Trabajo práctico especial

Primera parte

Grupo: 22

Integrantes:

Juan Ignacio Barthes

[juanbarthes@gmail.com](mailto:juanbarthes@gmail.com)

Marco Daniel Cordoba

[marcocordoba02@gmail.com](mailto:marcocordoba02@gmail.com)

Introducción

En este informe se intentará explicar la implementación de la primera parte del trabajo práctico especial de Análisis y Diseño de Algoritmos I propuesto por la cátedra.

Dicho proyecto consiste en la realización de la clase Lista y de la clase Árbol en el lenguaje C++. Estas se utilizarán junto con archivos de texto proporcionados por la cátedra, para crear estructuras en las que puedan almacenarse y buscar palabras, analizando si están o no contenidas, y el costo de dicho análisis.

Clase Lista:

Se nos ocurrieron dos posibles maneras para implementar la clase lista, la primera fue crearla mediante un arreglo, debido a que son una estructura de acceso arbitrario, lo cual beneficiaria la ejecución de tareas como agregar, eliminar y buscar elementos en posiciones específicas. El problema con los arreglos está en su uso de memoria

si definimos por ejemplo un arreglo de 5000 celdas y solo vamos a aprovechar 200, el arreglo de todas formas mantendrá en uso las 5000 celdas, lo cual es un desperdicio de memoria.

por esta razón optamos por implementarla como una lista de punteros, una estructura formada por nodos en la cual cada nodo posee un puntero al siguiente. Si bien una lista de punteros tiene un acceso secuencial, es decir, se debe pasar por todos los nodos anteriores para poder acceder a uno en específico, como compensación estamos obteniendo una estructura dinámica, capaz de incrementar o decrementar su tamaño según sea necesario, por lo que estaremos reservando sólo la memoria justa y necesaria.

a continuación se muestra la especificación en el lenguaje Nereus

correspondiente a la clase Lista.

Class Lista [elemento];

imports nat;

basic constructors iniclista, agregarelem;

effective

type Lista;

operations

iniclista: → Lista;

agregarinicio: Lista\*elemento → Lista;

agregarfin: Lista\*elemento → Lista;

longlista: Lista → nat;

agregarelem: Lista(L)\*elemento(E)\*nat(N) → Lista;

pre: (N >= 1) and (N <= longlista(L)+1);

listavacia: Lista → boolean;

pertenece: Lista(L)\*elemento(E) → boolean;

pre: not (listavacia(L));

eliminarelemento: Lista(L)\*elemento(E) → Lista;

pre: (pertenece(L\*E));

eliminarlista: Lista(L) → Lista;

pre: not (listavacia(L));

**Explicación, Clasificación y Complejidad Temporal de las Funciones:**

* inclista: es la inicialización de la lista, es la constructora básica de la lista y su complejidad temporal es O(1).
* agregarinicio: se encarga de agregar un elemento en la primera posición de lista, es un constructor modificador y su complejidad temporal es O(1).
* agregarfin: se encarga de agregar un elemento en la última posición de la lista, es un constructor modificador y su complejidad temporal es O(1).
* longlista: cuenta la cantidad de nodos que tiene la lista y devuelve un número natural, es una observadora y su complejidad temporal es O(1).
* agregarelem: se encarga de agregar un elemento en una posición específica, es una constructora básica y su complejidad temporal es O(1).
* listavacia: éste se encarga de verificar si la lista está vacía o no, es una observadora y complejidad temporal es O(1).
* pertenece: verifica si un elemento existe o no en la lista, es una observadora y su complejidad temporal es O(n).
* eliminarelemento: se encarga de eliminar un elemento en la lista, es una modificadora modificadora y su complejidad temporal es O(n).
* eliminarlista: se encarga de eliminar todos los nodos de la lista, es una constructora modificadora y su complejidad temporal es O(n).

Para disminuir el costo de las operaciones de búsqueda o adición de elementos, definimos dos variables auxiliares de tipo puntero para guardar en todo momento las direcciones de memoria del primer y último elemento de la lista respectivamente, esto agiliza el proceso de agregado en dichas posiciones, evitando recorrer la lista innecesariamente, aunque en el caso de añadir elementos en posiciones distintas a inicio o fin, esto se vuelve inevitable.

También definimos 2 variables de tipo entero:

la primera, contador, cuyo objetivo es llevar en todo momento la cantidad de elementos de la lista ya que cada vez que se agrega un elemento, contador se ve incrementado y cada vez que se quita un elemento, decrece. De esta manera podemos simplemente consultar su valor, en lugar de recorrer toda la lista.

Uno de los requisitos de nuestro programa es saber que tan costoso resulta buscar elementos en las estructuras, para lo cual existe costo, que funciona como un contador que es utilizado en las funciones de búsqueda y nos permite saber cuántas iteraciones fueron necesarias para determinar si existe o no un elemento específico.

Clase Arbol

Para esta clase creamos una estructura llamada Árbol Binario de Búsqueda, conformada por nodos ordenados, los cuales poseen cada uno, dos punteros: el izquierdo lo vincula a nodos con elementos menores al del nodo raíz, y el de la derecha a nodos con elementos mayores, estas características se repiten en cada nodo hasta llegar a las hojas, que son los últimos nodos del arbol, y no hay nada más a continuación de ellos.

A continuación se deja la implementación de la clase Árbol en el lenguaje Nereus:

class Arbol [elemento];

imports nat, Lista;

basic constructors inicarbol, agregar;

effective

type Arbol;

operations

inicarbol: → Arbol;

agregar: Arbol\*elemento → Arbol;

arbolvacio: Arbol → Boolean;

existe: Arbol(A)\*elemento(E) → Boolean;

pre: not (arbolvacio(A));

tamaño: Arbol → nat;

listar: Arbol\*Lista → Lista;

borrararbol: Arbol(A) → Arbol;

pre: not (arbolvacio(A));

**Explicación, Clasificación y Complejidad Temporal de las Funciones:**

* inicarbol: es la encargada de inicializar la estructura. Es una constructora básica y su complejidad es O(1).
* agregar: añade un nuevo elemento al arbol. Es una constructora básica y su complejidad es O(log n).
* arbolvacio: sirve para verificar si el arbol tiene elementos o no. Es una observadora y su complejidad es O(1).
* existe: verifica si un elemento pertenece o no al arbol. Es una observadora y su complejidad es O(log n).
* tamaño: devuelve la cantidad de nodos que posee el arbol. Es una observadora y su complejidad es O(1).
* listar: genera una lista a partir de recorrer el arbol in-order. Es una modificadora y su complejidad es O(n).
* borrararbol: Elimina todos los elementos del arbol. Es una modificadora y su complejidad es O(n).

Nuevamente declaramos una variable para guardar el valor de la cantidad de elementos del arbol, y otra para analizar el costo de realizar las búsquedas dentro del mismo.

Al poseer un orden en todo momento, el arbol nos brinda la ventaja de

no tener que pasar por todos los elementos de su colección para poder llegar a uno en especifico, ya que desde cualquier nodo distinto de vacío, podemos elegir uno entre dos caminos para seguir avanzando, ya que dependiendo de si el elemento buscado es menor o mayor al elemento actual, podemos suponer hacia donde se encuentra.

**Ejecución del Proyecto:**

En esta etapa utilizaremos las clases explicadas anteriormente, junto con los archivos de texto proporcionados por la cátedra para codificar un nuevo programa en C++.

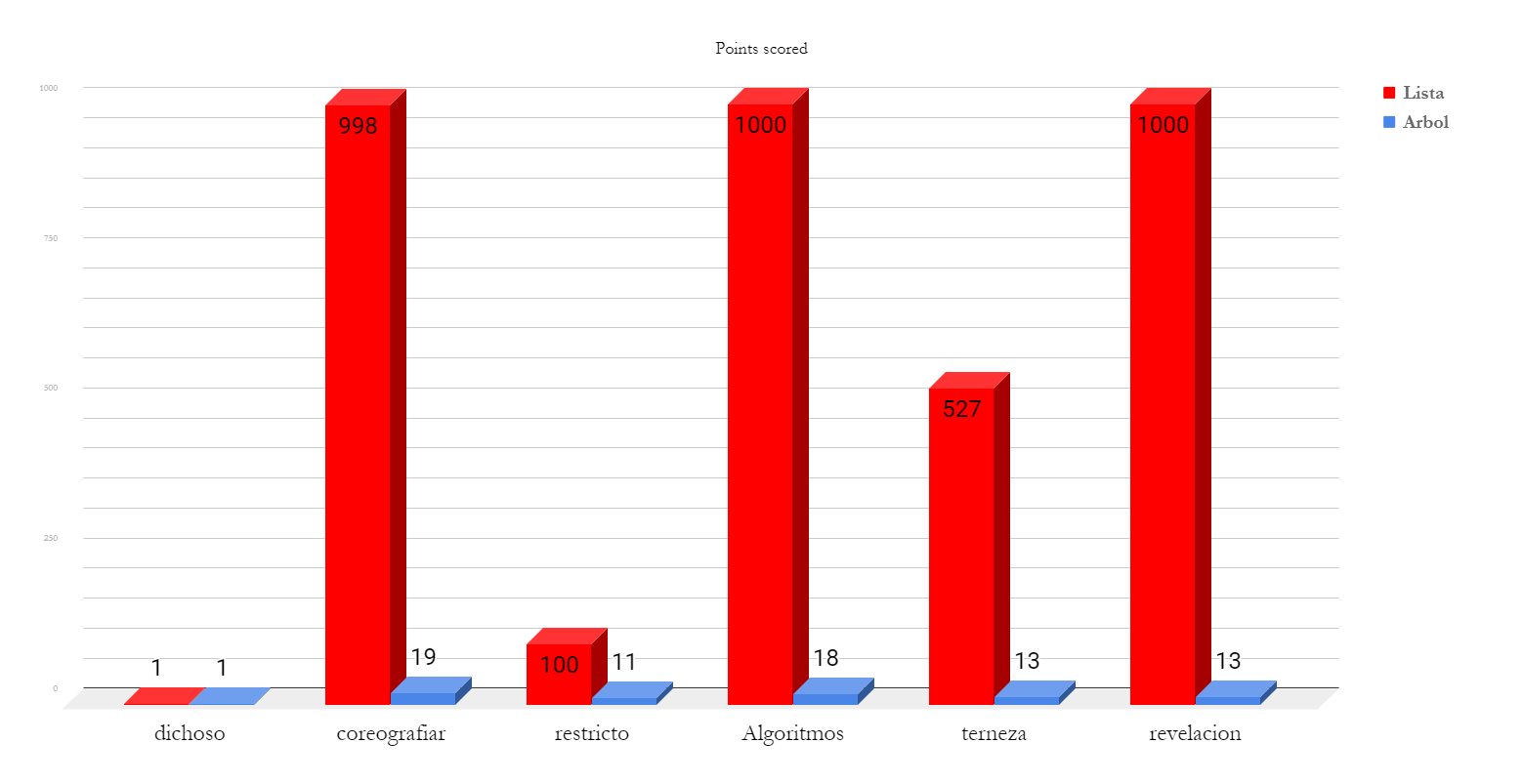
La carpeta de archivos posee tres archivos de colección de palabras y tres de búsqueda de palabras, enumerados respectivamente. A partir del archivo **Colección**, se genera una estructura de strings que contiene todas las palabras del archivo, luego, mediante la lectura del archivo **Búsqueda**,se procesa toda la estructura para finalmente escribir en un archivo **Resultado**, si las palabras existen dentro de la estructura, y cuan costoso fue averiguarlo.

Este proceso se realizará tanto en una Lista como en un Arbol y se repetirá por cada uno de los tres archivos de **Colección.**

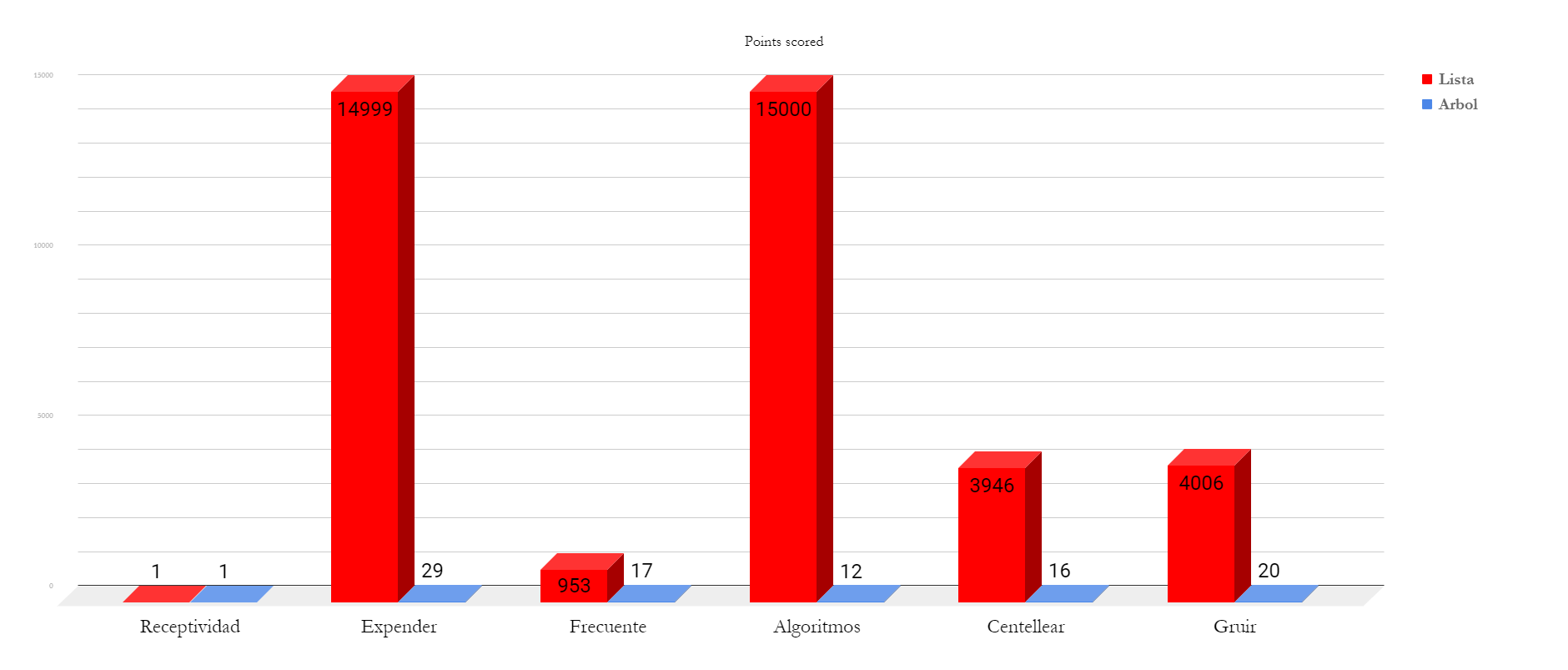
**Análisis de Resultados:**

A continuacion se puede observar una grafica comparativa del costo que conlleva realizar las busquedas de las palabras en cada estructura con los 3 archivos:

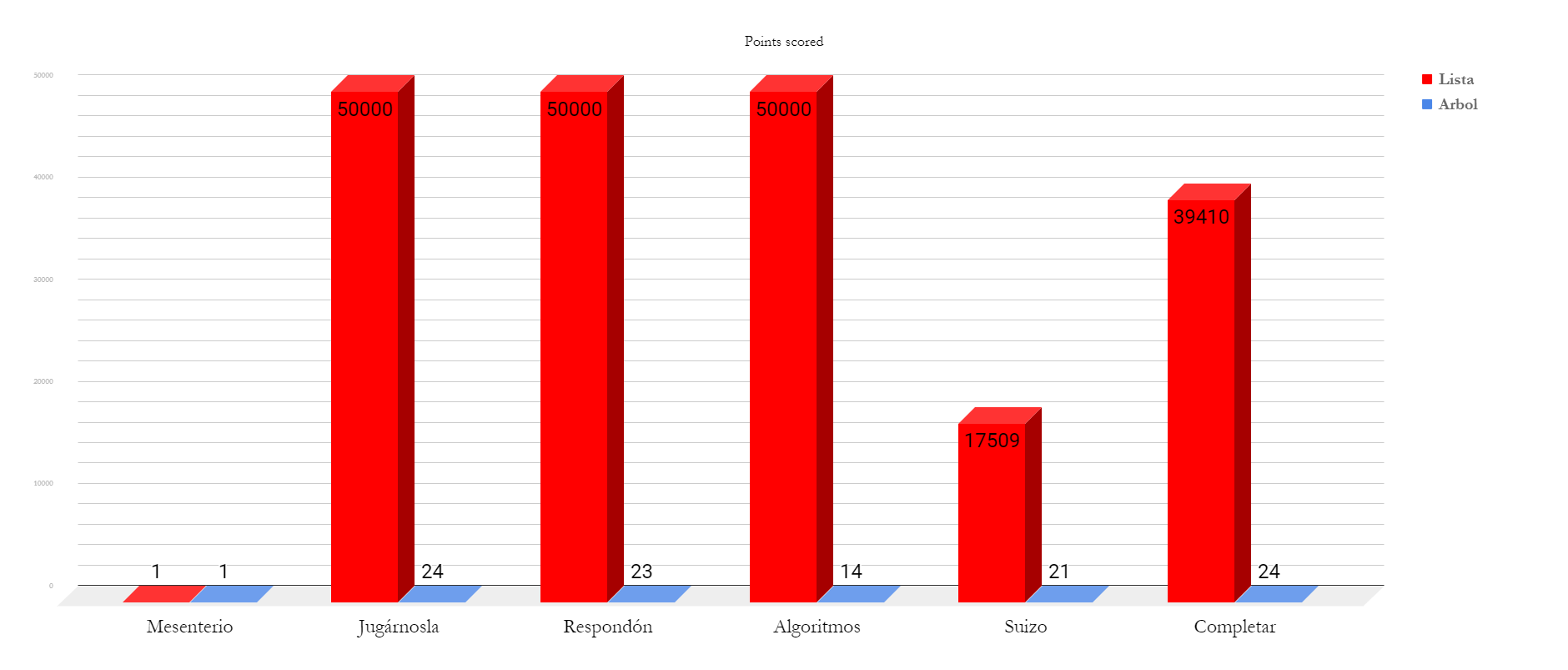
Colección 1:



Colección 2:



Colección 3:



Como se puede apreciar en los graficos, la busqueda de las palabras en la lista resulto muchisimo mas costosa que en el arbol en todos los casos, esto se debe a que a diferencia de la lista, el arbol ya se encontraba ordenado, y puede subdividirse en arboles mas pequeños. La lista se encontraba desordenada, ademas de que solo puede recorrerse en un sentido, en cambio un arbol puede subdividirse en subarboles mas pequeños, esto hace que bajo nuestro criterio de busqueda recursiva, el rango de busqueda se vaya haciendo cada vez mas pequeño, permitiendonos determinar si existe o no un elemento, a la vez que se evita pasar por todos los caminos incorrectos.

**Conclusión:**

El objetivo de este proyecto fue la implementacion de dos clases diferentes y realizar un analisis del costo de buscar las palabras solicitadas.

Pudimos percibir una diferencia significativa entre ambas estructuras, que se debe a dos factores importantes, primero que el arbol estaba ordenado, segundo que este utiliza un algoritmo de busqueda recursiva y puede subdividir el problema en casos mas pequeños, aumentando la eficiencia en comparacion con la estructura de lista simple.