ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

ESTUDIO DE LAS ESTRUCTURAS DE ÁRBOL EN JAVA

AUTOR: **Esperanza Madrid Alcaraz.**
DIRECTOR: **Mª Francisca Rosique Contreras.**

Julio 2012

Autor	Esperanza Madrid Alcaraz
E-mail del Autor	espe2785@hotmail.com
Director(es)	Mª Francisca Rosique Contreras
Título del PFC	ESTUDIO DE LAS ESTRUCTURAS DE ÁRBOL EN JAVA
Descriptores	

Resumen

La primera etapa de este proyecto se trataba de la implementación de un árbol multirrama o multicamino para su visualización a través de la consola del programa Eclipse como continuación a la parte de arboles estudiada en la asignatura “Fundamentos de la programación”.

Como mejora se ha querido desarrollar una interfaz grafica para que la inserción, eliminación y visualización de los datos del árbol sea más clara, rápida y sencilla. El caso de estudio seleccionado se centra en la creación y gestión de un árbol genealógico ya que es un claro ejemplo de árbol multirrama donde a los padres existentes se les podrá añadir uno, dos o más hijos. Además se podrán eliminar dichos hijos y la visualización de los mismos.

Titulación	Ingeniero Técnico en Telecomunicaciones, esp. Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Julio - 2012

Índice General

1. Alcance y contenido

1.1 Introducción

Hasta ahora el análisis de estructura de datos se ha limitado a los arboles en que cada nodo tiene como máximo dos descendientes o hijos, es decir, a los arboles binarios. Esto resulta perfectamente apropiado si , por ejemplo, se quieren representar relaciones familiares en las que cada persona se encuentre relacionada con sus padres. Pero si la relación es a la inversa necesitamos una estructura que asocie a cada padre un número arbitrario de hijos. A estas estructuras se les llama arboles multicamino o n-arios. Este ultimo termino viene del ingles n-ary, donde n es el grado máximo de los nodos del árbol, por ejemplo, si n es igual a 2 al árbol se le llama binario, para n igual a 3 se le llama terciario, para n igual a 4 se llama cuaternario, y así sucesivamente.

1.2 Estructura del documento

Este proyecto se divide en diferentes secciones con información detallada de los puntos más relevantes:

- **Alcance y contenido**: Se aporta una visión general del proyecto.
- **Estructuras de datos:** Se explica con detenimiento en que consisten los arboles en java.
- **Diagramas UML**: Se da información sobre el conjunto de herramientas para analizar y diseñar el programa**.**
- **Orden de complejidad de los algoritmos:** Se presentan los conceptos que utilizan los informáticos para analizar la eficacia de los algoritmos.
- **Caso de estudio:** Esta sección presenta un escenario que muestra cómo el programa interactúa para lograr un objetivo específico.
- **Conclusiones:** Se discuten los aspectos más importantes del proyecto.
- **Anexos:** Se incluye el código de Java con una explicación detallada.
- **Bibliografía y referencias:** Se dan a conocer los algunos de los libros de donde se ha extraído información para completar este proyecto.

2. Estructuras de datos

2.1 Introducción a los arboles.

Un árbol es una estructura jerárquica de datos que imita la forma de un árbol, un conjunto de nodos conectados. Un nodo es la unidad sobre la que se construye el árbol y puede tener ceros o mas nodos hijos conectados a él. Se dice que un nodo *a* es padre de un nodo *b* si existe un enlace desde *a* hasta *b*. Solo puede haber un único nodo sin padres, que llamaremos raíz. Un nodo que no tiene hijos se conoce como hoja y a los demás nodos se les conoce como ramas.

Formalmente, un árbol se puede definir de manera recursiva, se utiliza la recursión para definir un árbol porque es una característica inherente a los mismos, como:

1. Un solo nodo es, por si mismo, un árbol. Ese nodo es también la raíz de dicho árbol.

2. Se supone que n es un nodo y que A^1, A^2,..., A^k son arboles con raíces n^1, n^2,…, n^k respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n^1, n^2,…, n^k. En dicho árbol, n es la raíz y A^1, A^2,..., A^k son los subarboles (o arboles hijos) de la raíz. Los nodos n^1, n^2 ,… ,n^k reciben el nombre de hijos del nodo n y el nodo n recibe el nombre de padre de dichos nodos

Como ejemplo se puede considerar el índice de un libro.

T1 (Tema 1)

1.1.-(Pregunta 1 del Tema 1)

1.2.-(Pregunta 2 del Tema 1)

T2 (Tema 2)

2.1.-(Pregunta 1 del Tema 2)

2.1.1.-(Pregunta 1 de la pregunta 1 del Tema 2)

2.1.2.-(Pregunta 2 de la pregunta 1 del Tema 2)

2.2.-(Pregunta 2 del Tema 2)

2.3.-(Pregunta 3 del Tema 2)

T3 (Tema 3)

Tal índice es el árbol que se muestra en la Figura 2.1. La relación padre-hijo entre dos nodos se representa por una línea descendente que los une. Normalmente, los arboles se dibujan de arriba a abajo, con el padre encima de los hijos. En el ejemplo, la relación de paternidad representa inclusión: el libro está compuesto por los temas 1, 2 y 3.

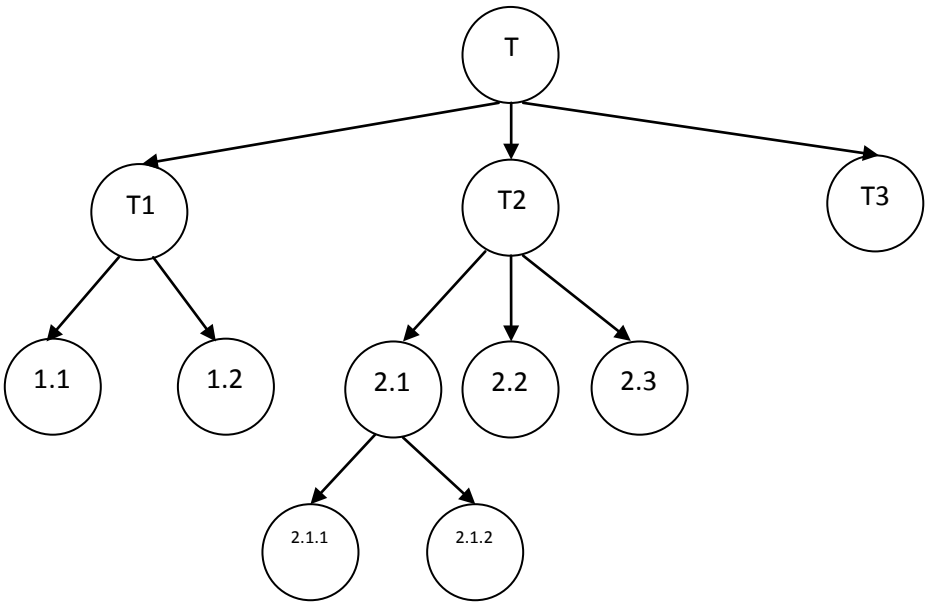


Figura 2.1. Ejemplo del índice de un libro.

Se llaman hermanos a los arboles hijos del mismo padre. Ejemplo: 2.1, 2.2 y 2.3 son hermanos.

Si n^1, n^2,…, n^k , es una sucesión de nodos de un árbol tal que n^i es el padre de n^{i+1} para i = 1 , 2 ,…, K-1, entonces a la sucesión se denomina del nodo n^1 al nodo n^k. Ejemplo: T2, 2.1, 2.1.1 es un camino.

La longitud de un camino es el número de nodos del camino menos 1. Por lo tanto, existen caminos de longitud cero, que son aquellos que van de cualquier nodo a sí mismo. Ejemplo: el camino T2, 2.1, 2.1.1 tiene longitud 2.

Si existe un camino de un nodo *a* a otro *b,* entonces *a* es un antecesor de *b*, y b es un descendiente de a. Cada nodo es a la vez un antecesor y un descendiente de sí mismo. En nuestro ejemplo, los antecesores de 2.1 son el mismo, T2 y Libro. Un antecesor o un descendiente de un nodo que no sea el mismo, recibe el nombre de antecesor propio o descendiente propio, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes propios se denomina hoja o nodo terminal.

Los atributos de un árbol son los siguientes:

- **El grado** de un nodo es el número de subarboles que tiene. En nuestro ejemplo el grado de T2 es 3. Por lo tanto son nodos terminales u hojas los nodos de grado 0.

- **La altura** de un nodo es la longitud del camino más largo de ese nodo a una hoja.
- **El nivel o profundidad** de un nodo es la longitud del único camino desde la raíz a ese nodo. Por definición la raíz tiene nivel 0. La profundidad de un árbol se define como el máximo de los niveles de los nodos del árbol.

2.1.1. Orden de los nodos

Se habla de un árbol no ordenado cuando explícitamente se ignora el orden de los hijos.

A menudo los hijos de un nodo se ordenan de izquierda a derecha. El orden de izquierda a derecha de los hermanos se puede extender para comparar dos nodos cualesquiera entre los cuales no exista la relación antecesor-descendiente. La regla es que si *a* y *b* son hermanos y *a* esta a la izquierda de *b*, entonces todos los descendientes de *a* están a la izquierda de todos los descendientes de *b*.

Dado un nodo n, una regla sencilla para determinar que nodos están a su izquierda y cuales a su derecha, consiste en dibujar el camino de la raíz a n. Todos los nodos que salen a la izquierda de este camino, y todos sus descendientes, están a la izquierda de m. Los nodos, y sus descendientes, que salen a la derecha, están a la derecha de n.

2.1.2. Recorridos de un árbol

Hay varias maneras de recorrer los nodos de un árbol para ser procesados con una cierta operación. Se va a estudiar dos estrategias para recorrer un árbol y procesar sus nodos:

- Recorrido en profundidad.
- Recorrido en anchura.

2.1.2.1 Recorrido en profundidad

Existen tres formas de recorrer un árbol en profundidad:

A) Preorden u orden previo.

B) Inorden u orden simétrico.

C) Postorden u orden posterior.

Estas tres formas de ordenar un árbol se definen de forma recursiva como sigue:

1. Si el árbol A es nulo, entonces la lista está vacía, es el listado de los nodos del árbol A en los órdenes preorden, inorden y postorden.
2. Si el árbol A tiene un solo nodo, entonces ese nodo constituye el listado del árbol A en los tres órdenes (preorden, inorden y postorden).
3. Si el árbol A tiene más de un nodo, es decir, tiene como raíz el nodo n y los subárboles $A^1, A^2, \ldots, A^k$ entonces:

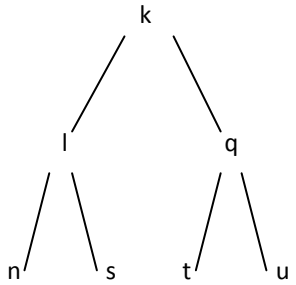


Figura 2.1.2.1. Ejemplo de recorrido de un árbol en profundidad.

A)El listado en preorden de los nodos del árbol A esta formado por la raíz del árbol A, seguida de los nodos del árbol A^1 en preorden, luego por los nodos A^2 en preorden y así sucesivamente hasta los nodos de A^k en preorden.

A continuación se ve cómo será el recorrido en preorden de la Figura 2.1.2.1.

El listado en preorden es: k – l – n – s – q – t – u

B) El listado en inorden de los nodos del árbol A esta constituido por los nodos del árbol de A^1 en inorden, seguidos de la raíz n y luego por los nodos de A^2,…, A^k en inorden.

A continuación se ve cómo será el recorrido en inorden de la Figura 2.1.2.1.

El listado en inorden es: n – l – s – k – t – q – u

Siempre se debe llegar al último nivel y luego se sube hacia arriba.

C) El listado en postorden de los nodos del árbol A esta formado por los nodos del árbol A^1 en postorden, luego los nodos de A^2 en postorden y así sucesivamente hasta los nodos de A^k en postorden y por último la raíz n.

A continuación se ve cómo será el recorrido en postorden de la Figura 2.1.2.1.

El listado en postorden en: n – s – l – t –u – q – k

2.1.2.2 Recorrido en anchura

Otro posible recorrido de un árbol es un recorrido en anchura, es decir explorando el árbol por niveles, y listando los nodos de izquierda a derecha empezando por el nivel 0, luego el nivel 1, etc.

El recorrido en anchura para el árbol de la Figura 2.1.2.2., daría la secuencia de nodos: n T1 T2 T3 T4 T5.

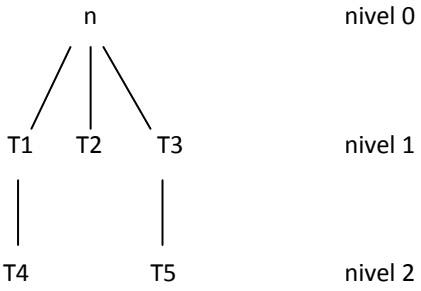


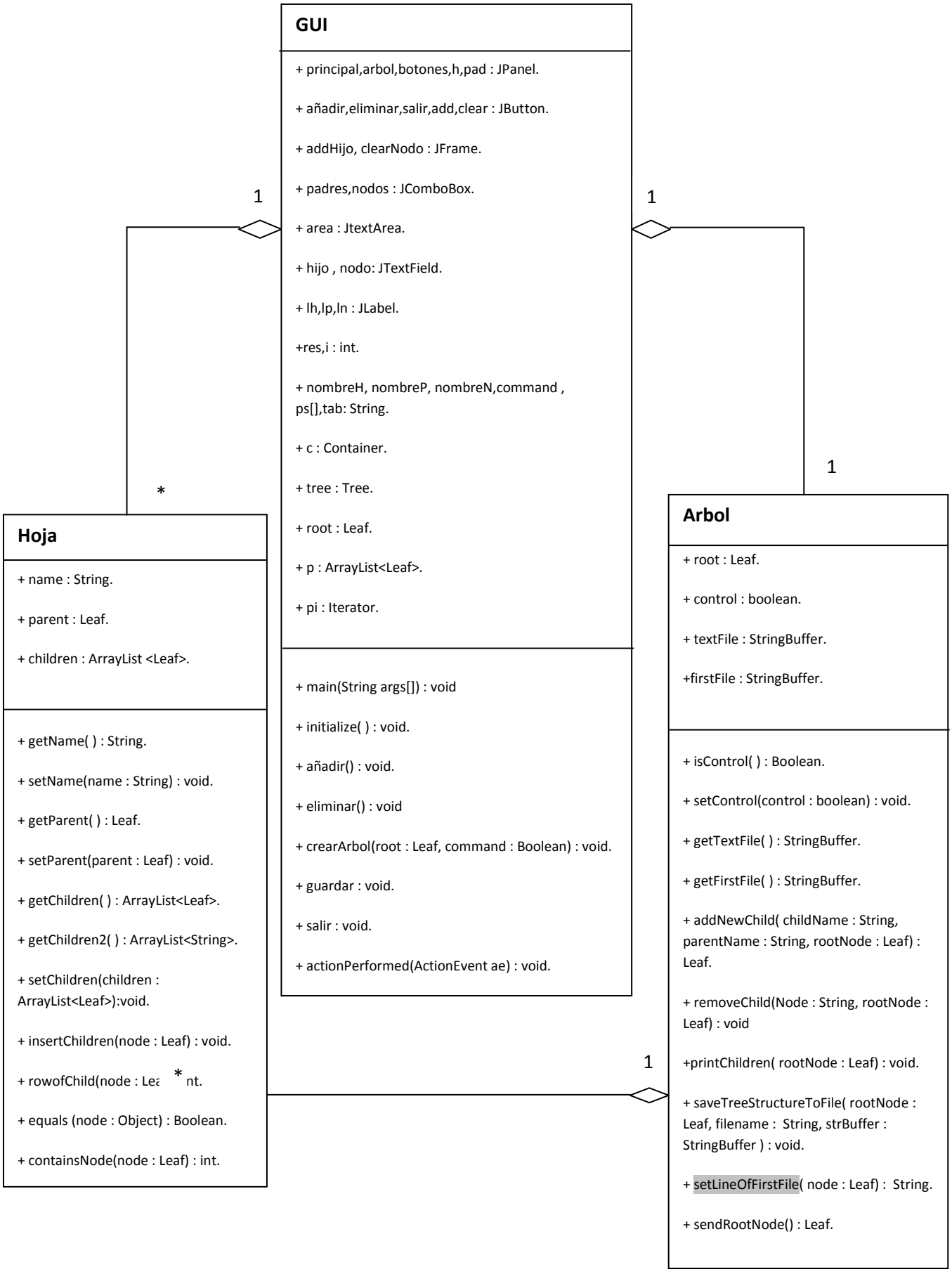
Figura 2.1.2.2. Ejemplo de recorrido de un árbol en anchura.

2.1.3. Ventajas y desventajas

La principal ventaja de los arboles multirrama consiste en que existen más nodos en un mismo nivel que en los arboles binarios con lo que se consigue que si el árbol es de búsqueda, los accesos a los nodos sean más rápidos.

El inconveniente más importante que tienen es la mayor ocupación de memoria, pudiendo ocurrir que en ocasiones la mayoría de los nodos no tengan descendientes o al menos no todos los que podrían tener desaprovechándose por tanto gran cantidad de memoria. Cuando esto ocurre lo mejor es transformar el árbol multicamino en su binario de búsqueda equivalente.

3. Diagrama UML



Las clases de este programa tienen una relación de asociación, es decir, una clase puede estar formada por objetos de otras clases.

A veces, en una relación de asociación se entiende expresamente que se cuenta con una composición total de partes. En este caso existe una relación de agregación y en el diagrama de clases un diamante vacío se utiliza en la parte que corresponde a todos. Un árbol puede tener infinitas hojas. A su vez la interfaz grafica de usuario (GUI) tiene un árbol e infinitas hojas.

4. Orden de complejidad de los algoritmos: Arboles

4.1 Introducción

Gran parte de la motivación para el diseño de árboles se debe a lacompatibilidad que presentan con algoritmos eficaces.

El análisis de algoritmos nos permite medir la dificultad inherente de un problema y evaluar la eficiencia de un algoritmo.

4.2 Tiempos de Ejecución

Una medida que suele ser útil conocer es el tiempo de ejecución de un algoritmo en función de *N*, lo que se denominara *T*(*N*). Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción.
Así, un trozo sencillo de programa como:

```
S1; FOR i:= 1 TO N DO S2 END;
```
Requiere:

```
T(N):= t1 + t2*N
```
Siendo *t*1 el tiempo que lleve ejecutar la serie "*S*1" de sentencias, y *t*2 el que lleve la serie "*S*2".

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que más que un valor *T*(*N*) debamos hablar de un rango de valores:
*T*min(*N*) <= *T*(*N*) <= *T*max(*N*) los extremos son habitualmente conocidos como "*caso peor*" y "*caso mejor*".

Entre ambos se hallará algún "*caso promedio*" o más frecuente. Cualquier fórmula *T*(*N*) incluye referencias al parámetro *N* y a una serie de constantes "*Ti*" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta.

Dado que es fácil cambiar de compilador y que la potencia de los ordenadores crece a un ritmo vertiginoso, se intentan analizar los algoritmos con algún nivel de independencia de estos factores, es decir, se buscan estimaciones generales ampliamente válidas.
No se puede medir el tiempo en segundos porque no existe un ordenador estándar de referencia, en su lugar se mide el número de operaciones básicas o elementales.

Las operaciones básicas son las que realiza el ordenador en tiempo acotado por una constante, por ejemplo:

· Operaciones aritméticas básicas
· Asignaciones de tipos predefinidos
· Saltos (llamadas a funciones, procedimientos y retorno)
· Comparaciones lógicas
· Acceso a estructuras indexadas básicas (vectores y matrices)

Es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, por ejemplo, las que más influyen en el tiempo de ejecución.
Para medir el tiempo de ejecución de un algoritmo existen varios métodos.
Veamos algunos de ellos:

a) Benchmarking

La técnica de *benchmark* considera una colección de entradas típicas representativas de una carga de trabajo para un programa.

b) Profiling

Consiste en asociar a cada instrucción de un programa un número que representa la fracción del tiempo total tomada para ejecutar esa instrucción particular. Una de las técnicas más conocidas es la *Regla 90-10*, que afirma que el 90% del tiempo de ejecución se invierte en el 10% del código.

c) Análisis

Consiste en agrupar las entradas de acuerdo a su tamaño, y estimar el tiempo de ejecución del programa en entradas de ese tamaño, *T(n)*. De este modo, el tiempo de ejecución puede ser definido como una función de la entrada. Se denotara *T(n)* como el tiempo de ejecución de un algoritmo para una entrada de tamaño *n*.

4.3 Concepto de Complejidad

La complejidad o costo de un algoritmo es una medida de la cantidad de recursos (tiempo, memoria) que el algoritmo necesita. La complejidad de un algoritmo se expresa en función del tamaño del problema. La función de complejidad tiene como variable independiente el tamaño del problema y sirve para medir la complejidad. Mide el tiempo/espacio relativo en función del tamaño del problema. El comportamiento de la función determina la eficiencia. No es única para un algoritmo: depende de los datos. Para un mismo tamaño del problema, las distintas presentaciones iniciales de los datos dan lugar a distintas funciones de complejidad. Es el caso de una ordenación si los datos están todos inicialmente desordenados, parcialmente ordenados o en orden inverso.

4.4 Órdenes de Complejidad

Se dice que *O* (*f*(*n*)) define un "orden de complejidad". Escogeremos como representante de este orden a la función *f*(*n*) más sencilla del mismo. Así tendremos:

O (1)	orden constante
O (*log n*)	orden logarítmico
O(*n*)	orden lineal
O (*n*2)	orden cuadrático
O (*na*)	orden polinomial (*a* > 2)
O (*an*)	orden exponencial (*a* > 2)
O (*n*!)	orden factorial

Es más, se puede identificar una *jerarquía de órdenes de complejidad* que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo *A* se puede demostrar de un cierto orden *O*1, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir, el menor orden de complejidad que lo cubra.

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en la habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En la Figura 4.4 se muestra un ejemplo de algunas de las funciones más comunes en análisis de algoritmos son:

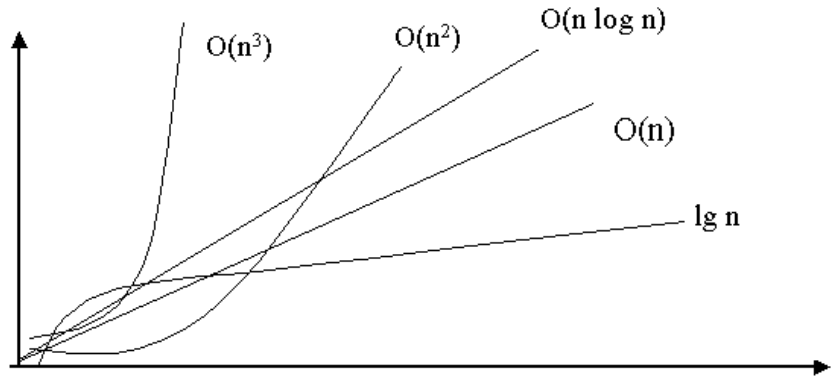


Figura 4.4. Ejemplo grafica análisis algoritmos.

La mejor técnica para diferenciar la eficiencia de los algoritmos es el estudio de los órdenes de complejidad. El orden de complejidad se expresa generalmente en términos de la cantidad de datos procesados por el programa, denominada *n*, que puede ser el tamaño dado o estimado.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales, tales como el lenguaje de programación, la habilidad del codificador, la máquina de soporte, etc. se suele trabajar con un cálculo asintótico que indica cómo se comporta el algoritmo para datos muy grandes y salvo algún coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

4.5 Complejidad en arboles

Uno de los muchos motivos por los que se estudian los árboles reside en que, aunque las listas enlazadas ofrecen tiempos de búsqueda de *O(n)*, la mayoría de las implementaciones de árboles admiten tiempos de búsqueda de *O (log n)*.

Para entender cómo se calcula el orden de complejidad en un árbol, primero se va exponer un ejemplo con una lista simplemente enlazada. Si se quiere insertar un elemento en ella se puede hacer al principio que sería directo (a esto se le llama O (1)), al final que sería lo peor al tener que recorrer todos los elementos (a esto se le conoce como O(n)) o un caso intermedio, que sería O (log n). Este último es un promedio ya que a veces estará más cerca del principio, otras más cerca del final y otras por en medio.

Ejemplo:

Si tienes 8 elementos (n=8) lo peor es insertar al final y hay qué recorrer los n elementos, es decir, 8. Lo mejor es 1, y el promedio daría log 8 =3 (el Log es en base 2).

Pues en un árbol si se tienen 8 elementos, estos se pueden repartir de una forma uniforme en ramas de forma qué por ejemplo cada una de ellas tenga 3 elementos, de esta forma al insertar un elemento en una rama como mucho se tendrá que recorrer 3 elementos, de aquí sale él O (Log n) ya qué O (Log 8) = 3.

5. Caso de estudio

5.1 Titulo

Creación de una interfaz grafica para la gestión de un árbol genealógico.

5.2 Introducción

Un árbol genealógico es un árbol que presenta la herencia biológica de una persona, comenzando con sus padres y retrocediendo a través de las generaciones, pasando por sus abuelos, bisabuelos, etc. La diferencia entre un árbol genealógico teórico y un árbol genealógico real es que el árbol real puede no estar perfectamente equilibrado, debido a la falta de datos o a la existencia de datos incompletos.

En lo que respecta a este caso de estudio, se va a crear una interfaz grafica para la gestión de un árbol genealógico que permitirá al usuario seleccionar un padre y luego ir introduciendo los hijos de cualquiera de los padres existentes en el programa. A su vez también se podrá eliminar cualquier nodo. El programa también permite guardar los datos introducidos en un documento de texto, imprimir el árbol guardado y salir del sistema.

5.3 Diseño inicial

El programa está formado por cuatro componentes: la clase árbol, la clase hoja, la clase test y la interfaz grafica de usuario (GUI).

En un principio el programa se ejecutaba a través de la clase test (donde se encuentra el método main). Por lo que por consola se mostraba un pequeño menú con cuatro opciones: añadir, eliminar, guardar, imprimir y salir.

Si se escribe la opción ***añadir*** se pedirá el nombre del hijo que se quiere añadir y el nombre del padre al que se quiere añadir ese nuevo hijo. Si el nombre del padre es igual a "null", es que este hijo va a ser un nuevo nodo raíz y si no se irá comparando el nombre del padre introducido por teclado con todos los nombres de los nodos que haya en el documento de texto guardado "read.txt" y cuando lo encuentre se insertara el nuevo hijo a ese padre.

Abajo se muestra el método para que se pueda entender mejor:

```
public Leaf addNewChild(String childName, String parentName, Leaf
       rootNode){

       //se da el número de hijos de ese nodo
       for(int i=0; i<rootNode.getChildren().size(); i++){

              // se recorren todos los hijos de esa hoja y en el nodo
              //en el que este en ese momento será el padre.
              Leaf parentNode = rootNode.getChildren().get(i);

              // se comparan los nombres
              if(parentNode.getName().equals(parentName)){
```

```
                    // si es el nombre que el programa está buscando,
                    //inserta el nodo
                    parentNode.insertChild(new Leaf(childName,
                    parentNode));

                    //se encuentra un nodo con el mismo nombre que el
                    //usuario ha introducido
                    control = true;


                    // en el caso de que el usuario ponga más de un nodo
                    //con el mismo nombre, el programa solo añade el hijo
                    //en el primer nodo. Otra posibilidad es quitar el
                    //"break" y el programa añade el nuevo hijo a todos
                    //los padres con el mismo nombre.
                    break;

             }//cierro if

             //si el nodo tiene hijos, el programa llama al método
             //recursivo
             if (parentNode.getChildren().size() != 0){
                    addNewChild(childName, parentName, parentNode);
             }

      }//cierro for
      return rootNode;

}//cierro método addNewChild
```

Si la opción es ***eliminar*** el programa pedirá el nombre del nodo a eliminar y se irá comparando con cada uno de los hijos del árbol .Si corresponde con uno de los nodos guardados en el documento de texto se eliminara dicho nodo introducido.

Abajo se muestra el método para su mejor compresión:

```
//metodo que elimina el nombre de nodo que se le pasa
public void removeChild(String Node, Leaf rootNode){

      //se da el número de hijos de este subárbol
      for(int i=0; i<rootNode.getChildren().size(); i++){

             //se recorre uno a uno los nodos del árbol
             Leaf parentNode = rootNode.getChildren().get(i);

             // se comparan los nombres
             if(parentNode.getName().equals(Node)){

                    // si es el nombre que el programa está buscando,
                    //elimina el nodo
                    rootNode.getChildren().remove(i);

                    // se encuentra un nodo con el mismo nombre que el
                    //usuario ha introducido
                    control = true;
                    break;


             }
             //llamada al metodo recursivo
             removeChild(Node,parentNode);

      }//cierra for
```

```
}//cierra removeChild()
```

Si la opción es ***guardar*** se almacenara en un documento de texto ese nuevo padre e hijo introducido anteriormente en la opción añadir o si se ha eliminado algún nodo también se modificara el documento de texto “read.txt”.

En la Figura 5.3.1 se muestra un ejemplo de lo que almacena el archivo “read.txt”:

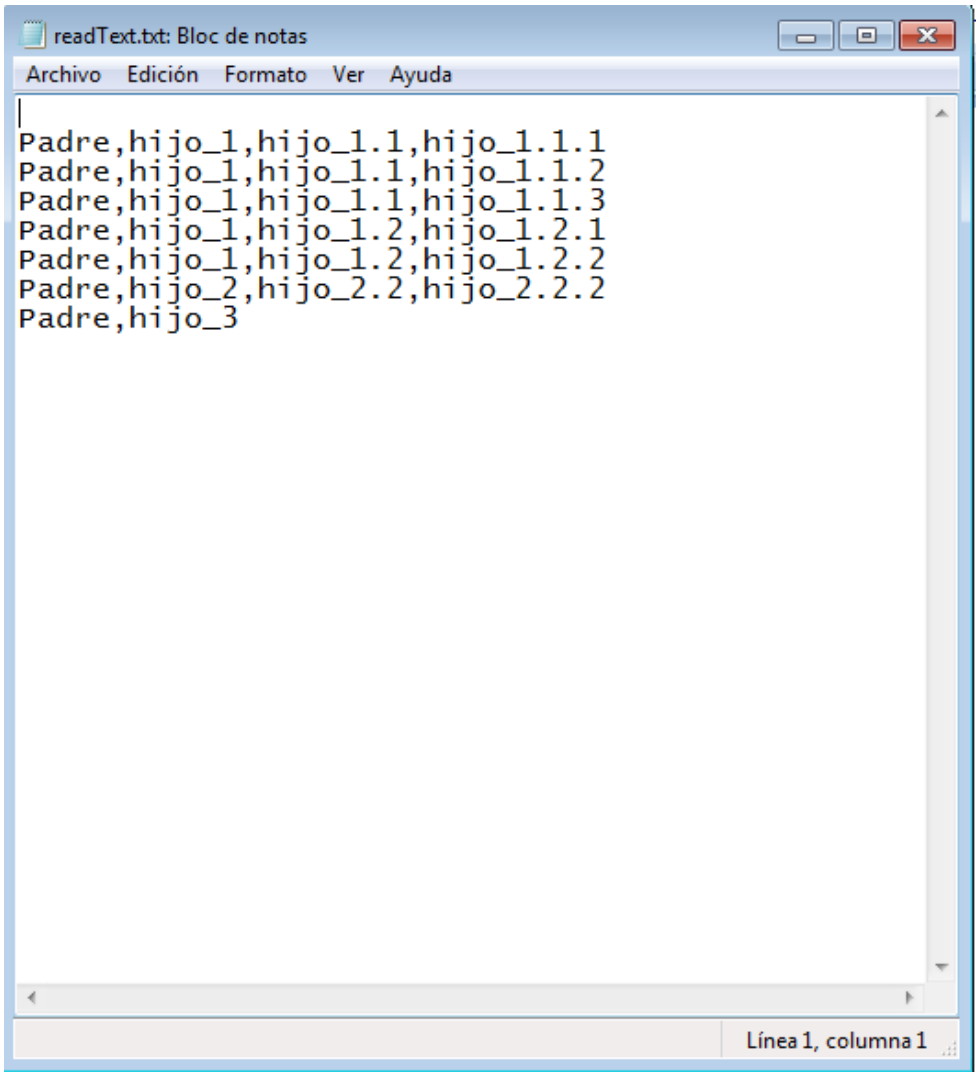


Figura 5.3.1. Ejemplo archivo “read.txt”.

Si la opción es ***imprimir*** se mostrara por consola el documento de texto donde se guardan los datos que se introducen por teclado pero en forma de árbol. El archivo que lo muestra así es el “out.txt”.

```
//este método visualiza el árbol
public void printChildren(Leaf rootNode, Boolean command){
	String tab = "";
	for(int i=0;i<rootNode.getChildren().size(); i++){
		tab = "";

		//con este bucle se puede ver la cantidad de nivel de
		//profundidad que hay.
		for(int j=0; j<rootNode.rowofChild(rootNode); j++){

			//el programa añade una nueva tabulación para cada
			//nivel
			tab += '\t';
		}

		String str = tab + rootNode.getChildren().get(i).getName();

		//si command es “true” actualiza el búfer que luego se
		//utiliza para imprimir el archivo out.txt
		if (command){

			//el programa añade las líneas al búfer que escribe
			//el árbol
			textFile.append(str + " " +
			System.getProperty("line.separator"));


		//si command es “false” el programa actualiza el búfer que
		//más tarde muestra el árbol
		}else{
			System.out.println(str);
		}

		//si el nodo tiene hijos, el programa llama de nuevo al
		//metodo recursivo
		if (rootNode.getChildren().get(i).getChildren().size() != 0){


			printChildren(rootNode.getChildren().get(i),command);

		}else{
			// el programa obtiene la línea para añadir en el
			//búfer del archivo readText.txt
			firstFile.append(setLineOfFirstFile(rootNode.
			getChildren().get(i))+
			System.getProperty("line.separator"));

		}
	}//cierra for
}//cierra metodo printChildren()
```

En la siguiente Figura 5.3.2 se muestra un ejemplo de lo que almacena el archivo “out.txt”:

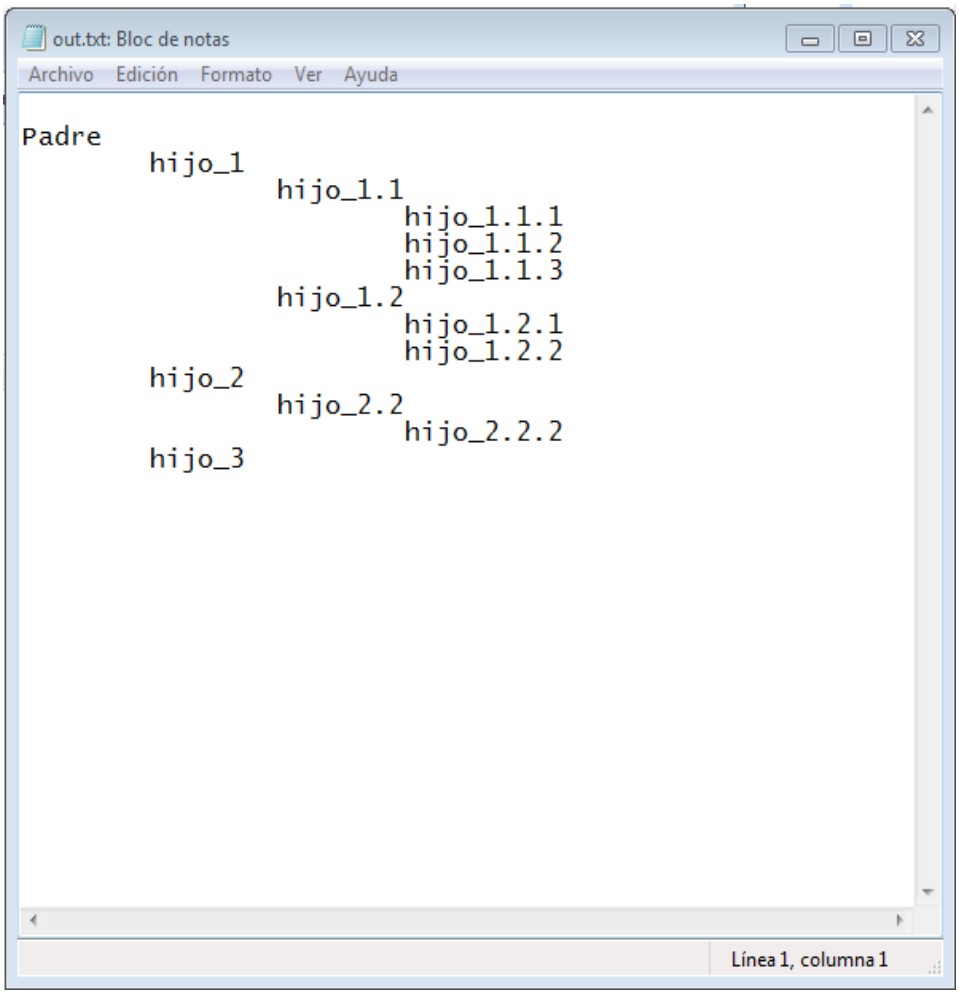


Figura 5.3.2. Ejemplo archivo “out.txt”.

La última opción es ***salir*** que cerrara el programa y se esperara a que el usuario lo vuelva a ejecutar.

5.4 Implementación de la interfaz grafica

Para diseñar la interfaz grafica que visualice de la forma más clara y sencilla el árbol genealógico se crea una ventana principal donde hay un área de texto donde se mostrara el árbol y unos botones que serán añadir, eliminar y salir. En la Figura 5.4.1 se muestra como quedaría dicha interfaz.

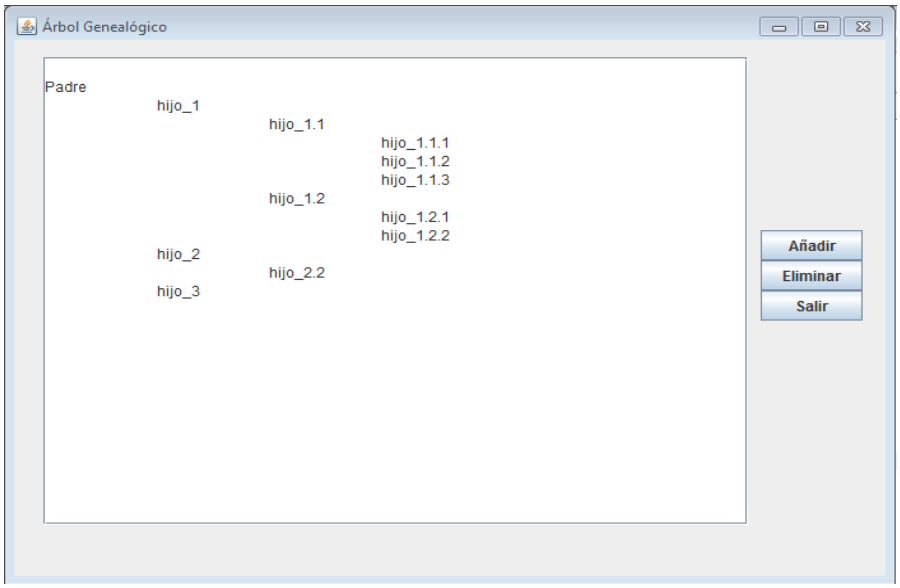


Figura 5.4.1. Ejemplo de la interfaz grafica

Si se pulsa el botón **añadir** se abrirá una nueva ventana donde se podrá introducir el nombre del hijo y se mostrara un desplegable con todos los nodos almacenados en el archivo "read.txt" para poder seleccionar el padre al que se quiere añadir ese hijo.

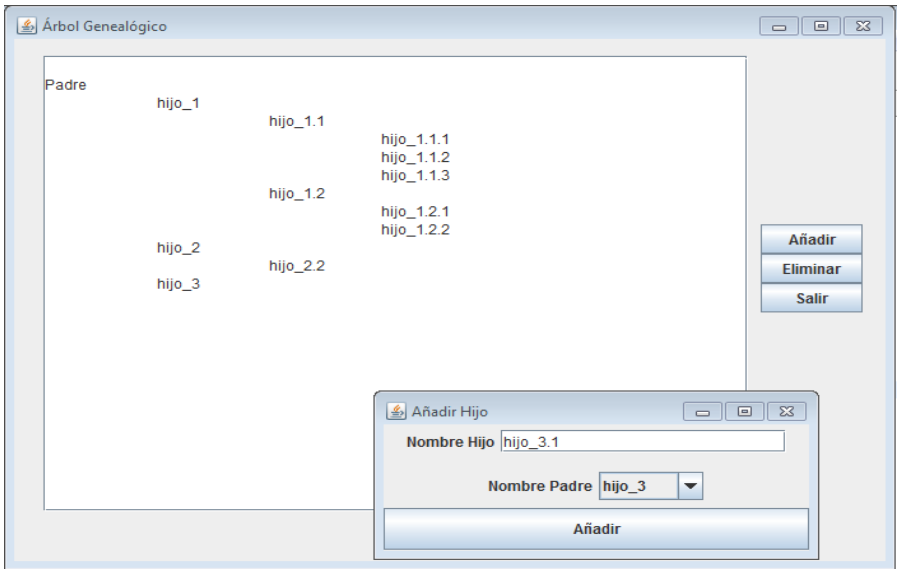


Figura 5.4.2. Ejemplo de la ventana que se muestra al pulsar el botón añadir.

Cuando se pulse el botón “Add” estos datos serán guardados en un documento de texto y a su vez el árbol se imprimirá en el área de texto de la ventana principal en forma de árbol.

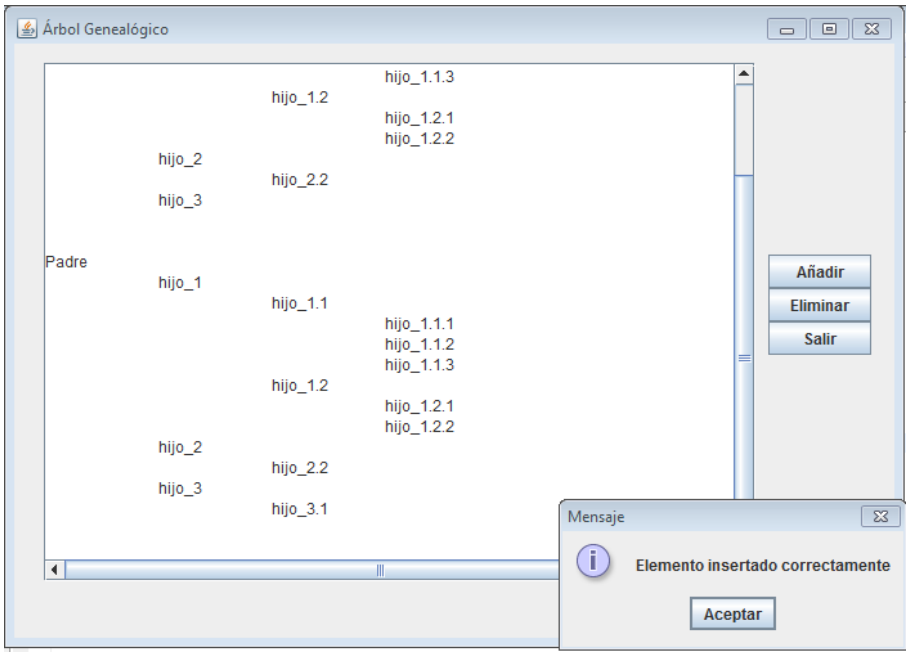


Figura 5.4.3. Ejemplo de la ventana que se muestra al añadir un hijo.

Si el botón seleccionado es **eliminar** se abrirá una nueva ventana donde se volverá a mostrar un desplegable con todos los nodos almacenados en el archivo “read.txt”. Entonces se elegirá el nodo a eliminar.

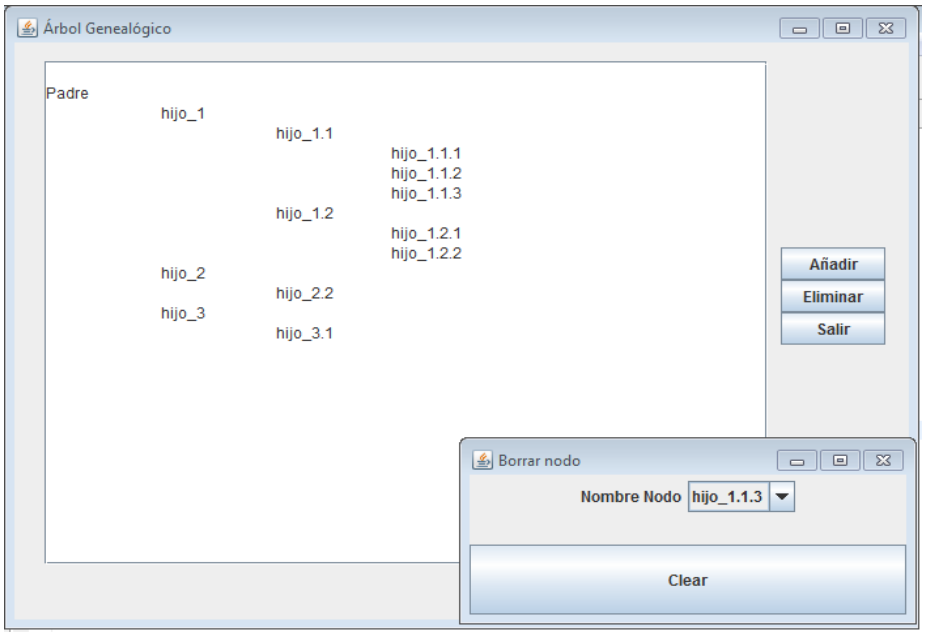


Figura 5.4.4. Ejemplo de la ventana que se muestra al pulsar el botón eliminar.

Cuando se pulse el botón “Clear” el nodo seleccionado se eliminara, se guardara dicho cambio y el árbol será mostrado en la ventana principal de nuevo.

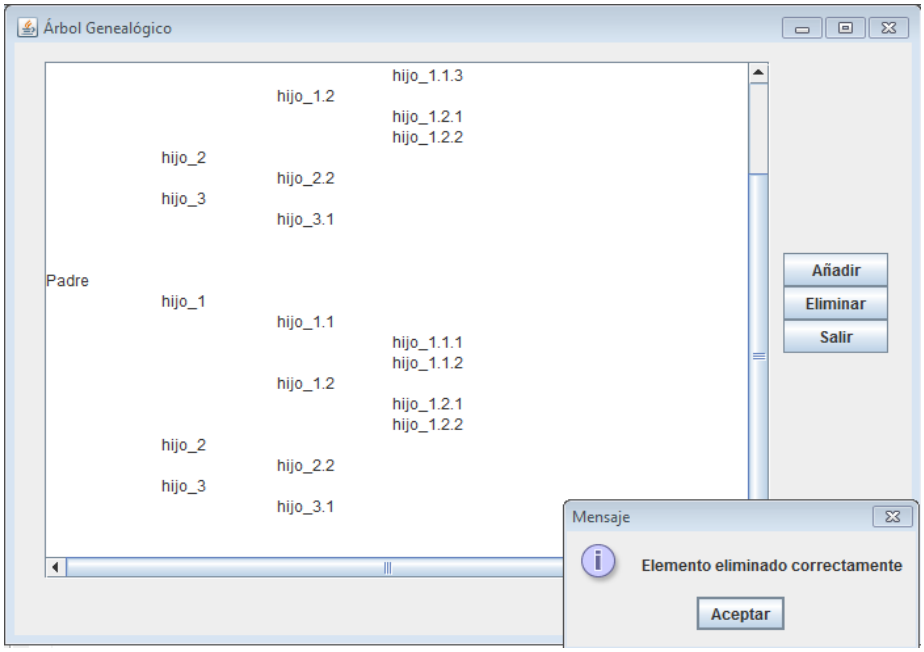


Figura 5.4.5. Ejemplo de la ventana que se muestra al borrar un hijo.

Al pulsar el botón de **salir** se abrirá una nueva ventana con un pequeño menú donde se le preguntara al usuario si está seguro de querer salir del programa. Si se pulsa que “Si” el programa se cerrara y si es que “No” el programa permanecerá abierto.

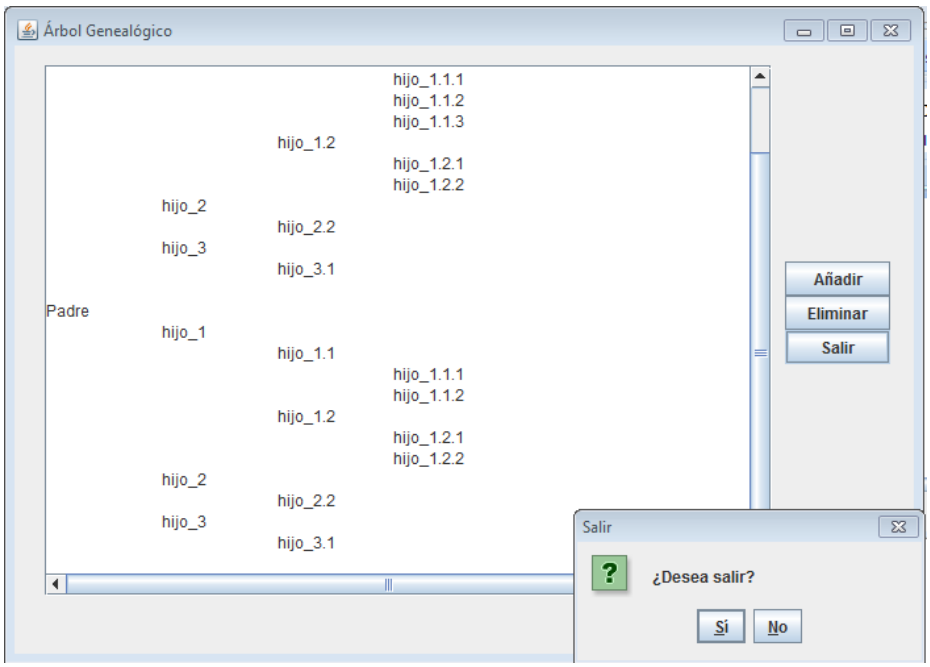


Figura 5.4.4. Ejemplo de la ventana que se muestra al pulsar el botón salir.

6. Conclusiones

Desde el comienzo de este proyecto se han intentado trazar los objetivos exigidos a fin de que la ejecución sea clara y viable.

Este proyecto puede ser útil en múltiples áreas ya que los arboles son estructuras compuestas por nodos, utilizadas para representar datos de forma jerárquica entre los elementos que la conforman y que son utilizados para la solución de una gran variedad de problemas, ya que pueden implementarse fácilmente en un ordenador.

Además el uso de arboles permite a los usuarios mejorar o modificar el programa de acuerdo a las necesidades que se tengan haciéndolo ya sea, con una interfaz mucho más dinámica a la ya existente o implementándole mejoras que ninguna otra aplicación antes tuviera de acuerdo a sus propiedades.

Por lo tanto, la realización de este proyecto ha sido interesante y me ha enriquecido como programadora.

7. Anexos

El programa se divide en cuatro clases java, que son: Leaf.java, Tree.java, Test.java y Teclado.java. Como resultado de la ejecución del programa se crean dos archivos, out.txt y readText.txt.

Para una mejor comprensión del código y los archivos resultantes se comentaran cada uno de ellos.

- La clase ***Leaf.java*** crea cada una de las hojas que más tarde formaran el árbol.

```
import java.util.ArrayList;
import java.util.Iterator;


public class Leaf {

	private String name;
	private Leaf parent;

	//array de hijos
	public ArrayList<Leaf> children=null;

	//constructor por defecto
	//un hijo tiene un nombre y un padre
	public Leaf(String name, Leaf parent){
		this.name = name;
		this.parent = parent;
		children = new ArrayList<Leaf>();
	}
```

```
//se da el nombre
public String getName() {
	return name;
}

//se cambia el nombre
public void setName(String name) {
	this.name = name;
}

//se da el padre
public Leaf getParent() {
	return parent;
}

//se cambia el nombre del padre
public void setParent(Leaf parent) {
	this.parent = parent;
}


//se da la lista de los nombres de los hijos
public ArrayList<String> getChildren2() {

	//se crea una nueva lista con los nombres de los hijos
	ArrayList<String> aux= new ArrayList<String>();

	if(!name.equals("RootNode"))aux.add((String)name);
		if(children==null){
			return aux;
		}else{
		//se crea un iterador para poder ir recorriendo todos los
		//elementos del array
		Iterator pi = children.iterator();

		//mientras que ese nodo tenga hijos se pasa al siguiente
		while(pi.hasNext()){
			// se recorren los hijos que tenga
			ArrayList<String>
			aux2=((Leaf)pi.next()).getChildren2();

			//se vuelve a recorrer esa lista
			Iterator pi2 = aux2.iterator();
			while(pi2.hasNext()){
				aux.add((String)pi2.next());
			}
		}
		return aux;
	}
}
//se da la lista de hijos
public ArrayList<Leaf> getChildren() {
	return children;
}

public void setChildren(ArrayList<Leaf> children) {
	this.children = children;
}
//se inserta un hijo en el nodo que se le pasa
public void insertChild(Leaf node) {

	// si este hijo no contiene el nodo que se le está pasando se le
	//añade en la siguiente línea
	if (!this.children.contains(node)){
		this.children.add(node);

	}
}
```

```
	//se recorre por filas hijo a hijo
	public int rowofChild(Leaf node){
		int row = 0;
		while (node.name!= "RootNode"){
			row +=1;
			node = node.parent;
		}
		return row;
	}

	//se da la cantidad de hijos que tiene un nodo
	public int ContainsNode(Leaf node){
		for(int i=0; i<this.children.size(); i++){

			//se compara si el nombre de ese hijo es igual al nombre
			//del nodo que te pasa
			if(this.children.get(i).getName().equals(node.getName())){
				return i;
			}
		}
		return -1;
	}

}//cierra clase Leaf
```

- La siguiente clase es ***Tree.java***, es la aplicación que permite la creación del árbol usando la clase Leaf.java.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;


public class Tree {

	// se controla si el usuario usa un padre que ya existe
	private boolean control;

	// se escribe la estructura árbol en el archivo out.txt
	private StringBuffer textFile = new StringBuffer();

	// se escribe la estructura árbol en el archivo readText.txt
	private StringBuffer firstFile = new StringBuffer();

	//constructor por defecto
	public Tree(){}

	public boolean isControl() {
		return control;
	}

	public void setControl(boolean control) {
		this.control = control;
	}

	//se da el archivo out.txt
	public StringBuffer getTextFile() {
		return textFile;
	}
```

```
//se da el archivo read.txt
public StringBuffer getFirstFile() {
	return firstFile;
}

public Leaf addNewChild(String childName, String parentName, Leaf
rootNode){

	//se da el número de hijos de ese nodo
	for(int i=0; i<rootNode.getChildren().size(); i++){

		//se recorren todo los hijos de esa hoja y en la posición
		//en la que este en ese momento será el padre
		Leaf parentNode = rootNode.getChildren().get(i);

		//se comparan los nombres
		if(parentNode.getName().equals(parentName)){

			// si es el nombre que el programa está buscando,
			//inserta el nodo
			parentNode.insertChild(new Leaf(childName,
			parentNode));

			// se encuentra un nodo con el mismo nombre que el
			//usuario ha introducido
			control = true;


			// en el caso de que el usuario ponga más de un nodo
			//con el mismo nombre,el programa solo añade el hijo
			//en el primer nodo. Otra posibilidad es quitar el
			//"break" y el programa añade el nuevo hijo a todos
			//los padres con el mismo nombre.
			break;

		}
		//si el nodo tiene hijos, el programa llama al método
		//recursivo
		if (parentNode.getChildren().size() != 0){
			addNewChild(childName, parentName, parentNode);
		}
	}
	return rootNode;
}

// método que elimina el nombre de nodo que se le pasa
public void removeChild(String Node, Leaf rootNode){

	//se da el número de hijos de este subárbol
	for(int i=0; i<rootNode.getChildren().size(); i++){

		//se recorre uno a uno los nodos del árbol
		Leaf parentNode = rootNode.getChildren().get(i);




		// se comparan los nombres
		if(parentNode.getName().equals(Node)){

			// si es el nombre que el programa está buscando,
			//elimina el nodo
			rootNode.getChildren().remove(i);

			// se encuentra un nodo con el mismo nombre que el
			//usuario ha introducido
			control = true;
```

```
                    break;


                }
                //llamada al método recursivo
                removeChild(Node,parentNode);

        }//cierra for

}//cierra removeChild()


//este método visualiza el árbol
public void printChildren(Leaf rootNode, Boolean command){
        String tab = "";
        for(int i=0;i<rootNode.getChildren().size(); i++){
                tab = "";

                //con este bucle se puede ver la cantidad de nivel de
                //profundidad que hay.
                for(int j=0; j<rootNode.rowofChild(rootNode); j++){

                        //el programa añade una nueva tabulación para cada
                        //nivel
                        tab += '\t';
                }

                String str = tab + rootNode.getChildren().get(i).getName();

                //si command es “true” actualiza el búfer que luego se
                //utiliza para imprimir el archivo out.txt
                if (command){

                        //el programa añade las líneas al búfer que escribe
                        //el árbol
                        textFile.append(str + " " +
                        System.getProperty("line.separator"));


                //si command es “false” el programa actualiza el búfer que
                //más tarde muestra el árbol
                }else{
                        System.out.println(str);
                }

                //si el nodo tiene hijos, el programa llama de nuevo al
                //metodo recursivo
                if (rootNode.getChildren().get(i).getChildren().size()!=0){



                        printChildren(rootNode.getChildren().get(i),
                        command);


                }else{
                        // el programa obtiene la línea para añadir en el
                        //búfer del archivo readText.txt
                        firstFile.append(setLineOfFirstFile(rootNode.
                        getChildren().get(i))
                        + System.getProperty("line.separator"));

                }
        }//cierra for

}//cierra metodo printChildren()
```

```
// método para ir guardando en el archivo de texto el árbol
public void saveTreeStructureToFile(Leaf rootNode, String filename,
StringBuffer strBuffer){
	try{
	    //se crea el archivo
	    FileWriter fstream = new FileWriter(filename);
	    BufferedWriter out = new BufferedWriter(fstream);
	    out.write(strBuffer.toString());
	    out.close();
	}catch (Exception e){
	    System.err.println("Error: " + e.getMessage());
	}
}

//se obtiene la línea que se añade en el archivo readText.txt con los
//predecesores del nodo
public String setLineOfFirstFile(Leaf node){

	String line = "";
	//se crea un objeto ArrayList
	ArrayList<String> lineList = new ArrayList<String>();

	//se continua mientras el bucle no obtenga todos los predecesores
	while (node.getName()!= "RootNode"){
		lineList.add(node.getName());
		node = node.getParent();
	}

	//el bucle obtiene los nombres de los nodos desde el último hasta
	//el primero. Es necesario invertirlo.
	Collections.reverse(lineList);

	for(int i=0; i< lineList.size();i++){
		line += lineList.get(i) + ",";
	}

	//se coge la línea sin la última coma
	return line.substring(0, line.length()-1);

}

//se devuelve la hoja raíz
public Leaf sendRootNode(){
	File file = new File("readText.txt");
	BufferedReader reader = null;

	//se crea una nueva hoja, que será la raíz
	Leaf root = new Leaf("RootNode", null);

	//se hace una copia para ejecutar el árbol
	Leaf broot = root;
	try {
		reader = new BufferedReader(new FileReader(file));
		String line = null;



		// se repite hasta que todas las líneas son leídas
		while ((line = reader.readLine()) != null) {
			String []fields = line.split(",");
			for(int i =0;i<fields.length;i++){

				//se crea una nueva hoja
				Leaf leaf = new Leaf(fields[i], broot);

				//containsIndex = obtiene el índice de la
				//hoja hijo
```

```
                                    int containsIndex = broot.ContainsNode(leaf);
                                    //se comprueba si el padre contiene esta hoja

                                    if (containsIndex != -1){
                                          //si el padre contiene este hijo, a
                                          //continuación se busca la fila de ese
                                          //hijo, y temporalmente hace que sea la
                                          //nueva raíz
                                          broot =
                                          broot.getChildren().get(containsIndex);

                                    }else{


                                          //si el padre no contiene esta hoja,
                                          //entonces se añade esta hoja como hijo
                                          //de este padre
                                          broot.insertChild(leaf);

                                          //esta hoja es el nuevo padre
                                          broot = leaf;

                                    }

                              }//cierra for

                              //para la nueva línea del archivo, se comienza desde
                              //el principio a hacer el mismo proceso
                              broot = root;

                        }//cierra while

                  } catch (FileNotFoundException e) {
                        System.out.println("El archivo no ha sido encontrado");

                  } catch (IOException e) {
                        System.out.println("Ha ocurrido un error");

                  } finally {
                  try {
                     if (reader != null) {
                        reader.close();
                     }
                  } catch (IOException e) {
                        e.printStackTrace();
                  }
            }//cierra primer try
            return root;

      }//cierra método

}//cierra clase
```

- La clase ***Test.java*** es donde está el método main.

```
public class test {

      public static void main (String args[]) {
            String child = null;
            String parent = null;
            String node = null;
```

```
//las opciones pueden se añadir,guardar,imprimir y salir
String command = null;

System.out.println("Bienvenido\n\n");
Tree tree = new Tree();
Leaf root = tree.sendRootNode();
do{
	System.out.println("Introduce el
	comando(añadir,eliminar,imprimir,guardar,salir): ");
	command = Teclado.readString();
	if(command.equals("añadir")){
		System.out.println( "Inserta el nombre del hijo: "
		);
		child = Teclado.readString();
		System.out.println( "Inserta el nombre del padre: "
		);
		parent = Teclado.readString();

		//si el archivo readText.txt esta vacio
		if (parent.equals("null")){


			//se inserta nuevo hijo
			root.insertChild(new Leaf(child, root));
			tree.setControl(true);

		//si el archivo readText.txt no está vacío el
		//programa ha de buscar al padre e insertar al hijo
		}else{
			root = tree.addNewChild(child, parent,root);
		}
		if (tree.isControl()){
			System.out.println("El nodo ha sido añadido
			satisfactoriamente...");
			tree.setControl(false);
		}else{
			System.out.println("El padre no ha sido
			encontrado!!!");
		}
	}
	else if (command.equals("imprimir")){

		// se va a la primera línea
		tree.getTextFile().setLength(0);
		tree.getFirstFile().setLength(0);
		tree.printChildren(root, false);

	}else if(command.equals("guardar")){
		tree.getTextFile().setLength(0);
		tree.getFirstFile().setLength(0);
		tree.printChildren(root, true);

		//se escribe el árbol estructura en el archivo
		//out.txt
		tree.saveTreeStructureToFile(root, "out.txt",
		tree.getTextFile());



		//se escribe el árbol estructura en el archivo
		//readText.txt

		tree.saveTreeStructureToFile(root, "readText.txt",
		tree.getFirstFile());

	}else if(command.equals("eliminar")){
		System.out.println( "Inserta el nombre del nodo a
		borrar: " );
```

```
                        node = Teclado.readString();
                        tree.removeChild(node,root);

                        if (tree.isControl()){
                              System.out.println("El nodo ha sido
                              eliminado...");
                              tree.setControl(false);
                        }else{
                              System.out.println("El nodo no ha sido
                              encontrado!!!");
                        }

                  }
            }while((!command.equals("salir")) && (!command.equals("" +"")));
            System.out.println("El programa ha sido cerrado. Vuelva a
            ejecutarlo");

      }//cierra do

}//cierra clase test
```

- La clase ***GUI.java*** es la clase donde se crea la interfaz grafica.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.util.*;
import java.io.IOException;
import javax.swing.tree.*;
public class GUI extends JFrame implements ActionListener{



      private JPanel principal,arbol,botones,h,pad,n;
      private JButton añadir,eliminar,guardar,imprimir,salir;
      private JFrame addHijo,clearNodo;
      private JButton add,clear;
      private JComboBox padres, nodos;//para crear un desplegable
      private JTextArea area;
      private JTextField hijo,nodo;
      private JLabel lh,lp,ln;
      int res;


      private String nombreH;
      private String nombreP;
      private String nombreN;
      private String command = null;
      private Tree tree ;
      private Leaf root;



      //constructor por defecto
      public GUI(){
            super ("Árbol Genealógico ");
            initialize();
      }
```

```
public static void main (String args[]){
       GUI frame = new GUI();
       frame.setSize(700,500);
       frame.setVisible(true);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

}//cierra main


private void initialize(){



      //se obtiene el contenedor asociado a la ventana
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      //se crean los paneles
      principal = new JPanel();
      arbol = new JPanel();

      //se crea el bordeado del área de texto
      Box cuadro = Box.createHorizontalBox();
      area = new JTextArea(25,50);
      area.setEditable(false);

      cuadro.add(new JScrollPane(area));
      arbol.add(cuadro);

      //se crean los botones y se añaden  a los escuchadores
      botones = new JPanel(new GridLayout(4,1));
      añadir = new JButton("Añadir");
      añadir.addActionListener(this);

      //guardar = new JButton("Guardar");
      //guardar.addActionListener(this);

      eliminar = new JButton("Eliminar");
      eliminar.addActionListener(this);

      //imprimir = new JButton("Imprimir");
      //imprimir.addActionListener(this);

      salir = new JButton("Salir");
      salir.addActionListener(this);

      botones.add(añadir);
      botones.add(eliminar);
      //botones.add(imprimir);
      //botones.add(guardar);
      botones.add(salir);


      principal.add(arbol);
      principal.add(botones);
      c.add(principal);

      tree = new Tree();
      root = tree.sendRootNode();

}//cierra método inicializa


//método añadir nodo
public void añadir(){
```

```
        addHijo = new JFrame("Añadir Hijo");
        addHijo.setSize(350,150);
        addHijo.setDefaultCloseOperation(addHijo.EXIT_ON_CLOSE);
        addHijo.setVisible(true);
        addHijo.setVisible(true);
        addHijo.setLayout(new GridLayout(3,1));

        h = new JPanel();
        lh = new JLabel ("Nombre Hijo");
        hijo = new JTextField(20);
        h.add(lh);
        h.add(hijo);


        //Obtener los padres existentes
        //se coge un array con todos los nodos que cuelgan de la raíz
        ArrayList<String> p = root.getChildren2();

        //se crea un iterador para poder ir recorriendo todos los
        //elementos del array
        Iterator pi = p.iterator();

        String []ps = new String[p.size()+1];
        ps[0] = "Root";
        int i = 1;

        //se pasa al siguiente
        while(pi.hasNext()){
               ps[i] = ((String)pi.next());
               i++;

        }

        pad = new JPanel();
        padres = new JComboBox(ps);
        lp = new JLabel("Nombre Padre");
        pad.add(lp);
        pad.add(padres);
        addHijo.add(h);
        addHijo.add(pad);
        add = new JButton("Añadir");
        add.addActionListener(this);
        addHijo.add(add);



}




//método eliminar nodo
public void eliminar(){

        clearNodo = new JFrame("Borrar nodo");
        clearNodo.setSize(350,150);
        clearNodo.setDefaultCloseOperation(clearNodo.EXIT_ON_CLOSE);
        clearNodo.setVisible(true);
        clearNodo.setVisible(true);
        clearNodo.setLayout(new GridLayout(2,1));

        n = new JPanel();

        //se obtienen los padres existentes
        //se coge un array con todos los nodos que cuelgan de la raíz
        ArrayList<String> p = root.getChildren2();
```

```
	//se crea un iterador para poder ir recorriendo todos los
	//elementos del array
	Iterator pi = p.iterator();
	String []ps = new String[p.size()+1];
	ps[0] = "Root";
	int i = 1;

	//se pasa al siguiente
	while(pi.hasNext()){
		ps[i] = ((String)pi.next());
		i++;

	}

	nodos = new JComboBox(ps);
	ln = new JLabel ("Nombre Nodo");
	n.add(ln);
	n.add(nodos);

	clearNodo.add(n);
	clear = new JButton("Clear");
	clear.addActionListener(this);
	clearNodo.add(clear);


}

//método imprimir en la clase Tree.java
public void crearArbol(Leaf root, Boolean command){
	String tab = "";
		for(int i=0;i<root.getChildren().size(); i++){
			tab = "";

			//con este bucle se puede ver la cantidad de nivel
			//de profundidad que hay.
			for(int j=0; j<root.rowofChild(root); j++){

				//el programa añade una nueva tabulación para
				//cada nivel
				tab += '\t';
			}
			String str = tab +
			root.getChildren().get(i).getName();

			//si comando es “true” actualiza el búfer que luego
			//se utiliza para imprimir el archivo out.txt




			if (command){

				//el programa añade las líneas al búfer que
				//escribe el árbol
				tree.getTextFile().append(str + " " +
				System.getProperty("line.separator"));

			//si el comando es “false” el programa actualiza el
			//búfer que más tarde muestra el árbol
			}else{

				//para que te muestre el árbol sin que se
				//pueda modificar
				area.append(str+"\n");
			}
```

```
                        //si el nodo tiene hijos, el programa llama de nuevo
                        //al método recursivo
                        if (root.getChildren().get(i).getChildren().size()
                        != 0){

                              crearArbol(root.getChildren().
                              get(i),command);
                        }else{

                        //el programa obtiene la línea para añadir en el
                        //búfer del archivo readText.txt



                        tree.getFirstFile().append(tree.setLineOfFirstFile
                        (root.getChildren().get(i)) +
                        System.getProperty("line.separator"));

                        }
                  }//for

}//cierra método crearArbol()



public void guardar(){
      System.out.println("guardando");
      tree.getTextFile().setLength(0);
      tree.getFirstFile().setLength(0);
      tree.printChildren(root, true);

      //se escribe el árbol estructura en el archivo out.txt
      tree.saveTreeStructureToFile(root, "out.txt",
      tree.getTextFile());

      //se escribe el árbol estructura en el archivo readText.txt
      tree.saveTreeStructureToFile(root, "readText.txt",
      tree.getFirstFile());
}

 public void salir(){
      res = JOptionPane.showConfirmDialog(null, "¿Desea salir?",
      "Salir", JOptionPane.YES_NO_OPTION);
      if (res==JOptionPane.YES_OPTION){
            System.exit(0);

      }
 }


public void actionPerformed (ActionEvent ae){

      if (ae.getSource()== añadir){

            añadir();
      }

      //este boton es el "OK" de la segunda ventana
      if (ae.getSource()== add){
             nombreH = hijo.getText();
             nombreP = padres.getSelectedItem().toString();

             if(nombreP == "Root"){
                  root.insertChild(new Leaf(nombreH, root));
             }
             else{
                  root =tree.addNewChild(nombreH,nombreP,root);
             }
```

```
                        guardar();
                        crearArbol(root,false);

                        //cuando se pulsa el boton "OK" se guarda el hijo en
                        //el padre seleccionado y se borra lo que
                        //haya en la ventana para poder introducir un nuevo
                        //hijo
                        addHijo.dispose();
                        JOptionPane.showMessageDialog(addHijo,"Elemento
                        insertado correctamente");

                 }

                if (ae.getSource()== eliminar){

                        eliminar();
                }

                //este boton es el "OK" de la segunda ventana
                if (ae.getSource()== clear){

                        nombreN = nodos.getSelectedItem().toString();
                        tree.removeChild(nombreN, root);
                        guardar();
                        crearArbol(root,false);

                        //cuando se pulsa el boton "OK" se elimina el nodo
                        //introducido y se borra lo que haya en la ventana para
                        //poder introducir un nuevo nodo a eliminar
                        clearNodo.dispose();
                        JOptionPane.showMessageDialog(clearNodo,"Elemento eliminado
                        correctamente");

                 }

                /*
                if (ae.getSource() == guardar){

                        guardar();//este boton se quitara de la interfaz

                 }
                */

                /*
                if (ae.getSource() == imprimir){

                        //area.setText("Arbol Genealogico\n");
                        crearArbol(root,false);
                }
                */

                if (ae.getSource() == salir){

                        salir();//este boton se quitara de la interfaz

                }
        }//cierra metodo actionPerformed

}//cierra clase
```

8. Bibliografía y referencias

1. “Introducción a la programación con orientación a objetos”. Camelia Muñoz Caro, Alfonso Niño Ramos y otros. Prentice Hall.

2. La Biblia de Java. Schildt, Herbert. Anaya

3. Java. Cómo programar. Dietel Harvey M., Deitel Paul J. Pearson Addison-Wesley.

4. Estructuras de datos en Java. John Lewis y Joseph Chase. Pearson Addison-Wesley.

5. Estructura de datos y algoritmos en Java. Adam Drozdek. Thomson.

6. An Introduction to Data Structures and Algorithms with Java, Rowe, Glen; Prentice Hall.

7. Estructura de datos, especificación, diseño e implementación. Xavier Franch Gutiérrez. Ediciones UPC.

8. Fundamentos de estructuras de datos, Soluciones en Ada, Java y C++. Zenón José Hernández Figueroa y otros. Thompson

9. Estructura de datos, algoritmos y programación orientada a objetos. Gregory L. Heileman. McGraw-Hill.