Introducción

En el presente informe se dará a conocer y se hará su respectiva explicación de cada parte y funcionalidad implementada, así como métodos y clases auxiliares, que componen al trabajo final de la materia Complejidad temporal, estructuras de datos y algoritmos. El enunciado de este mismo consta de un buscador de coincidencias aproximadas que tiene por objetivo indexar datos almacenados en un archivo csv y realizar búsquedas sobre los mismos, para el cual su funcionamiento estará basado en la estructura de datos conocida como árbol BK.
El objetivo de este trabajo es implementar y afianzar los conocimientos adquiridos durante el dictado de clases, desarrollando y aplicando los criterios más adecuados para cada solución dependiendo del problema/consigna y su contexto.

Clases, métodos y metodologías auxiliares

En esta sección se pasará a dar detalle acerca de aquellos recursos que fueron utilizados en el desarrollo del trabajo final, tanto para resolver los problemas como para que se realicen de manera mucho más eficiente.

1. StringBuilder: Esta clase me fue de mucha ayuda para poder desarrollar la consulta 3 y poder plasmar mi idea de resolución en el método que lo requería. El funcionamiento es simple: StringBuilder es usado para modificar una cadena sin la necesidad de crear un objeto en específico, a su vez mejora el rendimiento al concatenar muchas cadenas en un bucle. Esta última afirmación es la clave de por que lo use, en la consulta 3 se tenía que estar constantemente agregando strings a una cadena de texto, para lo cual si se usaba una variable de tipo string donde se irán insertando los datos, se estarían creando y desechando una gran cantidad de objetos ya que el objeto String es inmutable. A continuación se detallan los 3 métodos que se utilizaron:
    - ➔ AppendLine(): Agrega un salto de línea.
    - ➔ Append(): Se encarga de agregar al StringBuilder aquello que se pase como parámetro.
    - ➔ ToString(): Convierte al StringBuilder en un string.

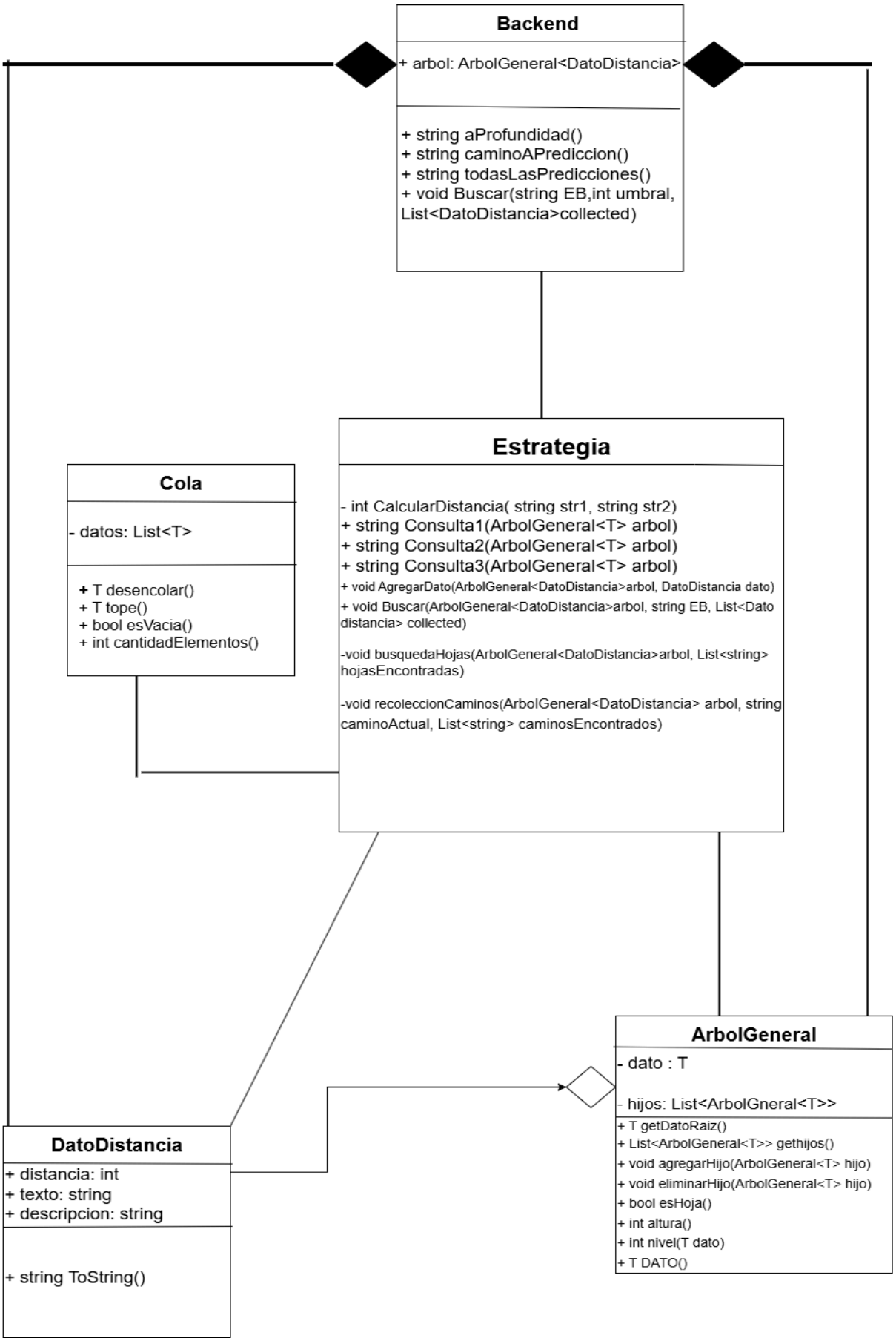
2. String.Join(): Este método me fue de gran ayuda para poder realizar las consultas 1 y 2, ya que su labor principal consiste en concatenar los elementos existentes de un arreglo o lista en un solo string, utilizando un separador específico entre cada elemento agregado. En mi implementación se utilizó “\n” como uno de sus parámetros para poder mostrar cada dato separado por una línea.

3. Optimización del método Buscar(): Dentro de la consulta 3 se inicializan las variables *distanciaMinima* y *distanciaMaxima* las cuales forman parte del mecanismo clave de

optimización del método en sí. Su principal propósito es crear un filtro que evita que el algoritmo explore innecesariamente cada nodo del árbol. De manera que se basa en la distancia calculada al nodo actual (distancia) y el umbral de tolerancia, estas variables definen un rango de búsqueda válido. Esto funciona gracias a una propiedad matemática (llamada desigualdad triangular), que garantiza que cualquier resultado válido debe estar en una rama cuya distancia almacenada se encuentre dentro de este rango. El algoritmo entonces solo visita los hijos que cumplen esta condición, ahorrando tiempo de búsqueda y recursos.

Desarrollo

Para poder comenzar nuestro desarrollo es importante conocer la arquitectura y cómo se relacionan los objetos entre sí, conociendo de igual manera todos sus métodos. Para esto es muy importante la realización de un diagrama UML que permita comprender lo anteriormente solicitiado.


Backend
+ arbol: ArbolGeneral<DatoDistancia>
+ string aProfundidad()
+ string caminoAPrediccion()
+ string todasLasPredicciones()
+ void Buscar(string EB,int umbral, List<DatoDistancia>collected)
Estrategia
- int CalcularDistancia(string str1, string str2)
+ string Consulta1(ArbolGeneral<T> arbol)
+ string Consulta2(ArbolGeneral<T> arbol)
+ string Consulta3(ArbolGeneral<T> arbol)
+ void AgregarDato(ArbolGeneral<DatoDistancia>arbol, DatoDistancia dato)
+ void Buscar(ArbolGeneral<DatoDistancia>arbol, string EB, List<Dato distancia> collected)
-void busquedaHojas(ArbolGeneral<DatoDistancia>arbol, List<string> hojasEncontradas)
-void recoleccionCaminos(ArbolGeneral<DatoDistancia> arbol, string caminoActual, List<string> caminosEncontrados)
Cola
- datos: List<T>
+ T desencolar()
+ T tope()
+ bool esVacia()
+ int cantidadElementos()
ArbolGeneral
- dato : T
- hijos: List<ArbolGneral<T>>
+ T getDatoRaiz()
+ List<ArbolGeneral<T>> gethijos()
+ void agregarHijo(ArbolGeneral<T> hijo)
+ void eliminarHijo(ArbolGeneral<T> hijo)
+ bool esHoja()
+ int altura()
+ int nivel(T dato)
+ T DATO()
DatoDistancia
+ distancia: int
+ texto: string
+ descripcion: string
+ string ToString()

Implementación de métodos

Una vez entendida la estructura de nuestro programa, así como los métodos que componen a cada clase que forma parte del mismo, podemos comenzar a desarrollar e implementar los métodos solicitados los cuales forman un total de 5.
Comenzaremos el detallamiento de cada uno en el orden en que fueron solicitados. Cabe aclarar que los dos primeros métodos(*AgregarDato()* y *Buscar()*) fueron pedidos de manera previa para su correspondiente y breve defensa mediante una exposición en clase, la cual tenía como objetivo demostrar el progreso de este mismo proyecto.
Se pasa a explicar cada funcionalidad a continuación:

- *AgregarDato (ArbolGeneral<DatoDistancia> arbol, DatoDistancia dato)* : Este método agrega un nuevo dato a un árbol BK, donde tanto el dato como el árbol son enviados como parámetro.

```
public void AgregarDato(ArbolGeneral<DatoDistancia> arbol, DatoDistancia dato)
{
    if (arbol.getDatoRaiz() == null)
    {
        arbol.DATO = dato;
    }

    else
    {
        int distancia = CalcularDistancia(arbol.getDatoRaiz().texto, dato.texto);
        bool estadoBusqueda = false;

        foreach (ArbolGeneral<DatoDistancia> hijoPuntero in arbol.getHijos())
        {
            if (hijoPuntero.getDatoRaiz().distancia == distancia)
            {
                estadoBusqueda = true;
                this.AgregarDato(hijoPuntero, dato);
                break;
            }
        }
        if (estadoBusqueda == false)
        {
            arbol.agregarHijo(new ArbolGeneral<DatoDistancia>(new DatoDistancia(distancia, dato.texto, dato.descripcion)));
        }
    }
}
```

Lo primero que se realiza es verificar si se trata de un árbol el cual se encuentra vacío, caso afirmativo el dato (el cual es una instancia de la clase DatoDistancia) que fue enviado como parámetro junto con el árbol pasa a ser la raíz de este mismo.
Caso contrario se procede a almacenar en una variable de tipo entera la distancia (obtenida gracias al método *CalcularDistancia()*) Levenshtein entre el texto que almacena el objeto ubicado en *la raíz del árbol* con el texto que se encuentra contenido en el objeto brindado. A su vez se inicializa una variable de tipo booleana en falso, la cual más adelante funcionará como una bandera.
El próximo paso a ejecutar es un bucle foreach el cual se encargará de buscar algún hijo del árbol pasado como parámetro el cual cuente con la misma distancia que la anteriormente calculada y almacenada en la variable *distancia*. Caso que logre encontrar algún hijo con esta condición, la bandera *estadoBusqueda* pasa ser verdadera y se llama de manera recursiva al mismo método para que se repita este ciclo de instrucciones dentro del hijo encontrado para luego finalizar con un break el cual evitará seguir buscando más coincidencias innecesarias.
Para finalizar, en el momento en que no se encuentre algún hijo que cumpla la condición anteriormente establecida por el *if*, volverá a pasar por otro condicional de este tipo para que en caso de que la bandera siga almacenando un booleano de tipo falso, se pase a

agregar un hijo (mediante el método *agregarHijo()*) en el último nodo al cual se llegó con la recursión.

- *Buscar(ArbolGeneral<DatoDistancia> arbol, string elementoABuscar, int umbral, List<DatoDistancia> collected)*: Retorna en la lista llamada collected el resultado de realizar una búsqueda de la cadena almacenada en *elementoABuscar* sobre el árbol BK almacenado en el parámetro árbol y con un nivel de tolerancia indicado por el parámetro umbral.

```
public void Buscar(ArbolGeneral<DatoDistancia> arbol, string elementoABuscar, int umbral, List<DatoDistancia> collected)
{
    if (arbol.getDatoRaiz() == null)
    {
        return;
    }
    else
    {
        int distancia = this.CalcularDistancia(elementoABuscar, arbol.getDatoRaiz().texto);

        int distanciaMinima = distancia - umbral;
        int distanciaMaxima = distancia + umbral;

        if (distancia <= umbral)
        {
            DatoDistancia datoActual = arbol.getDatoRaiz();
            collected.Add(new DatoDistancia(distancia, datoActual.texto, datoActual.descripcion));
        }

        foreach (var puntero in arbol.getHijos())
        {
            if (puntero.getDatoRaiz().distancia >= distanciaMinima && puntero.getDatoRaiz().distancia <= distanciaMaxima)
            {
                Buscar(puntero, elementoABuscar, umbral, collected);
            }
        }

    }
}
```

Al comenzar la ejecución de este método se procede a verificar de que el árbol pasado como parámetro no se encuentre vacío, caso de que así fuese se procede a retornar inmediatamente.
Si el árbol efectivamente se encuentra con datos en su interior, se procede a calcular la distancia entre el elemento a buscar y el texto que se encuentra almacenado en la raíz del árbol general. A continuación se procede a calcular tanto la distancia mínima como la distancia máxima (usando el parámetro umbral el cual especifica la tolerancia soportada), las cuales tienen como objetivo el optimizar la búsqueda aprovechando las propiedades de un árbol BK.
Caso de que la distancia calculada anteriormente entre la palabra a buscar y el nodo actual sea menor o igual al umbral, se procede a agregar a la lista collected un nuevo objeto de clase *DatoDistancia*, el cual tendrá el mismo texto y descripción que el dato original pero con su distancia ahora modificada, siendo que esta misma será reemplazada por la anteriormente calculada.
Por último se procede a usar un bucle *foreach* el cual se encargará de buscar aquellos hijos del árbol actual cuya distancia almacenada en el nodo se encuentre entre la distancia mínima y la distancia máxima calculada previamente. En caso de que encuentre a tal hijo, se procede a llamar de manera recursiva al método Buscar() siendo que se reemplaza al parámetro *árbol* por *puntero*(hijo del árbol en cuestión)

- *Consulta1 (ArbolGeneral<DatoDistancia> arbol): Retorna un texto con todas las hojas del árbol BK del sistema*.

```
public String Consulta1(ArbolGeneral<DatoDistancia> arbol)
{
    List<string> hojasEncontradas = new List<string>();

    busquedaHojas(arbol, hojasEncontradas);

    return String.Join("\n", hojasEncontradas);
}
```

Al momento de desarrollar el siguiente método tuve un inconveniente que definió en su totalidad como procedería en la creación de funcionalidades que generen el mismo problema. Si bien tenía una idea de cómo se podría resolver la consigna, mi propia solución no se estaba adhiriendo tanto a lo que retorna la consulta, es decir un string, como a la cantidad de parámetros formales. Por lo que mi solución fue el recurrir a un nuevo método privado el cual sería llamado por la misma consulta: *busquedaHojas()*
En la funcionalidad *Consulta1* unicamente se procede a crear una lista de strings, se llama al método auxiliar y por último se retorna un solo string el cual contiene a cada hoja por separada.

```
private void busquedaHojas(ArbolGeneral<DatoDistancia> arbol, List<string> hojasEncontradas)
{
    if (arbol.esHoja() == true)
    {
        hojasEncontradas.Add(arbol.getDatoRaiz().texto);
    }
    else
    {
        foreach (ArbolGeneral<DatoDistancia> hijo in arbol.getHijos())
        {
            busquedaHojas(hijo, hojasEncontradas);
        }
    }
}
```

El siguiente método toma como parámetros formales a un árbol General (el cual originalmente fue obtenido en la llamada de la *Consulta1()*) y una lista de strings (de manera similar con el árbol, la lista fue creada al principio de la *Consulta1()*).
Al comenzar su ejecución, lo primero que se verifica es si el árbol actual es una hoja, caso afirmativo se procede a agregar a la lista *hojasEncontradas* el texto que contiene la raíz.
En caso de que no fuera hoja, se comienza un bucle *foreach* el cual por cada hijo que tenga el árbol en cuestión, se procederá a llamar de manera recursiva al mismo método con la modificación de que ahora cada hijo va a ser analizado.
Cuando se complete cada llamada recursiva, la lista se encontrará actualizada con todas las hojas del árbol originalmente pasado como parámetro.

- *Consulta2 (ArbolGeneral<DatoDistancia> arbol): Retorna un texto que contiene todos los caminos hasta cada hoja.*

```
public String Consulta2(ArbolGeneral<DatoDistancia> arbol)
{
    List <string> caminosEncontrados= new List<string>();

    recoleccionCaminos(arbol,arbol.getDatoRaiz().texto,caminosEncontrados);

    return String.Join("\n", caminosEncontrados);
}
```

En la consulta 2 utilicé la misma metodología de resolución aplicada en la consulta 1, siendo que de igual manera se crea una lista de strings, se llama a un método auxiliar y se retorna un solo string que contiene todos los caminos por separado. Se procede a detallar el metodo *recoleccionCaminos()* :

```
private void recoleccionCaminos(ArbolGeneral<DatoDistancia> arbol,string caminoActual, List<string> caminosEncontrados)
{
    if (arbol.esHoja() == true)
    {
        caminosEncontrados.Add(caminoActual);
    }
    else
    {
        foreach (ArbolGeneral<DatoDistancia> hijo in arbol.getHijos())
        {
            string caminoActualizado= caminoActual + " ---> " + hijo.getDatoRaiz().texto;
            recoleccionCaminos(hijo,caminoActualizado,caminosEncontrados);
        }
    }
}
```

El segundo método auxiliar toma un total de tres parámetros: un arbol general (obtenido originalmente en la consulta 2), un string que contiene el camino formado hasta ese momento y una lista de strings que contendrá todos los caminos formados.
Para comenzar se verifica si el árbol actual es una hoja, caso afirmativo se procede a únicamente agregar el camino formado hasta ese entonces a la lista *caminosEncontrados*. Caso contrario, se procede a ejecutar un bucle *foreach* el cual se encargará de crear un nuevo string(*caminoActualizado*) el cual contendrá al camino formado hasta ahora más una concatenación del texto almacenado en el nodo del puntero del bucle(es decir su hijo). Luego de esto mismo se procede a llamar de manera recursiva al método actual con algunas modificaciones en los parámetros: el hijo ahora es quien va a ser analizado y el camino se reemplaza por aquel string inicializado anteriormente.
Al momento de finalizar todas las llamadas recursivas correspondientes, la lista *caminosEncontrados* contendrá todos los caminos del árbol.

- *Consulta3 (ArbolGeneral<DatoDistancia> arbol): Retorna un texto que contiene los datos almacenados en los nodos del árbol diferenciados por el nivel en que se encuentran.*

```
public string Consulta3(ArbolGeneral<DatoDistancia> arbol)
{
    System.Text.StringBuilder stringContenedor = new System.Text.StringBuilder();

    Cola<ArbolGeneral<DatoDistancia>> cola = new Cola<ArbolGeneral<DatoDistancia>>();

    cola.encolar(arbol);
    int nivelActual = 0;

    while (cola.cantidadElementos() != 0)
    {
        int nodosNivel = cola.cantidadElementos();

        stringContenedor.AppendLine($"Nivel : {nivelActual}");

        for (int i = 0; i < nodosNivel; i++)
        {
            ArbolGeneral<DatoDistancia> nodoActual = cola.desencolar();

            stringContenedor.Append(nodoActual.getDatoRaiz().texto + " ||| ");

            foreach (var hijo in nodoActual.getHijos())
            {
                cola.encolar(hijo);
            }
        }
        stringContenedor.AppendLine();
        nivelActual++;
    }
    return stringContenedor.ToString();
}
```

Por último tuve que desarrollar la consulta 3, para la cual opté por otra metodología respecto a las anteriores consultas. No se utiliza un método auxiliar ni mucho menos, en cambio se utiliza una adaptación de uno de los 4 recorridos que poseen los árboles: ***un recorrido por niveles***. Lo llamo adaptación ya que a lo largo de la cursada de la correspondiente materia, los recorridos mayormente tenían como objetivo el imprimir y mostrar en pantalla los datos que contenía un árbol (siendo que el orden era lo que variaba dependiendo del recorrido usado). Por ende logré realizar una adaptación para poder solucionar este problema usando de manera adicional la clase Cola provista por la cátedra.
El código sigue siendo igual en su mayoría a un recorrido por niveles que imprime en pantalla, pero ya antes de entrar en el bucle de repetición while aparecen las modificaciones: Se almacena en una variable de tipo entera el nivel actual en que se encuentra el recorrido y ya dentro del while comienzan las modificaciones más notorias. Se pasa a almacenar en una variable la cantidad de nodos presentes en ese nivel para que al paso siguiente se agregue en nuestro “String Builder” un mensaje indicando el nivel de los nodos que se vayan a mostrar en pantalla. Paso siguiente se procede a ejecutar un bucle *for* el cual se encargará de agregar en la variable *stringContenedor* únicamente los nodos correspondientes a ese nivel (aquí se ve el por que anteriormente obtuvimos la cantidad de nodos que existen en ese nivel), sumado a que se procede a agregar todos los hijos del nodo que esté procesando el bucle *foreach* a la cola, los cuales serán impresos en el siguiente nivel.
Finalizado el bucle de repetición (ya fueron agregados todos los nodos correspondientes a ese nivel), se agrega un salto de línea y se incrementa en 1 el nivel actual.
Una vez terminado este proceso por cada nivel existente, se procede a convertir nuestro *stringContenedor* en un string para así poder retornarlo.

Problemas Encontrados

Si bien la mayoría del desarrollo de proyecto lo pude enfrentar sin muchas dificultades, siendo que aquello que más tiempo tomó fue el aplicar la idea de solución en el lenguaje de programación, hubieron algunos errores en los métodos los cuales si bien no provocaban alguna excepción o que el programa se cierre de manera abrupta, no reflejaban el resultado esperado o si lo hacían, eran ineficientes en la manera en que se implementaron. A continuación se pasa a dejar constancia de estos mismos.

- **Estructura de datos desconocida:** Este es el único problema que difiere a los demás y radica en un hecho, al momento de comenzar a trabajar en el desarrollo del proyecto, la estructura de datos conocida como Árbol BK era totalmente desconocida para mi. A pesar de la breve explicación a forma de introducción que se encontraba en el archivo PDF de nuestro enunciado, aun no me quedaba claro de qué manera era que se comportaba, incluyendo la inserción y eliminación de elementos. La solución a esto fue buscar diversas fuentes en internet e inclusive algunos materiales audiovisuales, para que con una lectura nuevamente del archivo provisto por la cátedra, las dudas hayan quedado disipadas por completo y comenzar con el desarrollo.

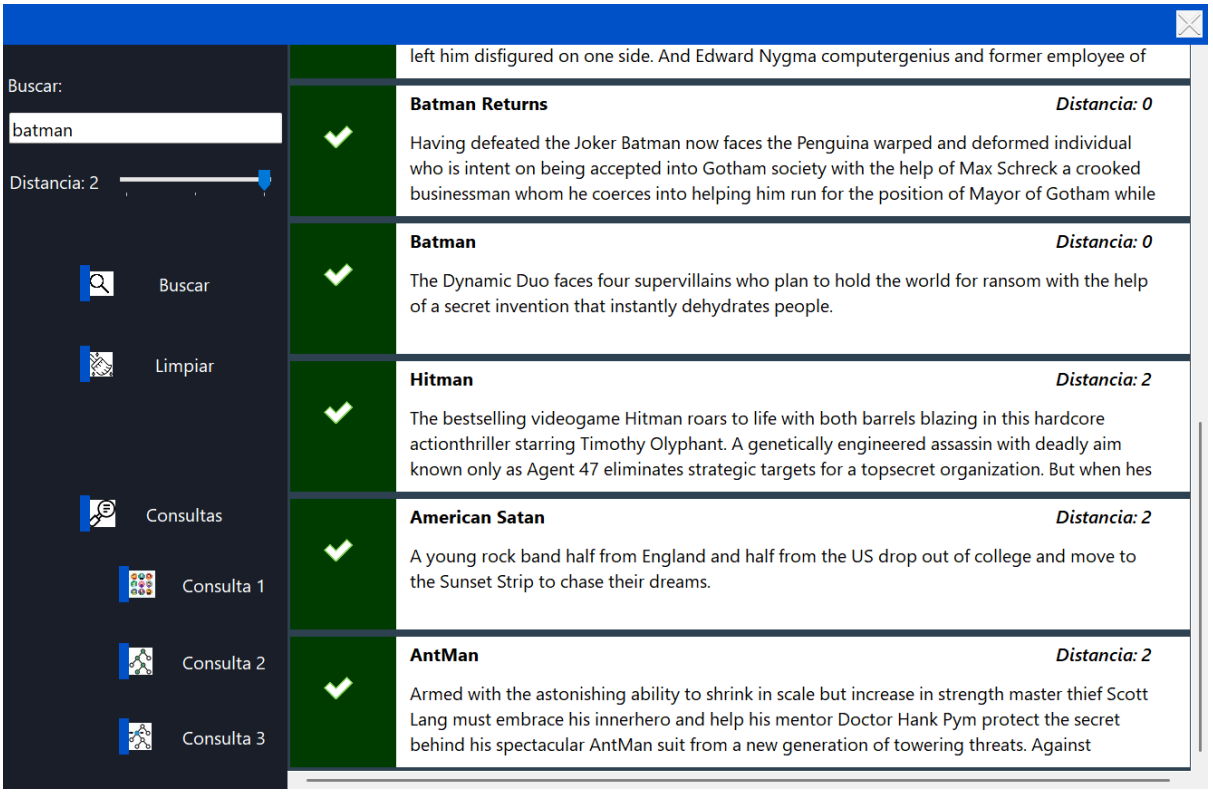
- **Método *Buscar()* :** Al momento de comenzar a desarrollar este método, mi idea principal y mi código en su fase más temprana utilizaba una búsqueda por todos los nodos existentes del árbol, sin alguna especie de filtro, por lo que estaríamos hablando de un algoritmo de orden lineal, es decir, O(n). Luego de buscar información en internet acerca del árbol BK y de cómo se basa el comportamiento a la hora de buscar algún elemento en el mismo, me encontré con que era posible una modificación en mi código que permitiera aprovechar en su totalidad la naturaleza de esta estructura. Es por eso que la versión final cuenta con una distancia mínima y distancia máxima la cual optimiza en gran medida el tiempo de búsqueda y recursos.

- **Método privado *recoleccionCaminos()* :** La idea en si era sencilla, pues el método no variaba demasiado su implementación respecto a la consulta 1, pero hubo un pequeño error el cual me llevó bastante tiempo solucionar a pesar de su tamaño. En este error, el resultado en pantalla se encargaba de mostrar la raíz de un nodo siempre repetida, de tal manera que un ejemplo claro sería:
  "Batman"→"Batman"→"Hitman"→"Hitman".
  El problema radicaba en la siguiente línea de código perteneciente al bucle foreach=
  string caminoActualizado= arbol.getDatoRaiz().texto + " ---> " + hijo.getDatoRaiz().texto;
  Aquí me encuentro añadiendo de forma repetida a la raíz seguida de su hijo, por lo que a la siguiente iteración se volverá a añadir la raíz (que anteriormente era su hijo) y así sucesivamente. La solución fue pasar al string *caminoActual* (el cual contenía todos los nodos visitados) y concatenarlo con su hijo correspondiente, evitando así volver a añadir un dato ya ingresado.

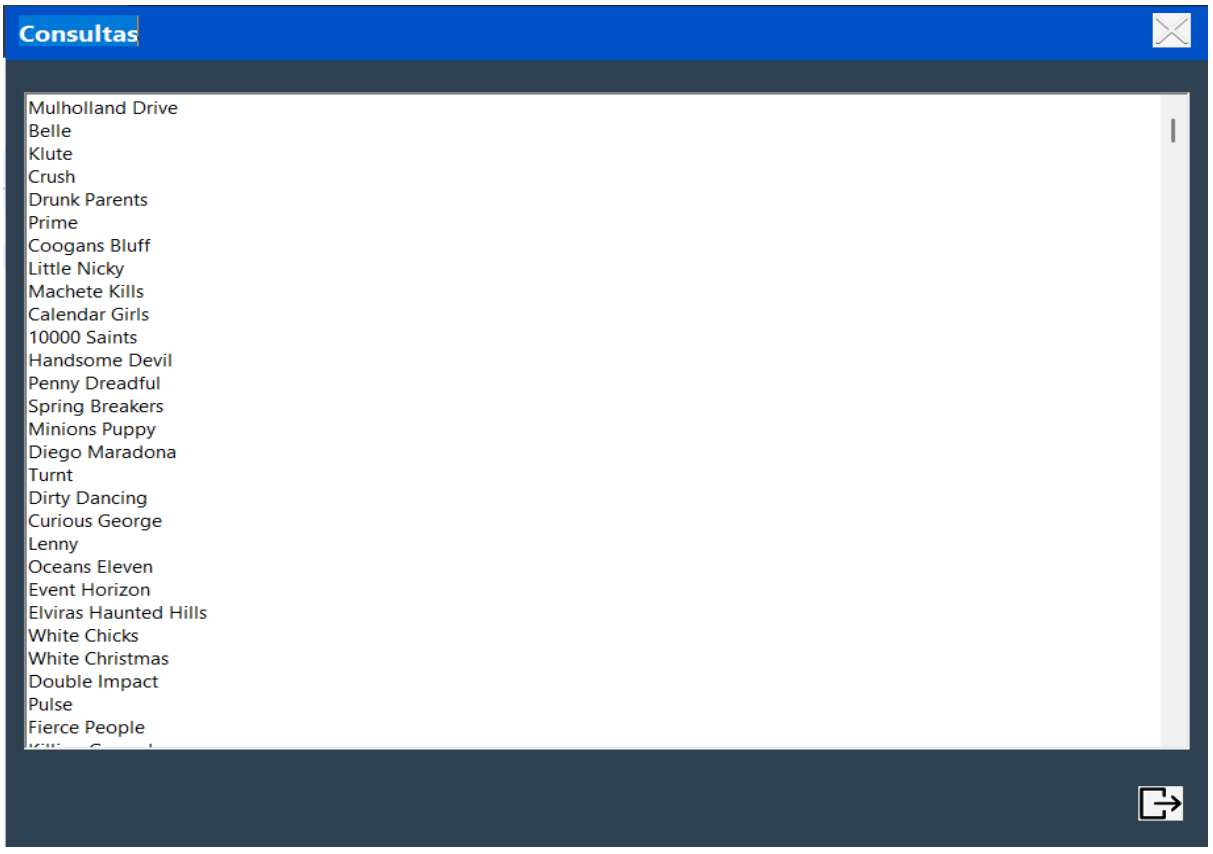
Puesta en ejecución

A continuación se adjuntan imágenes del proyecto en ejecución.

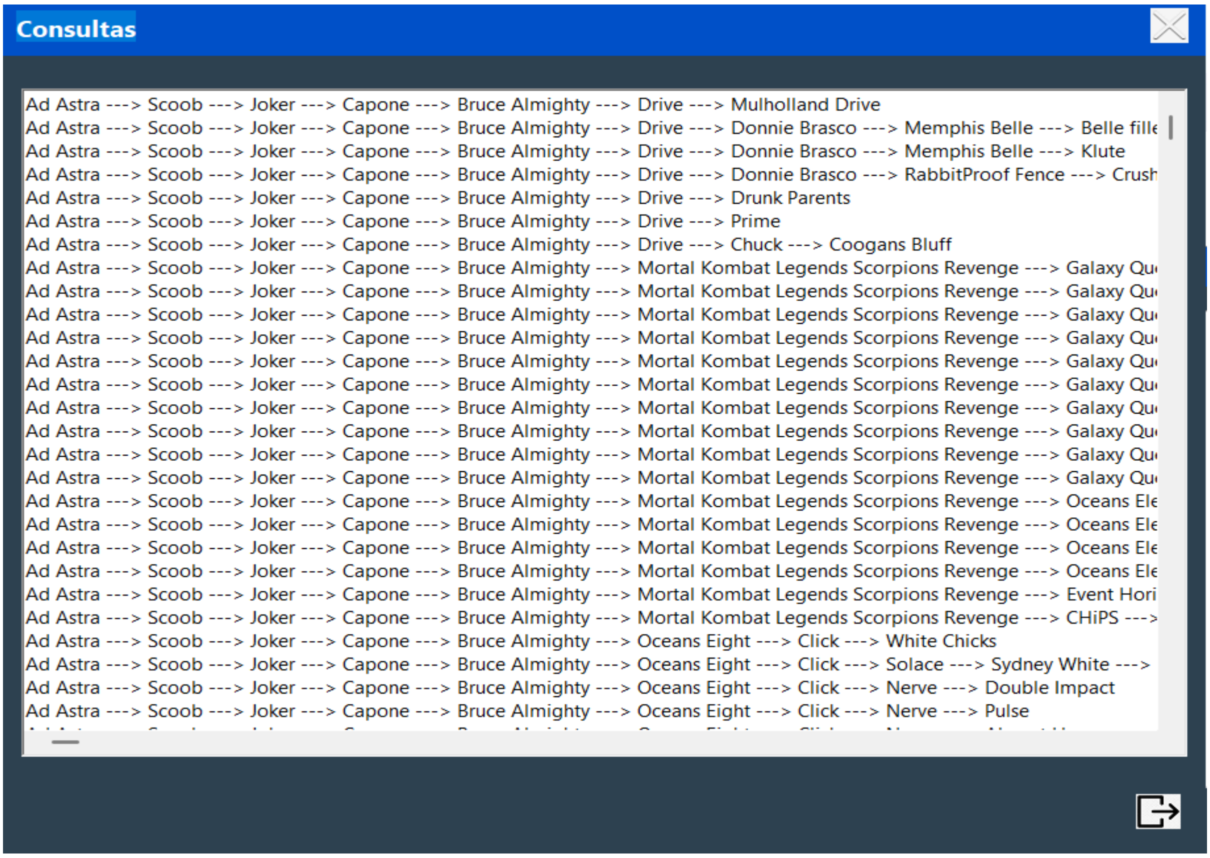
Búsqueda:



Consulta 1



Consulta 2:



Consulta 3:

Conclusión

Al momento de finalizar el trabajo puedo decir que logre consolidar y aplicar mucho de los conocimientos adquiridos a lo largo de la duración de la materia, logrando crear soluciones mucho más elegantes, comparar distintas versiones de código para un mismo problema y determinar cual es más eficiente, conocer nuevas estructuras de datos y su naturaleza y aplicar recorridos vistos en clase para búsqueda de datos. Para concluir, este proyecto significó un nuevo desafío en mi carrera como estudiante de Ingeniería en Informática y uno de mis primeros pasos en lo que respecta a estructura de datos y algoritmos.

Bibliografía

Se brindan los links a los documentos que fueron de gran utilidad para poder llegar a la resolución de este proyecto.

https://signal-to-noise.xyz/post/bk-tree/
https://medium.com/net-tips/convert-list-string-to-comma-separated-string-b2324f25362a
https://learn.microsoft.com/es-es/dotnet/standard/base-types/stringbuilder
https://learn.microsoft.com/es-es/dotnet/api/system.text.stringbuilder?view=net-8.0
https://learn.microsoft.com/es-es/dotnet/api/system.string.join?view=net-8.0