            getDescendants(generations -1, sup, ret);
        }

        return ret;
    }

    @Override
```

```
    public CollectionIF<DoctorIF> getDescendants(int generations)
{
        return getDescendants(
            generations, this, new
ListaDoctores<DoctorIF>(academia.size())
        ).toList();
    }

    @Override
    public CollectionIF<DoctorIF> getSiblings() {
        ListaDoctores<DoctorIF> ret = new
ListaDoctores<>(academia.size());

        if ( this.supervisores.isEmpty() )
            return ret;

        IteratorIF<DoctorIF> it = this.supervisores.iterator();

        while(it.hasNext()) {
            DoctorC doc = (DoctorC) it.getNext();

            if ( ret.contains(doc) )
                continue;

            IteratorIF<DoctorIF> st =
doc.getStudents().iterator();

            while(st.hasNext()) {
                DoctorC student = (DoctorC) st.getNext();

                if ( student.getID() != id )
                    ret.set(student.getID() - 1, student);
            }
        }

        return ret.toList();
    }

    /**
     * EStablece la academia de éste doctor
     * @param a Academia del doctor
     */
    public void setAcademia(AcademiaIF a) {
        this.academia = (AcademiaC) a;
    }

    /**
     * Añade un supervisor a la lista del doctor
     * @param supervisor Supervisor a añadir a la lista del
estudiante (se
     * comprueban duplicados)
     */
    public void addSupervisor(DoctorIF supervisor) {
        ListIF<DoctorIF> l = (ListIF<DoctorIF>) this.supervisores;

        if ( ! l.contains(supervisor) )
            l.insert(supervisor, 1);
```

```
    }

    /**
     * Añade un estudiante a la lista del doctor
     * @param student Estudiante a añadir. Se comprueban
duplicados.
     */
    public void addStudent(DoctorIF student) {
        ListIF<DoctorIF> l = (ListIF<DoctorIF>) this.estudiantes;

        if ( ! l.contains(student) )
            l.insert(student, 1);
    }
}
```

8 Anexo I: Tabla de datos empíricos

En este anexo se incluye la tabla de datos brutos que se utilizó para elaborar las gráficas.

8.1 AcademiaS

	getDescendants()						getAncestors()					
Nº doctores	1 generación	2 generaciones	3 generaciones	4 generaciones	5 generaciones	getSiblings()	1 generación	2 generaciones	3 generaciones	4 generaciones	5 generaciones	getSupervisor()
2	15	0	0	0	0	9	13					49,91
3	5					4	24					55,17
4	6					3	31					39,41
5	8					3	41					42,04
6	6					4	53					44,67
3	5	4				3	21	53				76,2
7	3	9				5	31	38				28,9
13	5	17				7	16	31				28,9
21	4	27				11	16	37				28,9
31	5	39				18	32	15				31,53
4	2	4	6			3	16	38	31			39,41
15	3	9	14			9	15	34	20			28,9
40	4	16	33			21	15	32	34			28,9
85	6	27	64			44	16	16	31			31,53
156	6	39	116			79	16	37	32			28,9
5	2	3	4	6		4	12	31	36	39		47,29
31	3	8	14	39		18	13	23	31	38		31,53
121	4	15	30	148		61	14	22	31	38		31,52
341	4	24	60	433		172	3	31	16	38		28,9
781	6	38	103	994		389	16	31	31	38		31,53
6	2	4	5	6	8	4	14	31	32	37	47	44,67
63	3	8	14	39	60	34	16	22	31	47	38	31,53
364	4	15	32	152	315	190	16	31	33	36	47	28,9
1365	4	25	75	453	1092	693	15	16	31	38	47	31,53
3906	6	38	105	1001	3232	1975	23	15	32	48	36	28,9

8.2 AcademiaC

	getDescendants()						getAncestors()						
Nº doctores	1 generación	2 generaciones	3 generaciones	4 generaciones	5 generaciones	getSiblings()	1 generación	2 generaciones	3 generaciones	4 generaciones	5 generaciones	getStudents	getSupervisors
2	19					11	1					28,622	27,198
3	12					11	10					28,901	27,061
4	16					17	11					28,375	27,061
5	21					17	10					28,113	27,325
6	27					25	11					28,638	27,062
3	11	16				8	10	14				28,375	27,062
7	16	31				13	10	18				28,638	27,062
13	23	51				16	13	17				28,901	27,062
21	24	82				22	16	23				28,901	27,587
31	34	117				29	21	26				28,375	27,062
4	12	17	20			10	10	16	20			28,638	26,799
15	19	34	63			15	14	18	22			28,638	27,061
40	30	62	161			28	22	28	34			28,638	26,799
85	51	106	328			45	40	45	51			27,85	27,062
156	78	164	587			75	65	71	75			28,638	27,062
5	9	16	20	25		9	10	16	20	23		28,375	27,324
31	24	40	67	130		25	23	26	29	35		28,375	27,062
121	60	91	192	489		59	53	58	62	66		28,375	27,062
341	150	202	427	1332		142	146	142	145	151		28,9	26,799
781	319	406	835	3025		316	306	312	322	325		28,901	26,799
6	9	16	20	25	31	10	10	17	20	25	30	28,375	26,798
63	35	50	80	141	270	35	32	37	41	45	52	28,638	26,273
364	155	184	292	594	1510	152	145	151	157	165	179	29,163	27,062
1365	580	655	834	1832	5843	539	540	603	643	584	616	30,215	27,324
3906	1617	1874	2234	4544	17944	1684	1736	1704	1727	1889	1900	28,638	27,062