Universidad Nacional del Centro de la
Provincia de Buenos Aires

FACULTAD DE CIENCIAS EXACTAS

Ingeniería de Sistemas

Trabajo Práctico Especial

Análisis y Diseño de Algoritmos I

Ayudante a cargo: Matías Antúnez

GRUPO 15

Bedini Crocci Pia pbedini@alumnos.exa.unicen.edu.ar

Burckhardt David burck432@gmail.com

02/06/2021

Índice

Introducción 2

Desarrollo 3

ESPECIFICACIÓN FORMAL EN NEREUS 3

TDA Empresa 3

TDA Lista 3

TDA Ranking_Empresas 4

ESTRUCTURAS DE DATOS 5

TDA Empresa 5

TDA Lista 5

TDA Ranking_Empresas 5

DESCRIPCIÓN Y COMPLEJIDAD TEMPORAL DE FUNCIONES 6

TDA Empresa 6

TDA Lista 7

TDA Ranking_Empresas 7

COMBINACIÓN DE TDA'S 8

DESCRIPCIÓN DETALLADA DE SERVICIOS 9

SERVICIO 1 9

SERVICIO 2 10

SERVICIO 3 10

CÓDIGO FUENTE 11

Main.cpp 11

servicios.h 12

servicios.cpp 12

ListaSimple.h 20

ListaSimple.cpp 21

Empresa.h 23

Empresa.cpp 23

Conclusión 26

Introducción

En el presente informe se abordarán los temas vistos hasta la fecha en la cátedra Análisis y Diseño de Algoritmos I. Los mismos son: análisis de eficiencia, complejidad temporal, especificación e implementación de TDAs (tipos de datos abstractos).

El objetivo de este trabajo es trasladar los conocimientos teóricos a la práctica, aplicándolo en el lenguaje propuesto por la cátedra, C++, utilizando el IDE Code Blocks. A su vez, este documento contribuirá en la fijación de los conceptos importantes tales como: la abstracción de datos, la separación de responsabilidades y el ocultamiento de información.

A continuación, se argumentarán las decisiones tomadas en relación al código programado y se detallará el impacto que poseen las estructuras de datos utilizadas, teniendo en cuenta su complejidad temporal y su consumo de memoria. Además, se las analizará respecto de la especificación algebraica (realizada en el lenguaje NEREUS), haciendo hincapié en la eficiencia de cada función.

Desarrollo

ESPECIFICACIÓN FORMAL EN NEREUS

TDA Empresa

CLASS Empresa
IMPORTS Natural, Cadena
BASIC CONSTRUCTORS crearEmpresa
EFFECTIVE
TYPE Empresa
OPERATIONS

CB crearEmpresa: Cadena * Natural * Cadena * Natural * Cadena → Empresa;

O obtenerRazonSocial: Empresa → Cadena;

O obtenerPosicionRanking: Empresa → Natural;

O obtenerPaisDeOrigen: Empresa → Cadena;

O obtenerCantEmpleados: Empresa → Natural;

O obtenerRubro: Empresa → Cadena;

AXIOMS
END_CLASS

TDA Lista

CLASS Lista [Elemento] //DESORDENADA
IMPORTS Natural
BASIC CONSTRUCTORS iniciaLista, agregar Lista
EFFECTIVE
TYPE Lista [Elemento]
OPERATIONS

CB inicLista: → Lista;

CB agregarLista: Lista * Elemento → Lista;

O longLista: Lista → Natural;

O recuperarLista: Lista(l) * Nat(i) → Elemento

pre: (i >= 1) and (i <= longLista(l));

AXIOMS

END_CLASS

TDA Ranking_Empresas

CLASS Ranking_Empresas

IMPORTS Natural, Lista [Empresa], Cadena

BASIC CONSTRUCTORS crearEmpresa

EFFECTIVE

TYPE Ranking_Empresas

OPERATIONS

CB inicRanking: → Ranking_Empresas;

CB agregar: Ranking_Empresas(r) * Empresa(e) → Ranking_Empresas

pre: not pertenece(r,e);

O pertenece: Ranking_Empresas * Cadena → Boolean;

O esVacio: Ranking_Empresas → Boolean;

O cantEmpresas: Ranking_Empresas → Natural;

O obtenerPosRanking: Ranking_Empresas(r) * Cadena(e) → Natural

pre: not esVacio(r) and pertenece(r,e) ;

O obtenerInfoEmpresa: Ranking_Empresas(r) * Natural(p) → Empresa

pre: not esVacio(r) and cantEmpresas >= p and p > 0;

O listarEmpresas: Ranking_Empresas * Natural(min) * Natural(max) → Lista

[Empresa]

pre: min > 0 and max >= min;

AXIOMS

END_CLASS

ESTRUCTURAS DE DATOS

TDA Empresa

Para el TDA Empresa no se utilizó una estructura de implementación, sino una clase Empresa. Esta clase nos permite crear un objeto de este tipo a partir de una razón social, rubro, país, posición y cantidad de empleados. Dicha clase facilitará almacenar las empresas en un arreglo dinámico para, de esta forma, no realizar copias de información ni abuso de memoria. Además de crear y destruir un objeto Empresa, es factible consultar y visualizar cada dato que conforma el objeto por separado a partir del llamado a las funciones de “obtenerRazonSocial”, “obtenerPosRanking”, “obtenerPais”, “obtenerCantEmpleados”, “obtenerRubro”. El constructor que otorga por defecto la creación de una nueva clase, no fue utilizado, sino que se formuló una operación “Empresa” que recibe los datos y los carga en un nuevo objeto.

TDA Lista

Para el TDA Lista se empleó una clase ListaSimple que trabaja con nodos de tipo Empresa y acepta las operaciones de agregar, preguntar si está vacía (además de crear y destruir), iniciar un cursor, avanzarlo, obtener la empresa que apunta, y preguntar si se encuentra en el final de la lista. Para lograr mostrar la lista cargada por pantalla, se realizó un procedimiento que se encuentra en el archivo servicios.cpp y utiliza estos últimos cuatro métodos de la clase sin realizar ninguna muestra de información de las empresas (con “couts”) dentro de ésta. La clase ListaSimple fue parametrizada de tal manera que sus nodos posean un campo para guardar un elemento de tipo T (elemento genérico), y un puntero al siguiente. Estos datos se guardan en un registro, es decir, un struct llamado NodoLista.

TDA Ranking_Empresas

Para el TDA Ranking_Empresas se llevó a cabo un menú que permitió resolver los tres servicios a partir de una opción ingresada por el usuario.

La información de las empresas fue guardada en un objeto Empresa y apuntado por punteros que conforman el arreglo dinámico, estableciendo así, la estructura de almacenamiento.

Para el servicio de obtener la posición en el ranking, se combinó un Árbol Binario de Búsqueda de punteros basado en las empresas cargadas en el arreglo.

Si el usuario deseaba obtener la información de la empresa, bastó con utilizar la misma estructura de almacenamiento para “pararnos”, según el índice (que hace referencia la posición en el ranking) en el lugar indicado y extraer los datos de esa empresa.

Por último, para el servicio que pide listar las empresas según un rango introducido por el usuario, se utilizó una lista simple a la que se le iban adjudicando empresas que cumplían las restricciones dadas por el valor mínimo y máximo del rango. Estas empresas se clasifican mediante el recorrido completo del arreglo dinámico.

DESCRIPCIÓN Y COMPLEJIDAD TEMPORAL DE FUNCIONES

TDA Empresa

- crearEmpresa: Es la constructora básica que nos permite crear un objeto de tipo empresa a partir de cinco variables, tres de tipo cadena (Razón Social, País y Rubro) y dos naturales (Posición en el ranking y Cantidad de empleados). Complejidad temporal: O(1) debido a que no requiere recorrer ninguna estructura, es decir, no depende de un ciclo, es constante.
- obtenerRazonSocial: Esta observadora nos devuelve la razón social de una empresa dada. Complejidad temporal: O(1) ya que basta con retornar el nombre de la empresa. Operación elemental.
- obtenerPosicionRanking: Esta observadora retorna la posición en el ranking de una empresa dada. Complejidad temporal: O(1) debido a que basta con retornar la posición de la empresa en el ranking y no es necesario recorrer ninguna estructura.
- obtenerPaisDeOrigen: Esta observadora retorna el país de origen de una empresa dada. Complejidad temporal: O(1) ya que basta con devolver el país al que pertenece la empresa. Operación constante.
- obtenerCantEmpleados: Esta observadora retorna la cantidad de empleados de una empresa dada. Complejidad temporal: O(1) debido a que la cantidad de empleados es un valor propio de cada empresa y su acceso no solicita el recorrido de una estructura.

- obtenerRubro: Esta observadora retorna el rubro de una empresa dada. Complejidad temporal: O(1) ya que la operación que permite conocer el rubro de cualquier empresa es constante.

TDA Lista

- inicLista: Es la constructora básica que nos permite inicializar una lista vacía. Complejidad temporal: O(1) debido a que simplemente requiere iniciar un nodo apuntando a NULL, es una operación constante.
- agregarLista: Esta constructora básica nos permite agregar un elemento a la lista. Complejidad temporal: O(1) debido a que la lista está desordenada y los elementos se agregan al final de ésta, por lo tanto no es necesario recorrerla.
- longLista: Esta observadora retorna la longitud de la lista a partir de contar los elementos de la misma. Complejidad temporal: O(n) siendo "n" la cantidad de elementos de la lista, ya que para saber la cantidad de elementos de debe contar uno por uno transitando la lista completa.
- recuperarLista: Esta observadora permite recuperar cualquier elemento de la lista a partir de la ubicación a la cual se quiere acceder mediante un índice (natural). Complejidad temporal: O(1) debido a que el índice posibilita pararse en la posición exacta de la estructura a la cual se quiere acceder sin recorrerla. En el código, no existe un “recuperarLista” como tal, sino un procedimiento que utiliza métodos de la clase para mostrar la lista completa. Debido a que se necesita avanzar por todos los nodos para visualizar su información, la complejidad temporal de la implementación es O(n) siendo “n” la cantidad de elementos de la lista.

TDA Ranking_Empresas

- inicRanking: Es la constructora básica que nos permite inicializar un ranking vacío. Complejidad temporal: O(1) ya que es una operación constante.
- agregar: Esta constructora básica nos permite agregar una empresa al ranking. Complejidad temporal: O(1) debido a que no es necesario recorrer el ranking para agregar una nueva empresa al final.
- pertenece: Esta observadora retorna true o false dependiendo de si la empresa está o no en el ranking. Complejidad temporal: O(log n) siendo "n" la cantidad

de empresas en el ranking, ya que para saber si una empresa pertenece o no, no sería necesario transitar el ranking en su totalidad, sino que se podría realizar una búsqueda binaria sobre el arreglo dinámico (estructura de almacenamiento seleccionada) y así reducir las posibilidades a la mitad.

- esVacio: Esta observadora nos permite saber si el ranking está vacío o no, retornando true si lo está o false si tiene elementos. Complejidad temporal: O(1) debido a que basta con saber si el ranking tiene al menos una empresa, no requiere recorridos.
- cantEmpresas: Esta observadora retorna la cantidad de empresas en el ranking. Complejidad temporal: O(n) siendo “n” el tamaño del arreglo, puesto que al igual que la operación longLista del TDA lista, esta operación precisa contar una por una las empresas en el ranking.
- obtenerPosRanking: Esta observadora retorna la posición en el ranking de una empresa dada. Complejidad temporal: O(n) considerando a "n" como la cantidad de nodos del árbol, ya que en el peor de los casos, cuando el árbol esté desbalanceado, sería necesario un recorrido lineal.
- obtenerInfoEmpresa: Esta observadora retorna una empresa que contiene la información solicitada (Razón social, País, Cantidad de empleados y Rubro) dado un ranking y una posición válida. Complejidad temporal: O(1) ya que la operación se puede realizar mediante el acceso a una celda determinada del arreglo.
- listarEmpresas: Esta observadora lista las empresas (Razón Social y Posición en el ranking) del ranking que estén dentro de un rango (mín y máx) de cantidad de empleados. Complejidad temporal: O(n) siendo “n” el tamaño del arreglo, porque es necesario clasificar cada empresa del ranking y para eso se debe avanzar por medio del arreglo completo analizando celda por celda.

COMBINACIÓN DE TDA’S

El almacenamiento de las empresas se maneja mediante un arreglo dinámico que se carga al leer el archivo una única vez. Al realizarse una sola vez durante la jornada laboral, no es necesario que sea tan eficiente a comparación de los servicios que de seguro se realizarán múltiples veces al día. El árbol ordenado por posición en el ranking (ordenado descendentemente) no es viable ya que el archivo está ordenado

por ese mismo criterio, entonces esta estructura dejaría a cada nodo con un hijo mayor (menos al último) tomando la forma de una lista simple, es decir, no simplificaría la búsqueda.

El procedimiento se basa en crear un objeto Empresa e insertarlo en las celdas del arreglo ordenado por posición, aprovechando que el archivo ya está ordenado con el mismo criterio. Así, la empresa 1 se encuentra en la posición 0 del arreglo, y la última, en la posición n-1, donde “n” es la cantidad de empresas en el archivo. No hay copia de información debido a que se trabaja con punteros a empresa.

DESCRIPCIÓN DETALLADA DE SERVICIOS

Seguidamente, se procederá a explicar cómo fueron utilizados los TDA’s para implementar los servicios, se presentarán las complejidades temporales totales y se analizará la dependencia de las estructuras de datos elegidas en la complejidad final de cada servicio. Asimismo, se intentará responder preguntas como: ¿La estructura de datos beneficia/perjudica el costo de la resolución? ¿Se podría mejorar dicho costo? En caso afirmativo, ¿Cómo se podría mejorar el costo del servicio? ¿Qué costo o desventajas generaría esa mejora?.

SERVICIO 1

Para el primer servicio, teniendo en cuenta que el cliente lo utilizará repetidamente durante la jornada laboral, se tomó la decisión de usar un árbol binario de punteros empresa ordenado por razón social como estructura de acceso. Además, al utilizar punteros evitamos la copia de información de todas las empresas del arreglo. Este árbol fue implementado mediante una clase llamada “ABB” (Árbol Binario de Búsqueda) que utiliza un struct de nodos que contienen entre sus campos, punteros a izquierda, derecha y a una empresa del arreglo.

En el mejor caso, si el árbol estuviera balanceado, es posible reducir la complejidad temporal de la búsqueda de la posición a O(log n) siendo "n" la cantidad de nodos del árbol, ya que el recorrido en un árbol binario se va reduciendo a la mitad a medida que avanzamos de nivel.

En el peor caso, si el árbol estuviese desbalanceado, el recorrido sería lineal y la complejidad temporal sería de O(n), porque al recorrerlo no se descartarían tantas posibilidades como lo haría si estuviera balanceado.

En síntesis, la utilización de un árbol puede mejorar la eficiencia para algunos casos, ya que si las empresas estuvieran ubicadas en el arreglo en una forma tal que al construir el árbol éste quedara desbalanceado, entonces no serviría y en el peor caso quedaría como una lista. En consecuencia, el buscar una empresa tendría una complejidad O(n). Aun así, si las empresas se pudieran ordenar por razón social en un arreglo, se podría mejorar la eficiencia realizando directamente una búsqueda binaria sobre el arreglo para así alcanzar una complejidad O(log n).

La estructura de datos beneficia el costo de resolución del servicio ya que la complejidad de búsqueda esperada para todos los casos sería O(n), es decir, recorrer el arreglo o el archivo ordenado por posición y, mediante la razón social de la empresa, determinar su posición en el ranking. Con un árbol binario de búsqueda balanceado se logra reducir la complejidad a O(log n) en el mejor de los casos, y se mantiene la complejidad O(n) en el peor de ellos.

SERVICIO 2

Para el segundo servicio se aprovechó la estructura de almacenamiento, es decir, el arreglo dinámico de punteros empresa, debido a que al requerir la información de una de ellas no fue necesario el uso de una estructura auxiliar de acceso.

La complejidad temporal del servicio es O(1) ya que al conocer la posición de la empresa de la que queremos saber su información, es posible pararse en la celda del arreglo correcto sin tener que recorrerlo completo. Por ejemplo: si la empresa de la que se quiere conocer su información está en la posición n, entonces basta con detenernos en la posición n-1 del arreglo.

La información de esta empresa la guardamos en un nuevo objeto empresa para no realizar ninguna muestra de información dentro del procedimiento que resuelve el servicio. Por esa razón se genera una copia de información que es aceptable en la resolución. No es posible mejorar la eficiencia de este servicio ya que O(1) es la complejidad mínima de un algoritmo.

SERVICIO 3

Para el tercer servicio se decidió tratar con una estructura de acceso auxiliar: una lista simple. Para ello declaramos la clase lista que hace posible coleccionar las empresas que cumplen con el rango de la cantidad de empleados.

En este caso el recorrido es completo, se realiza sobre el arreglo dinámico, y puesto que hay que analizar empresa por empresa para saber si su cantidad de empleados cumple o no con el rango, la complejidad temporal de la búsqueda de ese servicio es O(n).

No es posible mejorar esta eficiencia ya que al intentar tratar con un árbol binario de búsqueda ordenado por cantidad de empleados, se llegó a la conclusión de que la complejidad del servicio sería igual que la mencionada anteriormente. Esto es debido a que en el peor caso, al ir avanzando, se va descartando únicamente el nodo raíz, por lo que la búsqueda continúa por ambos subárboles, con lo cual estaríamos cayendo nuevamente en una complejidad O(n).

No fue implementado en el código dado que conlleva una desventaja de complejidad y de legibilidad muy grande y no aporta ningún beneficio en cuanto a la eficiencia. Se consideró más conveniente el recorrido sobre el mismo arreglo.

CÓDIGO FUENTE

Cambios realizados respecto de la primer corrección:

- Se parametrizó la clase ListaSimple;
- Se modificó la forma de mostrar la lista en el servicio 3.

Main.cpp

```
#include <iostream>
#include "Empresa.h"
#include "Servicios.h"
#include "ABB.h"

using namespace std;

int main()
{
   unsigned int tamanioArreglo;
   ABB * Arbol = new ABB();
   Empresa * Arreglo =
Procesar_Archivo_Entrada("Ranking.csv",tamanioArreglo,Arbol);
   Resolver(Arreglo,Arbol,tamanioArreglo);
   delete [] Arreglo; //Eliminamos el arreglo para que no quede espacio ocupado en
la memoria
   delete Arbol;
   return 0;
```

```
}
```

servicios.h

```
#ifndef SERVICIOS_H_INCLUDED
#define SERVICIOS_H_INCLUDED
#include "Empresa.h"
#include "ABB.h"
#include "ListaSimple.h"

void Menu(unsigned int & Opcion);
void ObtenerPosicion(ABB * Arbol,string & razonSocial,unsigned int & Posicion);
void ObtenerInfo(Empresa * Arreglo,unsigned int & posicion,unsigned int
tamanioArreglo,Empresa & Informacion);
void ListarEmpresas(Empresa * Arreglo, unsigned int tamanioArreglo);
void MostrarLista(ListaSimple<Empresa *> ListaEmpleados);
void Resolver(Empresa * Arreglo,ABB * arbol, unsigned int tamanioArreglo);
Empresa * Procesar_Archivo_Entrada(string origen, unsigned int &
tamanioArreglo,ABB * & arbol); // Procesa el archivo y carga las estructuras (arreglo
y árbol)

#endif // SERVICIOS_H_INCLUDED
```

servicios.cpp

```
#include <fstream>
#include "ABB.h"
#include "Servicios.h"
#include "ListaSimple.h"
#include "Empresa.h"

//Este procedimiento nos muestra el menu y nos pide que elijamos el servicio a
realizar
void Menu(unsigned int & Opcion)
{
    cout << endl;
    cout << "BIENVENIDO A CONSULTORIA" << endl;
    cout << "1- Obtener posicion en el ranking de una empresa dada" << endl;
    cout << "2- Obtener informacion de una Empresa dada una posicion" << endl;
    cout << "3- Listar las empresas junto a su posicion en el ranking, que cuentan con
un nro de empleados dentro de un rango dado" << endl;
    cout << "4- Salir" << endl;
    cout << endl;
    cout << "Elija el servicio a realizar: ";
```

```
    cin >> Opcion;
    while ((Opcion != 1) && (Opcion != 2) && (Opcion != 3) && (Opcion != 4)){
        cout << endl;
        cout << "OPCION INVALIDA" << endl;
        cout << endl;
        cout << "Elija el servicio a realizar: ";
        cin >> Opcion;}
    cout << endl;
}

//Este procedimiento nos permite obtener la posicion en el ranking de una empresa determinada a partir de saber su nombre
void ObtenerPosicion(ABB * Arbol,string & razonSocial,unsigned int & PosicionEmpresa)
{
    cout << "Ingrese la razon social de la empresa a conocer su posicion: ";
    cin.ignore();
    getline(cin, razonSocial);
    cout << endl;
    PosicionEmpresa = Arbol->obtenerPosicion(razonSocial);
}
//Este procedimiento nos devuelve un objeto empresa con la información de la empresa que se encuentra en una posición dada
void ObtenerInfo(Empresa * Arreglo,unsigned int & Posicion,unsigned int tamanioArreglo,Empresa & Informacion)
{
    cout << "Ingrese la posicion de la empresa a conocer su informacion: ";
    cin >> Posicion;
    cout << endl;
    if ((Posicion > 0) && (Posicion <= tamanioArreglo))
        Informacion = Arreglo[Posicion-1];
    else
        Posicion = 0;
}

//Este procedimiento nos permite listar las empresas que cuentan con una cantidad de empleados determinada.
void ListarEmpresas(Empresa * Arreglo,ListaSimple<Empresa *> & ListaEmpleados, unsigned int tamanioArreglo)
{
    unsigned int RangoMax;
    unsigned int RangoMin;
    cout << "Ingrese el rango de la cantidad de empleados que desee" << endl;
    cout << "Minimo: ";
```

```
    cin >> RangoMin;
    cout << "Maximo: ";
    cin >> RangoMax;
    cout << endl;
    if (RangoMin <= RangoMax){
        cout << "RANGO: [" << RangoMin << "," << RangoMax << "]" << endl;
        for(unsigned int i=0; i<tamanioArreglo; i++){
            if ((Arreglo[i].obtenerCantEmpleados() >= RangoMin) &&
(Arreglo[i].obtenerCantEmpleados() <= RangoMax)){
                Empresa * puntEmpresa = &Arreglo[i];
                ListaEmpleados.Agregar(puntEmpresa);}}
    }else
        cout << "RANGO INVALIDO" << endl;
}

void MostrarLista(ListaSimple<Empresa *> ListaEmpleados) //O(n)
{
    ListaEmpleados.IniciarCursor(); //O(1)
    while (!(ListaEmpleados.CursorEsFinal())){

        Empresa * Info = (ListaEmpleados.ObtenerDesdeCursor()); //O(1)
        cout << Info->obtenerPosRanking() << " | " << Info->obtenerRazonSocial() << "
| " << Info->obtenerCantEmpleados() << endl;
        ListaEmpleados.AvanzarCursor(); //O(1)
    }
}

//Este procedimiento se encarga de mostrar el menú al usuario y ejecutar la opción
que este desee
void Resolver(Empresa * Arreglo, ABB * Arbol, unsigned int tamanioArreglo)
{
    unsigned int Opcion;
    Menu(Opcion);
    switch(Opcion){
            case 1: {string razonSocial; //SERVICIO 1)
                    unsigned int PosicionEmpresa;
                    ObtenerPosicion(Arbol,razonSocial,PosicionEmpresa);
                    if (PosicionEmpresa != 0)
                        cout << "La empresa " << razonSocial << " se encuentra en la
posicion " << PosicionEmpresa << " del ranking." << endl;
                    else
                        cout << "La empresa no se encuentra en el listado" << endl;
                    Resolver(Arreglo,Arbol,tamanioArreglo);
                    break;}
```

```
        case 2: {unsigned int Posicion; //SERVICIO 2)
                Empresa Informacion;
                ObtenerInfo(Arreglo,Posicion,tamanioArreglo,Informacion);
                if (Posicion != 0){
                   cout << "| RAZON SOCIAL | PAIS | CANT EMPLEADOS | RUBRO |"
<< endl;
                   cout << "   " << Informacion.obtenerRazonSocial() << "  |  " <<
Informacion.obtenerPais() << "  |  " << Informacion.obtenerCantEmpleados() << "  |  "
<< Informacion.obtenerRubro() << endl;}
                else
                   cout << "La posicion ingresada no se encuentra en el ranking." <<
endl;
                Resolver(Arreglo,Arbol,tamanioArreglo);
                break;}

        case 3: {
                ListaSimple<Empresa *> ListaEmpleados; //SERVICIO 3)
                ListarEmpresas(Arreglo,ListaEmpleados,tamanioArreglo);
                cout << endl;
                if (ListaEmpleados.Vacia())
                   cout << "No existen empresas con el rango de cantidad de
empleados especificado" << endl;
                else{
                   cout << endl;
                   cout << "Las empresas que tienen una cantidad de empleados dentro
del rango son: " << endl;
                   MostrarLista(ListaEmpleados);}
                Resolver(Arreglo,Arbol,tamanioArreglo);
                //delete ListaEmpleados;
                break;}

        case 4: {cout << "Hasta la proxima!" << endl;
                break;}
   }
}


/**
 * Abre el archivo según el origen, procesa las líneas del mismo y
 * almacena la información resultante en el contenedor pasado por referencia.
 **/
//Comentarios: atoi y atof requieren un char * para converter a número, usamos
c_str de la clase string.
```

```
Empresa * Procesar_Archivo_Entrada(string origen, unsigned int & tamanioArreglo,
ABB * & Arbol)
{
   ifstream archivo(origen);
   if (!archivo.is_open()){
      cout << "No se pudo abrir el archivo: " << origen << endl;
      return NULL;}
   else {
      string linea;
      getline(archivo, linea);
      unsigned int cantEmpresas = atoi(linea.c_str());
      tamanioArreglo = cantEmpresas;
      Empresa* Arreglo = new Empresa[cantEmpresas]; //arreglo en el heap, ya no
se destruye pq no está en la pila, trabajo con un bloque de memoria continuo en el
heap
      cout << "Se cargaron " << cantEmpresas << " empresas." << endl;

      unsigned int posicion = 0;

      //Leemos de una linea completa por vez (getline).
      while (getline(archivo, linea)) {
         //Primer posición del separador ;
         int pos_inicial = 0;
         int pos_final = linea.find(';');

         //Informacion entre pos_inicial y pos_final
         string razonSocial = linea.substr(pos_inicial, pos_final);

         //Segunda posición del separador ;
         pos_inicial = pos_final + 1;
         pos_final = linea.find(';', pos_inicial);
         string pais = linea.substr(pos_inicial, pos_final - pos_inicial);

         //Tercera posición del separador ;
         pos_inicial = pos_final + 1;
         pos_final = linea.find(';', pos_inicial);
         int cantEmpleados = atoi(linea.substr(pos_inicial, pos_final -
pos_inicial).c_str());

         //Cuarta posición del separador ;
         pos_inicial = pos_final + 1;
         pos_final = linea.find(';', pos_inicial);
         string rubro = linea.substr(pos_inicial, pos_final - pos_inicial);
```

```
            unsigned int posRanking = posicion + 1;

            Empresa e(razonSocial, pais, cantEmpleados, rubro, posRanking);
            Arreglo[posicion] = e; //Se carga la estructura de almacenamiento a medida
que se recorre el archivo.
            Empresa * puntEmpresa = &Arreglo[posicion];
            Arbol->InsertarNodo(puntEmpresa); //Se carga la estructura de acceso que
se utiliza para el 1er servicio.
            posicion++;
        }
        return Arreglo;
    }
}
```

ABB.h

```
#ifndef ABB_H
#define ABB_H
#include "Empresa.h"

class ABB{

public:

        ABB();
        ~ABB();
        void InsertarNodo(Empresa * puntEmpresa);
        unsigned int obtenerPosicion(string razonSocial);
        void MostrarArbol();

private:

    struct NodoArbol{
        Empresa * puntArreglo;
        NodoArbol *izq;
        NodoArbol *der;
    };
    void crearNodo(NodoArbol * & Nuevo, Empresa * puntEmpresa);
        void InsertarNodo(NodoArbol * & Raiz, Empresa * puntEmpresa);
        unsigned int obtenerPosicion(NodoArbol * & Raiz, string razonSocial);
        void MostrarArbol(NodoArbol * Raiz);
        void EliminarArbol(NodoArbol * Raiz);
        NodoArbol * Raiz;
        NodoArbol * Nuevo;
```

```
};

#endif // ABB_H
```

ABB.cpp

```
#include "ABB.h"
#include <iostream>

using namespace std;

ABB::ABB()
{
        Raiz = NULL;
}

void ABB::EliminarArbol(NodoArbol * Raiz)
{
   if(Raiz != NULL){
                EliminarArbol(Raiz->izq);
                EliminarArbol(Raiz->der);
                delete Raiz;}
}

ABB::~ABB()
{
   EliminarArbol(Raiz);
}

void ABB::crearNodo(NodoArbol * & Nuevo, Empresa * puntEmpresa)
{
   Nuevo = new NodoArbol;
   Nuevo->puntArreglo = puntEmpresa;
   Nuevo->izq = NULL;
   Nuevo->der = NULL;
}

void ABB::InsertarNodo(NodoArbol * & Raiz,Empresa * puntEmpresa)
{
   if (Raiz == NULL)
   {
      crearNodo(Nuevo,puntEmpresa);
      Raiz = Nuevo;}
```

```
   else if (Raiz->puntArreglo->obtenerRazonSocial() > puntEmpresa-
>obtenerRazonSocial())
      InsertarNodo(Raiz->izq,puntEmpresa);
   else
      InsertarNodo(Raiz->der,puntEmpresa);
}

void ABB::InsertarNodo(Empresa * puntEmpresa)
{
        if(Raiz == NULL){
      crearNodo(Nuevo,puntEmpresa);
      Raiz = Nuevo;}
   else{
      InsertarNodo(Raiz,puntEmpresa);
        }
}

unsigned int ABB:: obtenerPosicion(NodoArbol * & Raiz, string razonSocial)
{
   if (Raiz == NULL)
      return 0;
   else if (Raiz->puntArreglo->obtenerRazonSocial() < razonSocial)
      return obtenerPosicion(Raiz->der,razonSocial);
   else if (Raiz->puntArreglo->obtenerRazonSocial() > razonSocial)
      return obtenerPosicion(Raiz->izq,razonSocial);
   else if (Raiz->puntArreglo->obtenerRazonSocial() == razonSocial)
      return Raiz->puntArreglo->obtenerPosRanking();
}

unsigned int ABB:: obtenerPosicion(string razonSocial)
{
   if (Raiz == NULL)
      return 0;
   else
      return obtenerPosicion(Raiz,razonSocial);
}

void ABB::MostrarArbol(NodoArbol *Raiz) //MostrarArbol no se utilizó para resolver
los servicios, simplemente nos permitió verificar que la carga se realizaba
exitosamente
{
        if(Raiz != NULL){
                MostrarArbol(Raiz->izq);
                cout << Raiz->puntArreglo->obtenerRazonSocial() << " | ";
```

```
                MostrarArbol(Raiz->der);
        }
}

void ABB::MostrarArbol()
{
        MostrarArbol(Raiz);
}
```

ListaSimple.h

```
#ifndef LISTASIMPLE_H
#define LISTASIMPLE_H
#include "Empresa.h"

template <typename T>
class ListaSimple{

public:

   ListaSimple();
   ~ListaSimple();
   void Agregar(T & puntEmpresa);
   bool Vacia() const;
   void IniciarCursor();
   void AvanzarCursor();
   bool CursorEsFinal() const;
   Empresa * ObtenerDesdeCursor();

private:

   struct NodoLista{
        T puntArreglo;
        NodoLista * sig;
   };
   void crearNodoLista(NodoLista * & Nuevo,T & puntEmpresa);
   void Agregar(NodoLista * & Primero,T & puntEmpresa);
   bool Vacia(NodoLista * Primero) const;
   NodoLista * Primero;
   NodoLista * Nuevo;
   NodoLista * Cursor;
   NodoLista * Borrar;
};
```

```
#endif // LISTASIMPLE_H
```

ListaSimple.cpp

```
#include "ListaSimple.h"
#include <iostream>
#include <cassert>

using namespace std;

template <typename T>
ListaSimple<T>::ListaSimple()
{
   Primero = NULL;
}

template <typename T>
ListaSimple<T>::~ListaSimple()
{
   while(Primero != NULL){
      Borrar = Primero->sig;
      delete Primero;
                  Primero = Borrar;}
   Primero = NULL;
}

template <typename T>
void ListaSimple<T>::crearNodoLista(NodoLista * & Nuevo,T & puntEmpresa)
{
   Nuevo = new NodoLista;
   Nuevo->puntArreglo = puntEmpresa;
   Nuevo->sig = NULL;
}

template <typename T>
void ListaSimple<T>::Agregar(NodoLista * & NodoSig,T & puntEmpresa)
{
   if(NodoSig == NULL){
      crearNodoLista(Nuevo,puntEmpresa);
      NodoSig = Nuevo;}
   else
      Agregar(NodoSig->sig,puntEmpresa);
}
```

```
template <typename T>
void ListaSimple<T>::Agregar(T & puntEmpresa)
{
   if(Primero == NULL){
      crearNodoLista(Nuevo,puntEmpresa);
      Primero = Nuevo;}
   else
      Agregar(Primero->sig,puntEmpresa);
}

template <typename T>
bool ListaSimple<T>::Vacia(NodoLista * Primero) const
{
   if (Primero == NULL)
      return true;
   else
      return false;
}

template <typename T>
bool ListaSimple<T>::Vacia() const
{
   return Vacia(Primero);
}

template <typename T>
void ListaSimple<T>::IniciarCursor()
{
   Cursor = Primero;
}

template <typename T>
void ListaSimple<T>::AvanzarCursor()
{
   assert(!CursorEsFinal());
   Cursor = Cursor->sig;
}

template <typename T>
bool ListaSimple<T>::CursorEsFinal() const
{
   return (Cursor==NULL);
}
```

```
template <typename T>
Empresa * ListaSimple<T>::ObtenerDesdeCursor()
{
   assert(!CursorEsFinal());
   return Cursor->puntArreglo;
}

template class ListaSimple<Empresa *>;
```

Empresa.h

```
#ifndef EMPRESA_H
#define EMPRESA_H
#include <iostream>

using namespace std;

class Empresa{

public:

   Empresa();
   virtual ~Empresa();
   Empresa(string razonSocial, string pais, unsigned int cantEmpleados, string rubro,
unsigned int posRanking);
   string obtenerRazonSocial() const;
   string obtenerPais() const;
   unsigned int obtenerCantEmpleados() const;
   string obtenerRubro() const;
   unsigned int obtenerPosRanking() const;

private:

   string razonSocial;
   string pais;
   unsigned int cantEmpleados;
   string rubro;
   unsigned int posRanking;
};

#endif // EMPRESA_H
```

Empresa.cpp

```
#include "Empresa.h"
```

```
Empresa::Empresa()
{
    //ctor
}
Empresa::Empresa(string razonSocial, string pais, unsigned int cantEmpleados,
string rubro, unsigned int posRanking)
{
    this->razonSocial = razonSocial;
    this->pais = pais;
    this->cantEmpleados = cantEmpleados;
    this->rubro = rubro;
    this->posRanking = posRanking;
}

string Empresa::obtenerRazonSocial() const
{
    return razonSocial;
}

string Empresa::obtenerPais() const
{
    return pais;
}

unsigned int Empresa::obtenerCantEmpleados() const
{
    return cantEmpleados;
}

string Empresa::obtenerRubro() const
{
    return rubro;
}

unsigned int Empresa::obtenerPosRanking() const
{
    return posRanking;
}

Empresa::~Empresa()
{
    //dtor
}
```

Conclusión

Luego de haber finalizado la primer entrega de este trabajo y haber analizado y profundizado los temas propuestos por la cátedra, se puede concluir que la eficiencia no siempre implica que el código/el programa sea el mejor ya que éste puede verse complejizado en exceso innecesariamente, tanto en legibilidad como en el código mismo. Asimismo, fue posible apreciar el trabajo de un programador y la adaptación a los requerimientos de un cliente. Otra de las ventajas de la realización de este trabajo, fue el poder aprender e interiorizar un nuevo lenguaje de programación (C++). Esto permitió fijar conceptos teórico-prácticos, poniendo a prueba, a su vez, nuestra capacidad de programar un código con la mejor relación posible entre su complejidad y su eficiencia. Esto no significa que la estrategia utilizada haya sido la mejor que se podría haber empleado, pero fue la que más se adaptó a nuestros conocimientos.