

Universidad ORT

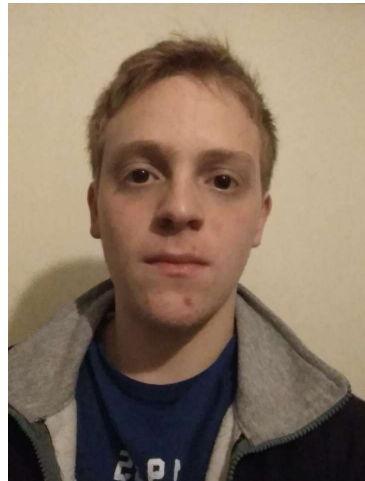
Facultad de Ingeniería

Algoritmos I

Obligatorio I - Grupo M3A



Joaquín Lamela (233375)



Martín Gutman (186783)

Junio de 2019

Índice

1.	Análisis y Diseño del sistema.....	3
2.	Implementaciones escogidas.....	5
3.	Conclusiones.....	8
4.	Memoria de las tareas desarrolladas por cada integrante del grupo.....	9

Declaración de autoría

Nosotros, Joaquín Lamela y Martín Gutman con los números de estudiante 233375 y 186783, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

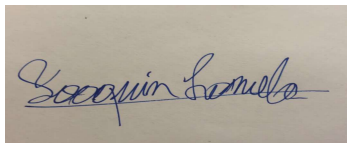
1. Esta obra fue producida durante el plazo de entrega del Obligatorio de estructura de datos y Algoritmos 1.
2. En este caso no hemos consultado material de otros, de tal manera que todo lo presentado es de nuestra autoría.
3. Por otra parte no hay citas presentes debido a que lo realizado es meramente realizado por nosotros.

En la obra, hemos acusado recibo de las ayudas recibidas;

4. Hemos concurrido a ayudantía para resolver errores de implementación, recibiendo la ayuda del ayudante de Cátedra, Maximiliano García.
5. Ninguna parte de este trabajo ha sido publicada previamente a su entrega, debido a que por parte de la Cátedra no se ha solicitado.

[Joaquín Lamela]

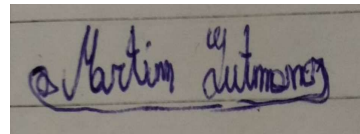
Firma



(16/06/19)

[Martín Gutman]

Firma

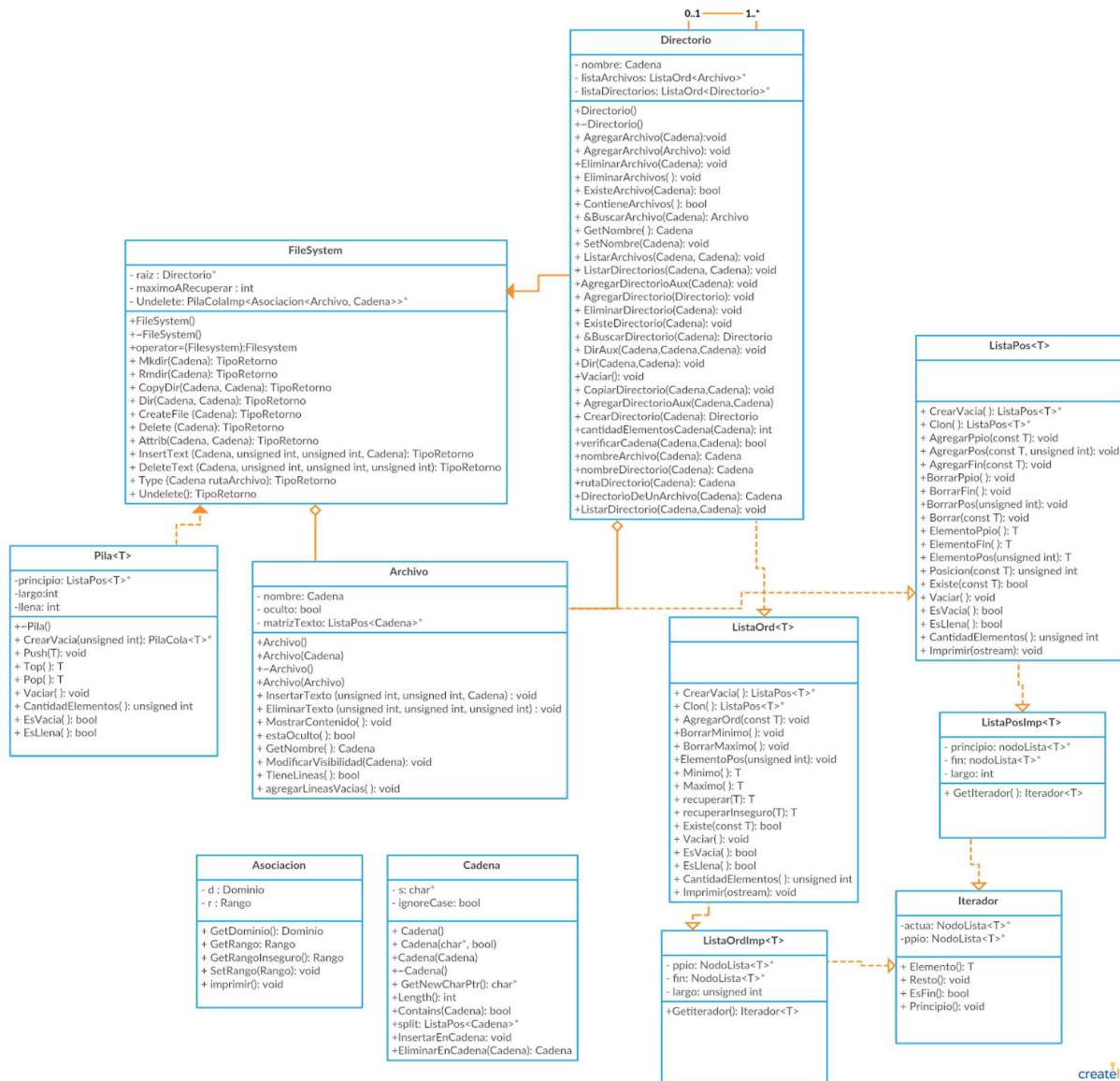


(16/06/19)

1. Análisis y Diseño del sistema

En esta sección mostraremos cómo fue nuestro diseño, a través del diagrama de clases. En este caso aparecen solamente las clases que nosotros utilizamos para nuestro diseño. De manera que debido a la implementación que nosotros realizamos hay algunas clases que no hubo necesidad de utilizarlas.

A continuación se presenta el diagrama de clases:



En dicho diagrama, se pueden observar los atributos y métodos de cada clase, de tal manera que deja expresado claramente el diseño que nosotros realizamos en él obligatorio. En nuestro caso, decidimos no utilizar los árboles generales debido a que sabíamos que posiblemente se nos iba a complicar a la hora de realizar las funciones de directorios. Es por esto que para nuestra implementación decidimos utilizar solamente los TAD's listas. De tal manera que en nuestra implementación los métodos de la clase Directorios, lo pasamos a la clase Directorio.

Ya que nos planteamos a nosotros, que sí hacíamos la clase Directorios, a la hora de encontrar las rutas se nos iba a complicar mucho más, que si solamente utilizamos la clase Directorio.

De tal manera que planeabamos que el FileSystem, fuera obviamente la raíz y que dicha raíz tenga una lista de Directorio y una lista de Archivos. De tal manera que cada Directorio tiene una lista de Directorio y una lista de Archivo.

Implementaciones para las clases por defecto:

Archivo: Para esta clase se usaron los siguientes atributos:

Se utilizaron: Un nombre de archivo, del tipo Cadena (clase implementada por la clase para su uso en el obligatorio), un atributo que indica si el archivo está o no oculto, y una ListaPos de Cadena, qué es la que lleva la información del texto que se inserta y modifica en el archivo. Esta implementación se realizó debido a que el acceso a las listas de texto y la modificación de las mismas se realiza de manera eficiente, y son los métodos que más utiliza esta clase.

Directorio: Se usaron las siguientes variables:

Se utilizaron: Un nombre del tipo cadena, una lista de directorio (lo que permite agregar directorios dentro de otros directorios), y una lista de Archivos, los cuales se implementaron con ListaOrd, debido a que se pide, al imprimirlos, que se haga de forma ordenada, ya que es una operación que se realiza de forma continua, y se realiza, gracias a esta implementación, de forma eficiente., desde la cual se manejan estos. Esta implementación nos permite generar el FileSystem a partir de listas de directorios, en el cual, el directorio padre, es el que contiene las listas de directorios(subdirectorios del mismo) y los archivos.

FileSystem:

Para esta parte del obligatorio, se usaron:

- Variables: Se usó un directorio raíz, que funciona como la carpeta madre del filesystem, una Cadena nombre, un int máximo a recuperar, para el undelete, y una pila (Stack), que almacena los archivos eliminados (utilizado para el undelete).
- Implementación: Para la implementación de esta clase, se usaron métodos que se habían construido en clase archivo y clase directorio. Se navegaba en el filesystem con un método de directorio llamado buscar ruta.

2. Implementaciones escogidas

En nuestro caso, los TAD que utilizamos fueron ListaOrd, ListaPos y PilaCola.

Para comenzar hablando, hablaremos que reutilizamos el TAD (ListaOrd) el cual ya estaba implementado por la cátedra, de tal manera que nosotros lo vimos viable utilizarlo debido a que para los directorios y los archivos hay que mostrarlos ordenados alfabéticamente. De tal manera que cuando fuésemos a realizar algunas de las funciones que están implementadas en la clase Directorio, con este TAD se nos simplificarán las operaciones. Un ejemplo concreto de esto es cuando nosotros agregamos un archivo, en este caso la manera más sencilla de hacerlo es utilizar el agregar ordenado. De tal manera que cuando vayamos a realizar la impresión por pantalla, se mostrará “directamente” ya que no va a haber que ordenarlo previamente. En este caso se ordena por el nombre del archivo.

Es por esto que decidimos utilizar el TAD ListaOrd para poder realizar las implementaciones correspondientes en la clase Directorio, ya que en ella utilizamos una lista ordenada de archivos y una lista ordenada de directorio. Ya que como mencionamos antes, nos es favorable a la hora de realizar las impresiones, ya que se ordena alfabéticamente.

Por otra parte utilizamos el TAD ListaPos, ya que este mismo nos convenía a la hora de implementar las operaciones en la clase Archivo. Esto debido a que si uno lee la letra detalladamente, se puede dar cuenta que cuando se inserta un texto, se hace en una determinada numero de línea, y también en una determinada de posición de línea. De tal manera que la forma más sencilla es utilizando una posición. Es por esto que lo que realizamos fue la implementación del TAD ListaPos. También lo que nos facilitó fue el eliminar texto, de tal manera que lo que nuevamente se elimina en una determinado numero de línea, también en una posición de línea y se eliminan una cantidad de k caracteres.

En este caso lo que se realizó fue la implementación planteada por la cátedra sin modificarlo. De tal manera que cuando nosotros lo implementamos dieran que las pruebas estuvieran OK. La implementación que se usó en este caso, fue a través de una clase ya implementada y usada en el curso, NodoLista no acotada (sin ser por la memoria), con un puntero al primer elemento y al último, y un atributo largo, que lleva la cuenta de la cantidad de elementos de la lista. Esta implementación se usó debido a que fue la que brindaba la mayor eficiencia para realizar las operaciones planteadas. Las más usadas, agregarPrincipio y agregarFinal, se realizan en orden 1. Esto hace que sus algoritmos sean eficientes, de tal manera que todas las veces que utilizemos sus funciones en las operaciones de Archivo, estas se realizarán eficientemente.

Por otro lado utilizamos el TAD “PilaCola”, que en realidad lo utilizamos fue el TAD Pila, agregando los métodos que nosotros queríamos. Ya que el TAD Pila y el TAD Cola, por separados, no era necesarios para utilizarlos en ninguna parte del obligatorio.

De tal manera que cuando se nos leyó la letra del Obligatorio y paramos en la función Undelete, lo que nosotros pensamos es que es el comportamiento de una Pila y una Cola a la vez.

Es decir que podíamos realizar un TAD propio, de tal manera que utilizaremos los métodos que nosotros quisiéramos y darle el comportamiento necesario para que funcionara como una pila y una cola al mismo tiempo. Que es muy parecido a lo que se solicita para la

función Undelete. Para la implementación, se usó el TAD ya implementado, ListaPos, con un puntero al primero de la lista, una variable de instancia largo y otra lleno, que indican la cantidad de elementos en la Pila, y la cantidad máxima que la misma admite. Esta implementación permite que las operaciones más importantes del TAD, Pop y Push, sean en orden 1 en tiempo de ejecución. Esto permite que las operaciones que usan este TAD sean eficientes.

A continuación hablaremos de los dos algoritmos que nos parecieron más interesantes:

- 1) En este caso hablaremos de Copiar Directorio, adjuntamos una imagen de lo qué es el código del mismo

```
void Directorio::CopiarDirectorio(Cadena rutaOrigen, Cadena rutaDestino)
{
    bool esSubruta = this->verificarCadena(rutaOrigen, rutaDestino);

    if (!esSubruta)
    {
        Directorio dCopia = Directorio();

        dCopia.nombre = this->buscarRuta(rutaOrigen).nombre;
        dCopia.listaArchivos = this->buscarRuta(rutaOrigen).listaArchivos->Clon();
        dCopia.listaDirectorio = this->buscarRuta(rutaOrigen).listaDirectorio->Clon();

        this->crearDirectorio(rutaDestino);
        while (!dCopia.listaDirectorio->EsVacia())
        {
            this->buscarRuta(rutaDestino).listaDirectorio->AgregarOrd(dCopia.listaDirectorio->Minimo());
            dCopia.listaDirectorio->BorrarMinimo();
        }
        while (!dCopia.listaArchivos->EsVacia())
        {
            this->buscarRuta(rutaDestino).listaArchivos->AgregarOrd(dCopia.listaArchivos->Minimo());
            dCopia.listaArchivos->BorrarMinimo();
        }
    }
}
```

Como convenciones, establecemos que:

- Al copiar un directorio se copiarán todos archivos, los subdirectorios y los archivos que contengan.
- Los directorios y archivos copiados deben ser completamente independientes (no comparten memoria).
- El directorio de destino no puede ser un subdirectorio del origen.
- Todos los atributos de los directorios y los archivos se deben mantener.
- El directorio destino no debe existir.

En este caso lo que nosotros hacemos en el código es verificar primero que la ruta Destino no sea subruta de la ruta Origen, porque sino estaríamos moviéndonos dentro de la ruta Origen, algo que no tiene sentido. Está verificación la hacemos mediante un método externo que verifica si son subruta. Entonces se va a poder copiar un directorio si ocurre que la ruta Destino no es subruta de la ruta Origen.

Luego lo que se hace es crearse el directorio copia que es el que queremos mover a la ruta destino. De tal manera que copiamos el nombre del mismo, clonamos la lista de Archivos y también clonamos la lista de directorio. Ya que por letra se especificaba que debía contener

todo igual. En este caso lo que hacemos luego de haber utilizado la función implementada en las listas para clonar, es crear el directorio en la ruta Destino.

Esto se hace debido a que la ruta Destino no debe existir de tal manera que luego lo que se hace es asignar los archivos y los directorios correspondientes al directorio en la ruta Destino. De tal manera que queda la copia con todos los elementos que tenía el directorio original.

La segunda función que consideramos interesante es la función que está en la clase Archivo la cual es insertar texto. La consideramos interesante debido a que en un primer momento en los atributos de Archivo, teníamos una lista Pos de tipo char*. De tal manera que nosotros implementamos esta función con la forma de char*. Luego cuando quisimos realizar las pruebas sucedió que estaba bien implementada, pero daba problemas de memoria, de tal manera que decidimos cambiar el atributo y pasarse a Cadena. De tal manera que se nos simplificó el razonamiento del código.

A continuación se adjunta una imagen del Insertar texto:

```
void Archivo::InsertarTexto(unsigned int nroLinea, unsigned int posLinea, Cadena texto)
{
    int contador = 0;
    contador = this->matrizTexto->CantidadElementos();
    while (contador < nroLinea) {
        agregarLineasVacias();
        contador++;
    }
    Cadena linea = this->matrizTexto->ElementoPos(nroLinea-1);
    Cadena izq = "";
    Cadena der = "";

    for (int i = 0; i < linea.Length(); i++)
    {
        char c = linea[i];
        char * pc = new char[2];
        pc[0] = c;
        pc[1] = '\\0';
        Cadena caden = Cadena(pc);
        if (i < posLinea) {
            izq = Cadena(izq + caden);
        }
        else {
            der = Cadena(der + caden);
        }
    }
    this->matrizTexto->ElementoPos(nroLinea-1) = izq + texto + der;
}
```

Agrega un texto en la línea y posición del archivo. Si el número de línea no existe, se deben insertar líneas vacías hasta llegar a la nueva línea. Si existen caracteres en la posición marcada entonces el nuevo texto se insertará corriendo los caracteres en esa posición hacia adelante.

En caso de que no haya caracteres que permitan llegar a la posición se deberán insertar espacios en blanco. El número de línea y la posición comienzan en 1. No hay límite en la cantidad de caracteres por línea ni en la cantidad de líneas. Todos los archivos se pueden escribir estén ocultos o no.

3. Conclusiones

Para concluir con él obligatorio, en nuestro caso está funcionando todo correctamente. Están todas las funciones implementadas, de tal manera que sí se corre el archivo de corrección del obligatorio debería funcionar correctamente. Por otra opinamos que él obligatorio se nos debería haber dado con por lo menos una semana de anterioridad, ya que casi siempre a la hora de realizarse surgen complicaciones que uno jamás se imagina. De tal manera que por lo menos una semana más, beneficia al estudiante y no perjudica al profesor. Debido a que la fecha de entrega sería la misma. Lo que cambiaría sería únicamente que en las clases de práctico se dedicará una semana más al obligatorio.

4. Memoria de las tareas desarrolladas por cada integrante del grupo.

Operación	Fue Imple- men- tada	Estado de función	Desarrollad o por		Test de las Operacione s		Comentarios
			Joaq uin Lam ela	Mart in Gut man	Joaq uin Lam ela	Mart in Gut man	
TIPO 1							
Destructor	SI	OK	X	X	X	X	Funciona correctamente
Constructor	SI	OK	X	X	X	X	Funciona correctamente
Mkdir	SI	OK	X	X	X	X	Funciona correctamente
Dir	SI	OK		X	X	X	Funciona correctamente
CreateFile	SI	OK	X	X	X	X	Funciona correctamente
InsertText	SI	OK	X		X	X	Funciona correctamente
Type	SI	OK	X	X	X	X	Funciona correctamente
TIPO 2							
Rmdir	Si	OK	X	X	X	X	Funciona correctamente
Copydir	Si	OK		X	X	X	Funciona correctamente
Delete	SI	OK	X	X	X	X	Funciona correctamente
Attrib	SI	OK	X	X	X	X	Funciona correctamente
DeleteText	SI	OK	X		X	X	Funciona correctamente
Undelete	SI	OK		X	X	X	Funciona correctamente

OTROS							
Pila	SI	OK		X	X	X	Funciona correctamente
ListaPos	Si	OK	X	X	X	X	Funciona correctamente