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

30/06/2021

Índice

Introducción

El presente trabajo intenta tratar los temas vistos hasta la fecha en la cátedra Análisis y Diseño de Algoritmos I.

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.

Para contextualizar, es preciso aclarar que el servicio presente en esta segunda parte del trabajo requiere obtener la información de las empresas que están dentro de los primeros “n” puestos del ranking y que, además, cuentan con una cantidad de empleados dentro de un rango determinado.

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.

Desarrollo

SERVICIO 4 CON ARREGLO DINÁMICO Y LISTA SIMPLE

COMPLEJIDAD TEMPORAL Y ESTRUCTURAS DE DATOS

Comenzando, se procederá a analizar la complejidad temporal al resolver el servicio 4 con un arreglo dinámico y una lista simple, y a detallar las estructuras de datos utilizadas para almacenar en memoria la colección de empresas.

En la primera entrega del trabajo se tomó la decisión de recorrer el arreglo dinámico celda por celda para conocer la cantidad de empleados de cada empresa y de esta manera saber si cumple con el rango especificado o no. De cumplir con el rango, el algoritmo almacenaba un puntero a la empresa en una lista desordenada. Esta lista es un objeto que se crea y se trabaja mediante los métodos de la clase ListaSimple parametrizada. En la misma se detalló un struct NodoLista, donde se apunta a un elemento T genérico (en este caso un puntero a Empresa) y al siguiente elemento de la lista. En este caso, el recorrido para la carga de la colección es completo, y la complejidad temporal de la búsqueda en ese servicio es O(n) siendo “n” la cantidad de elementos de la lista, ya que en el peor de los casos la cantidad de empresas que especifique el usuario para analizar serán todas las del ranking.

SERVICIO 4 CON PRIORITY SEARCH TREE

CONSTRUCCIÓN, DECISIONES DE DISEÑO Y UTILIZACIÓN

Principalmente, se distinguirá cómo se fue construyendo el algoritmo del Priority Search Tree haciendo hincapié en las decisiones de diseño tomadas para construir las estructuras de almacenamiento y búsqueda, e implementar dicho algoritmo. Asimismo, se ampliará acerca de cómo se utilizó el Priority Search Tree para resolver el servicio solicitado.

Para la construcción del PST se empleó la estrategia que se detallará a continuación. Cabe aclarar que cada procedimiento explicado se resolvió mediante los métodos creados en una Clase llamada PrioritySearchTree. Los cuatro parámetros fundamentales para su construcción fueron: las empresas (elemento a ordenar), posición en el ranking (prioridad del elemento), cantidad de empleados

(clave del elemento) y la mediana de la cantidad de empleados de las empresas restantes (clave de distribución de los elementos).

Para implementar la construcción se realizaron los siguientes pasos:

- Replicar el Arreglo Original con punteros empresa (sin copiar información);
- Ordenar el arreglo según la cantidad de empleados de las empresas;
- Construir el árbol:
    - Calcular el elemento de menor prioridad del arreglo;
    - Llevar ese elemento al principio del arreglo;
    - Calcular la mediana del resto del arreglo;
    - Construir el nodo para agregarlo al árbol;
    - Encontrar el final del subarreglo izquierdo;
    - Dividir el arreglo en dos partes según la mediana;
    - Repetir el proceso para los dos nuevos subarreglos;

Para realizar la copia del arreglo original se creó uno auxiliar con las mismas características que el original, es decir, un arreglo de punteros a empresa. Para cargarlo, se recorre completamente el arreglo, y se copia celda por celda el puntero a la empresa correspondiente. En ningún momento se copian las empresas ni su información, sólo se repiten los punteros.

Una vez que obtenemos este arreglo se procede a ordenarlo por cantidad de empleados mediante el método de ordenamiento QuickSort. Este fue seleccionado ya que se ahorra el uso de una nueva estructura auxiliar (comparándolo con el método MergeSort, que posee una complejidad de O(n log n)) que ocupe espacio en la memoria. Aun así, tuvimos en cuenta que dicho algoritmo tiene excepciones en las que su complejidad alcanza el O(n^2) pero igualmente priorizamos el espacio en memoria.

Estos dos primeros pasos se realizaron debido a que posteriormente facilitan el cálculo de la mediana y la división de los nuevos subarreglos. Existe una manera de calcular la mediana sin ordenar los elementos pero su implementación para la construcción del PST se dificultaba significativamente, por lo que optamos por la primera opción.

Para la construcción del árbol se comienza buscando la posición del elemento de menor prioridad en el arreglo, es decir, la empresa que posee el puesto más elevado en el ranking.

En el momento que se encuentra esa empresa, se procede a trasladar el puntero a la primera celda del arreglo auxiliar mediante un corrimiento a derecha para no perder el orden de la estructura.

Posteriormente, para calcular la mediana, es imprescindible saber si la cantidad de elementos en el arreglo a analizar es par o impar. De ser impar basta con pararse a la mitad del arreglo para obtener la mediana. En el caso de que la cantidad sea par debemos obtener el promedio entre los dos elementos del medio. Para alcanzar estos elementos, se utilizó un cálculo matemático que nos ahorró recorridos en el arreglo.

Luego se construye el nodo que va a ser insertado automáticamente a la vuelta de la recursión. Este nodo posee, además de un puntero a izquierda y derecha, un puntero a empresa que nos permite saber la información de ella y un campo para contener la mediana de las empresas que se ubicarán en sus subárboles.

Una vez que se conoce la mediana y el nodo ha sido construido, se sucede a dividir el arreglo en dos, no literalmente, sino que mediante nuevos índices que nos van a permitir obtener dos “subarreglos”. Estos índices son: el inicio +1, el límite que está directamente a la izquierda de la mediana y tiene estrictamente un valor de cantidad de empleados inferior, este límite + 1 y finalmente el final del arreglo.

El límite final del subarreglo izquierdo se determina mediante una función que devuelve el valor menor estricto que se encuentra directamente a la izquierda de la mediana. En el caso de que no haya una empresa con una cantidad de empleados menor estricto que la mediana del lado izquierdo, la función devuelve un -1 y se procede a procesar un único subarreglo que será insertado del lado de los mayores del nodo en cuestión.

Los subarreglos repiten el proceso detallado anteriormente hasta que el arreglo sea indivisible. Llegado a este punto, se retornan los nodos construidos hasta el momento insertándose en la posición correspondiente. Esta inserción se realiza correcta y automáticamente ya que los llamados recursivos fueron realizados de manera estratégica.

Para construir el algoritmo de búsqueda se utilizó un método de la clase que recibe como parámetro una prioridad, la cantidad de empresas que se quieren analizar, un valor MIN (Mínimo) y otro MAX (Máximo) que toman el lugar del rango de cantidad de empleados. Estos tres parámetros son ingresados por el usuario cada vez que utiliza el servicio. Además, se recibe un cuarto parámetro, un objeto de tipo

ListaSimple, utilizado para coleccionar las empresas que cumplen con los requisitos. Se tomó la decisión de enviar esta colección por referencia y en el momento que finaliza el procedimiento de búsqueda, obtener todas las empresas en la lista.

Para implementar esta búsqueda se realizaron los siguientes pasos:

- Se comprueba que el nodo a analizar no sea NULL;
- Se verifica que el nodo no sea hoja:
    - Si es hoja y cumple los requisitos se agrega a la colección;
    - Si NO es hoja:
        - Se agrega solo si cumple los requisitos;
        - Se hace un llamado recursivo de los subárboles que puedan contener empresas dentro del rango.

Para que cumpla y sea agregado a la colección, el nodo debe contener el puntero a una empresa que tenga una cantidad de empleados mayor o igual a MIN y menor o igual a MAX. Además, esta empresa debe tener una posición en el ranking menor o igual al parámetro Prioridad.

Luego para proseguir analizando el árbol, si el nodo en cuestión posee el campo Mediana ocupado por un valor mayor estricto que el MIN se deben procesar el subárbol izquierdo. También, si el valor es menor o igual que MAX se debe hacer un llamado recursivo con el subárbol derecho. Si la mediana excede a estas condiciones, el subárbol correspondiente no es examinado.

Otro parámetro que nos permite cortar con la búsqueda es la prioridad, es decir, si un nodo no es hoja, y posee una empresa con una posición en el ranking que es mayor que Prioridad, todos sus subárboles quedan excluidos del procedimiento ya que no cumplirían con los requisitos. Este razonamiento es correcto debido a que la organización del Priority Search Tree nos asegura que a medida que descendemos en los niveles, la prioridad siempre aumenta hacia ambos nodos hijos.

Por último, si encontramos un nodo que es hoja, éste es estudiado y agregado a la colección de ser necesario, dando por finalizado así el procedimiento de búsqueda.

COMPLEJIDAD TEMPORAL

A continuación, se efectuará un análisis detallado de la complejidad temporal de la resolución del servicio requerido completo. No está de más aclarar que para

calcular la complejidad temporal del servicio, es necesario analizar paso por paso la construcción del árbol y luego el costo de realizar una búsqueda en el mismo. Se denotará una variable “n” que referirá al tamaño del arreglo.

Para la construcción, se comienza creando una copia del arreglo original que conlleva un recorrido lineal, es decir, una complejidad O(n).

Luego se requiere un ordenamiento del mismo a partir del método QuickSort, que conlleva una complejidad O(n log n) en el caso promedio, y O(n^2) en algunas excepciones, como fue mencionado anteriormente.

Una vez hecho esto, se ingresa en el procedimiento recursivo que comienza buscando el elemento de menor prioridad, esto sobrelleva una complejidad O(n) ya que en el peor de los casos la empresa con menor prioridad está apuntada por un puntero que se encuentra al final del arreglo.

Para llevar ese elemento de menor prioridad al principio del arreglo se debe realizar un corrimiento a derecha. Este desplazamiento, en el peor de los casos, será desde el final del arreglo al inicio -1, por lo que finalmente su complejidad es O(n).

Luego, se invoca el procedimiento que nos permite calcular la mediana. Al retornar un número que es obtenido por un cálculo matemático, la complejidad de este algoritmo es O(1). Dado que la creación del nodo y la división en subarreglos son operaciones constantes, sus respectivas complejidades temporales también estarán acotadas por O(1).

A su vez, encontrar el límite del subarreglo izquierdo conlleva una complejidad O(n) debido a que es un recorrido lineal.

Finalmente, como los procedimientos anteriormente mencionados se repiten una cantidad de veces dependiente del tamaño del arreglo y en consecuencia de que el llamado recursivo parte al arreglo en dos, la complejidad temporal total de la construcción del árbol es de O(n log n).

Para la búsqueda en el PST es posible alcanzar una complejidad O(log n) siempre y cuando los parámetros permitan acotar en algún punto la búsqueda. Esto es así ya que una de las características principales del Priority Search Tree es que siempre se obtendrá un árbol lo más balanceado posible a la hora de su construcción. En el peor de los casos, la diferencia entre los subárboles será de un solo nodo, lo que permite ganar mucho en eficiencia cuando se requiere realizar la búsqueda. En cada instancia que se presenta, se pueden llegar a descartar dos subárboles enteros en el caso de superar la prioridad. Además, la mediana posibilitará limitar la búsqueda

a nodos que sean posibles candidatos para la colección. Si la mediana supera el máximo se continúa por el subárbol izquierdo y en el caso inverso, se recorre el subárbol derecho. Por último, si no supera el máximo ni es menor que el mínimo, se procede a analizar ambos subárboles.

CÓDIGO FUENTE

En esta segunda entrega se agregó un nuevo servicio en el menú que permite realizar la búsqueda de una determinada cantidad de empresas que cumplen con un rango especificado de cantidad de empleados utilizando la nueva estructura (PST). El código fuente será el de la primera entrega junto a sus correcciones, más la adaptación y agregado del cuarto servicio.

Main.cpp

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

using namespace std;

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

servicios.h

```
#ifndef SERVICIOS_H_INCLUDED
```

```
#define SERVICIOS_H_INCLUDED
#include "Empresa.h"
#include "ABB.h"
#include "ListaSimple.h"
#include "PrioritySearchTree.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 ListarEmpresasPST(PrioritySearchTree * ArbolPrioridad,ListaSimple<Empresa
*> & S, unsigned int tamanioArreglo);
void Resolver(Empresa * Arreglo,ABB * Arbol,PrioritySearchTree * ArbolPrioridad,
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"
#include "PrioritySearchTree.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- Listar una cantidad de empresas dada, en un determinado rango,
utilizando un Priority Search Tree" << endl;
    cout << "5- Salir" << endl;
```

```
    cout << endl;
    cout << "Elija el servicio a realizar: ";
    cin >> Opcion;
    while ((Opcion != 1) && (Opcion != 2) && (Opcion != 3) && (Opcion != 4) &&
(Opcion != 5)){
        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 ListarEmpresasPST(PrioritySearchTree * ArbolPrioridad,ListaSimple<Empresa
*> & S, unsigned int tamanioArreglo)
{
    unsigned int Prioridad;
    unsigned int RangoMax;
    unsigned int RangoMin;
    cout <<  "Cuantas empresas del ranking se van a analizar? " << endl;
    cin >> Prioridad;
    while ((Prioridad <= 0) || (Prioridad > tamanioArreglo)){
        cout << "Ingrese una cantidad valida" << endl;
        cin >> Prioridad;}
    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;
        ArbolPrioridad->BuscarPST(Prioridad,RangoMin,RangoMax,S);
    }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,PrioritySearchTree * ArbolPrioridad,
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,ArbolPrioridad,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,ArbolPrioridad,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,ArbolPrioridad,tamanioArreglo);
                //delete ListaEmpleados;
                break;}
        case 4: {
                ListaSimple<Empresa *> S;
                ListarEmpresasPST(ArbolPrioridad,S,tamanioArreglo);
                cout << endl;
                if (S.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(S);}
                Resolver(Arreglo,Arbol,ArbolPrioridad,tamanioArreglo);
                //delete S;
                break;}

        case 5: {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
}
```

PrioritySearchTree.h

```
#ifndef PRIORITYSEARCHTREE_H
#define PRIORITYSEARCHTREE_H
#include "Empresa.h"
#include "ListaSimple.h"


class PrioritySearchTree
{
   public:
      PrioritySearchTree();
      void ConstruirArbol(Empresa * ArregloOriginal, int inicio, int fin);
      void BuscarPST(int P, int MIN, int MAX,ListaSimple<Empresa *> & S);
      virtual ~PrioritySearchTree();

   private:
      struct NodoPST{
      Empresa * puntEmpresa;
      float Mediana;
      NodoPST * der;
      NodoPST * izq;};
      NodoPST * CrearArbol(Empresa * Arreglo[], int inicio, int fin);
      void CrearNodoPST(Empresa * Dato, float Mediana, NodoPST * &
NodoNuevo);
      int ElementoMenorPrioridad(Empresa * Arreglo[], int inicio, int fin);
      void CorrimientoDerecha(Empresa * Arreglo[], int inicio, int limite, Empresa *
Dato);
      float CalcularMediana(Empresa * Arreglo[], int inicio, int fin);
      void EliminarArbol(NodoPST * Raiz);
      int IzquierdaAMediana(Empresa * Arreglo[], int inicio, float Mediana);
      bool Eshoja(NodoPST * Raiz) const;
      void BuscarPST(NodoPST * Nodo,int P, int MIN, int MAX,ListaSimple<Empresa
*> & S);
      int UbicarPivote(Empresa * Arreglo[], int ini, int fin);
      void OrdenarPorQuicksort(Empresa * Arreglo[], int ini, int fin);
```

```
        void CopiarArreglo(Empresa * arreglo, Empresa * auxiliar[], int
tamanio_arreglo);
        NodoPST * Raiz;
        NodoPST * Nodo;
};

#endif // PRIORITYSEARCHTREE_H
```

PrioritySearchTree.cpp

```
#include "PrioritySearchTree.h"
#include "ListaSimple.h"
#include "Empresa.h"

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

void PrioritySearchTree::BuscarPST(NodoPST * Nodo ,int P, int MIN, int
MAX,ListaSimple<Empresa *> & S)
{
   if(Nodo != NULL){
      if (Eshoja(Nodo)){
         if ((Nodo->puntEmpresa->obtenerPosRanking() <= P) && (MIN <= Nodo-
>puntEmpresa->obtenerCantEmpleados()) && (Nodo->puntEmpresa-
>obtenerCantEmpleados() <= MAX))
            S.Agregar(Nodo->puntEmpresa);}
      else
         if (Nodo->puntEmpresa->obtenerPosRanking() <= P){
            if ((MIN <= Nodo->puntEmpresa->obtenerCantEmpleados()) && (Nodo-
>puntEmpresa->obtenerCantEmpleados() <= MAX))
               S.Agregar(Nodo->puntEmpresa);
            if (MIN < Nodo->Mediana)
               BuscarPST(Nodo->izq,P,MIN,MAX,S);
            if (MAX >= Nodo->Mediana)
               BuscarPST(Nodo->der,P,MIN,MAX,S);}}
}

void PrioritySearchTree::BuscarPST(int P, int MIN, int MAX,ListaSimple<Empresa *>
& S)
{
```

```
   if(Raiz != NULL){
      if (Eshoja(Raiz)){
         if ((Raiz->puntEmpresa->obtenerPosRanking() <= P) && (MIN <= Raiz-
>puntEmpresa->obtenerCantEmpleados()) && (Nodo->puntEmpresa-
>obtenerCantEmpleados() <= MAX))
            S.Agregar(Raiz->puntEmpresa);}
      else
         if (Raiz->puntEmpresa->obtenerPosRanking() <= P){
            if ((MIN <= Raiz->puntEmpresa->obtenerCantEmpleados()) && (Raiz-
>puntEmpresa->obtenerCantEmpleados() <= MAX))
               S.Agregar(Raiz->puntEmpresa);
            if (MIN < Raiz->Mediana)
               BuscarPST(Raiz->izq,P,MIN,MAX,S);
            if (MAX >= Raiz->Mediana)
               BuscarPST(Raiz->der,P,MIN,MAX,S);}}
}

int PrioritySearchTree::UbicarPivote(Empresa * Arreglo[], int ini, int fin)
{
   Empresa * Copia;
   while(ini<fin){
      while(((Arreglo[fin]->obtenerCantEmpleados()) >= (Arreglo[ini]-
>obtenerCantEmpleados()))&& (ini<fin))
         fin--;
      Copia=new Empresa();
      *Copia=*Arreglo[fin];
      *Arreglo[fin]=*Arreglo[ini];
      *Arreglo[ini]=*Copia;
      while((Arreglo[ini]->obtenerCantEmpleados() <= Arreglo[fin]-
>obtenerCantEmpleados())&& (ini<fin))
         ini++;
      Copia=new Empresa();
      *Copia=*Arreglo[ini];
      *Arreglo[ini]=*Arreglo[fin];
      *Arreglo[fin]=*Copia;}
   return ini;
}

void PrioritySearchTree::OrdenarPorQuicksort(Empresa * Arreglo[], int ini, int fin)
{
   if(ini<fin){
      int PosCorrectaPiv = UbicarPivote(Arreglo,ini,fin);
      OrdenarPorQuicksort(Arreglo,ini,PosCorrectaPiv-1);
      OrdenarPorQuicksort(Arreglo,PosCorrectaPiv+1,fin);}
```

```
}

void PrioritySearchTree::CopiarArreglo(Empresa * arreglo, Empresa * auxiliar[], int
tamanio_arreglo)
{
   for (int i = 0; i < tamanio_arreglo; i++) {
      //auxiliar[i].prioridad = i + 1;
      auxiliar[i] = new Empresa();
      *auxiliar[i] = arreglo[i];
   }
}

void PrioritySearchTree::CrearNodoPST(Empresa * Dato, float Mediana,NodoPST *
& NodoNuevo)
{
   NodoNuevo = new NodoPST;
   NodoNuevo->puntEmpresa = Dato;
   NodoNuevo->Mediana = Mediana;
   NodoNuevo->izq = NULL;
   NodoNuevo->der = NULL;
}

int PrioritySearchTree::ElementoMenorPrioridad(Empresa * Arreglo[], int inicio, int
fin)
{
   int menor = inicio;
   inicio++;
   while (inicio <= fin) {
      if (Arreglo[inicio]->obtenerPosRanking() < Arreglo[menor]-
>obtenerPosRanking())
         menor = inicio;
      inicio++;}
   return menor;
}

void PrioritySearchTree::CorrimientoDerecha(Empresa * Arreglo[], int inicio, int
limite, Empresa * Dato)
{
   for (limite; limite > inicio; limite--)
      Arreglo[limite] = Arreglo[limite-1];
   Arreglo[inicio] = Dato;
}

float PrioritySearchTree::CalcularMediana(Empresa * Arreglo[], int inicio, int fin)
```

```
{
    if ((fin - inicio + 1)%2 == 0)
        return (Arreglo[fin - ((fin - inicio + 1)/2)]->obtenerCantEmpleados() + Arreglo[fin
- ((fin - inicio + 1)/2) + 1]->obtenerCantEmpleados())/2.0;
    else
        return Arreglo[(fin - (fin - inicio)/2)]->obtenerCantEmpleados();
}

int PrioritySearchTree::IzquierdaAMediana(Empresa * Arreglo[], int inicio, float
Mediana)
{
    if (Arreglo[inicio]->obtenerCantEmpleados() < Mediana){
        int i = inicio;
        while (Arreglo[i]->obtenerCantEmpleados() < Mediana)
            i++;
        return i-1;}
    else
        return -1; //Significa que el valor en esa celda posiblemente sea igual a la
mediana y en todas las celdas siguientes tambien
}


PrioritySearchTree::NodoPST* PrioritySearchTree::CrearArbol(Empresa * Arreglo[],
int inicio, int fin)
{
    if (inicio <= fin)
    {
        int PosicionMenorPrioridad = ElementoMenorPrioridad(Arreglo, inicio, fin);
        CorrimientoDerecha(Arreglo, inicio, PosicionMenorPrioridad,
Arreglo[PosicionMenorPrioridad]);
        float Mediana = -1;
        if (inicio < fin)
            Mediana = CalcularMediana(Arreglo, inicio + 1, fin);
        NodoPST * NodoNuevo;
        CrearNodoPST(Arreglo[inicio], Mediana,NodoNuevo);
        if (inicio < fin){
            int MenorAMediana = IzquierdaAMediana(Arreglo,inicio + 1,Mediana);
            if (MenorAMediana != -1){
                NodoNuevo->izq = CrearArbol(Arreglo, inicio + 1, MenorAMediana);
                NodoNuevo->der = CrearArbol(Arreglo, MenorAMediana + 1, fin);}
            else
                NodoNuevo->der = CrearArbol(Arreglo, inicio + 1, fin);}
        return NodoNuevo;
    }
```

```
    else
        return NULL;
}

void PrioritySearchTree::ConstruirArbol(Empresa * ArregloOriginal, int inicio, int
tamanio_arreglo)
{
    Empresa * Auxiliar[tamanio_arreglo];
    CopiarArreglo(ArregloOriginal, Auxiliar, tamanio_arreglo);
    OrdenarPorQuicksort(Auxiliar, 0, tamanio_arreglo-1);
    Raiz = CrearArbol(Auxiliar, inicio, tamanio_arreglo-1);
}

bool PrioritySearchTree::Eshoja(NodoPST * Raiz) const
{
    if ((Raiz->izq == NULL) && (Raiz->der == NULL))
        return true;
    else
        return false;
}

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

PrioritySearchTree::~PrioritySearchTree()
{
    EliminarArbol(Raiz);
}
```

Conclusión

Para finalizar, se puede concluir que la construcción de esta particular estructura expuso que la eficiencia de la búsqueda por rangos se puede resolver de una manera más eficiente en comparación con la primera entrega. Aun así, esta reducción en la complejidad se ve equilibrada por el elevado costo que conlleva la construcción. Por esa razón, antes de decidirse a utilizar un Priority Search Tree es recomendable analizar en profundidad el contexto del problema.

Debido a que en este caso la construcción del árbol se realiza una vez al día y se utiliza para buscar repetidas veces, se podría afirmar que la implementación de este tipo de estructuras es la más acertada.