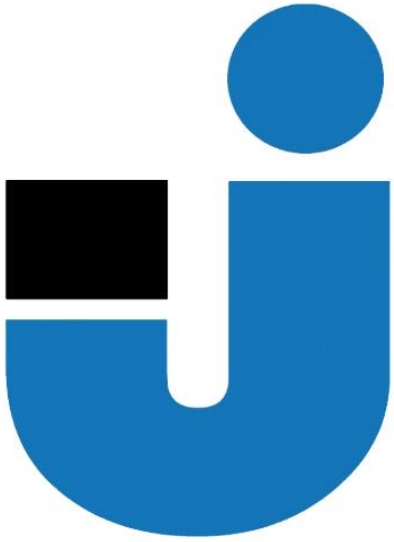

Universidad Nacional
ARTURO JAURETCHE

COMPLEJIDAD TEMPORAL, ESTRUCTURAS DE DATOS Y ALGORITMOS

INFORME TRABAJO FINAL

Docente: Ing. Mauro David Salina

Alumno: Roldan, Nicolas Emanuel

Legajo: 11205

Índice

Introducción

En el presente informe se mostrará el desarrollo del trabajo final de la materia complejidad temporal, estructura de datos y algoritmos. El lenguaje usado para este sistema será C# que es el utilizado en dicha materia. La catedra brindo la mayor parte de la estructura del sistema para sus alumnos, y el trabajo será implementar las consultas e integrar los contenidos vistos en la materia. Se procede a codificar y desarrollar un sistema para buscar las máximas ocurrencias de datos almacenados en un archivo csv. El objetivo es mostrar al usuario cuales son los elementos con más apariciones. (Imagen 1)

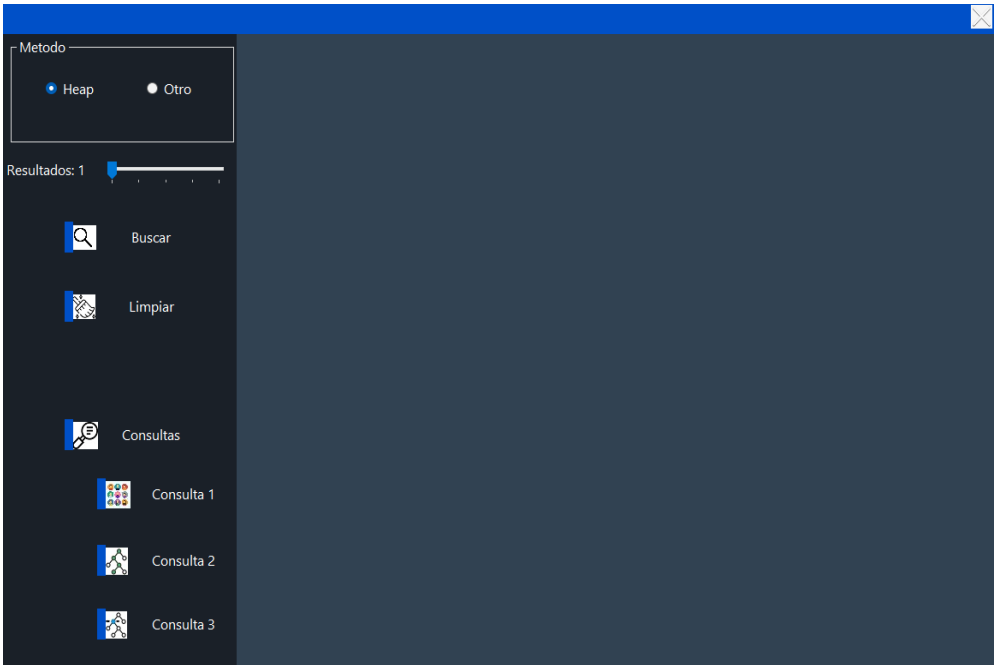

Imagen 1. Pantalla principal del sistema.

Detalle de la implementación

Para empezar el sistema pide al usuario que se seleccione un archivo scv donde se encontraran los recursos. Luego, se debe seleccionar con que estructura se desea realizar la búsqueda, en este sistema se desarrolla dos tipos de estructuras, una heap y una lista. A continuación, se presenta un diagrama UML con las clases utilizadas:

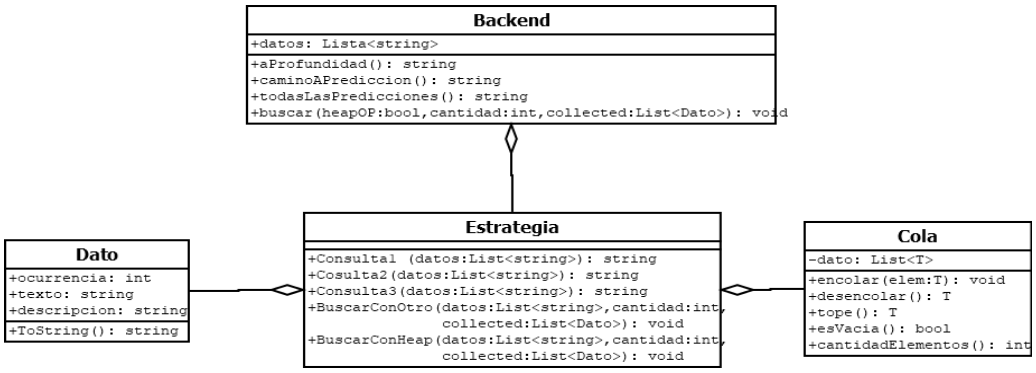

Imagen 2. Diagrama UML representando la estructura del sistema.

Para el armado del sistema se implementaron las funciones Consulta1, Consulta2, Consulta3, BuscarConHeap y BuscarConOtro. Dichas funciones están ubicadas en la clase de Estrategia, que, a su vez, incorpora las clases Dato y Cola, de las cuales la catedra de la materia proveyó para simplificar la construcción del sistema. A continuación, se detalla el objetivo e implementación de estas funciones:

- **Consulta1**: Retornara el tiempo que consume la búsqueda de 5 elementos con la estructura heap y lista. (Imagen 3).

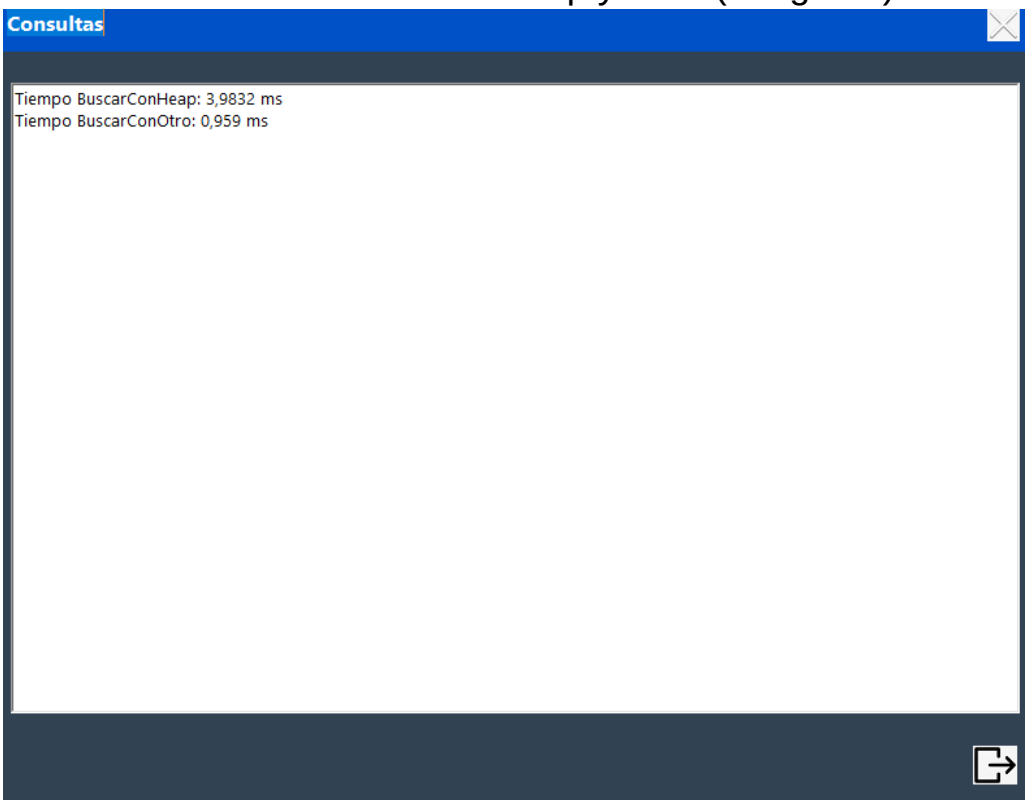

Imagen 3. Consulta1 se muestra el tiempo de ejecución de los métodos de búsqueda.

```
1 referencia
public String Consulta1(List<string> datos)
{
    //Listas de heap y otro metodo
    List<Dato> heap = new List<Dato>();
    List<Dato> otro = new List<Dato>();

    //Metodo heap, se guarda el tiempo actual luego se resta el tiempo de fin. Se hace lo mismo con buscar con otro.
    DateTime inicioHeap = DateTime.Now;
    BuscarConHeap(datos, 5, heap);
    DateTime finHeap = DateTime.Now;
    double tiempoHeap = (finHeap - inicioHeap).TotalMilliseconds;

    DateTime inicioOtro = DateTime.Now;
    BuscarConOtro(datos, 5, otro);
    DateTime finOtro = DateTime.Now;
    double tiempoOtro = (finOtro - inicioOtro).TotalMilliseconds;

    return "Tiempo BuscarConHeap: " + tiempoHeap + " ms\nTiempo BuscarConOtro: " + tiempoOtro + " ms";

}
```

Imagen 3.1. Código de la Consulta1.

- **Consulta2:** Retorna el camino de la hoja contenida más a la izquierda de la heap (Imagen 4).

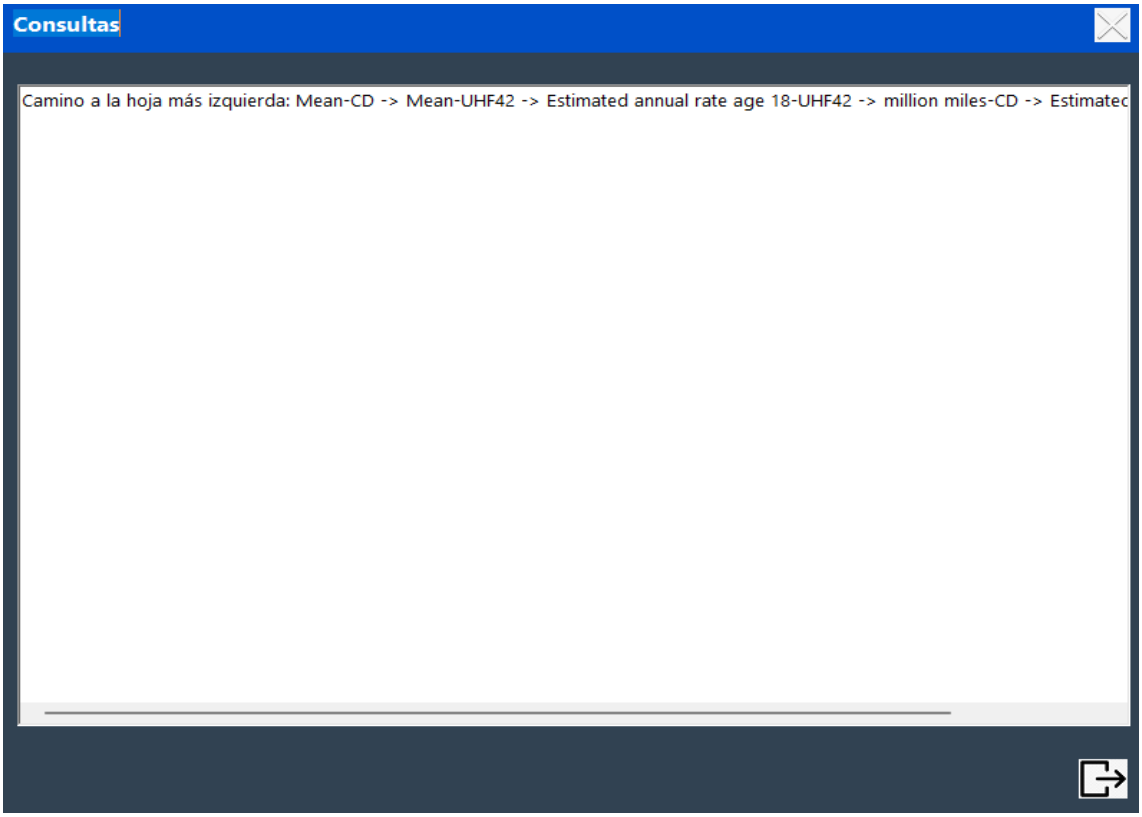

Imagen 4. Camino de la hoja más a la izquierda de la heap.

```
1 referencia
public String Consulta2(List<string> datos)
{
    //Se crea una heap y una lista de camino luego se llama al metodo buscarConHeap
    List<Dato> heap = new List<Dato>();
    BuscarConHeap(datos, datos.Count, heap);
    List<string> camino = new List<string>();

    //Si la heap no esta vacia agrega el hijo mas a la izquierda a la lista camino
    if (heap.Count > 0)
    {
        camino.Add(heap[0].texto);
        int index = 0;
        while (2 * index + 1 < heap.Count)
        {
            index = 2 * index + 1;
            camino.Add(heap[index].texto);
        }
    }

    //Imprime el camino
    string resultado = "Camino a la hoja más izquierda: ";
    for (int i = 0; i < camino.Count; i++)
    {
        if (i > 0)
        {
            resultado += " -> ";
        }
        resultado += camino[i];
    }

    return resultado;
}
```

Imagen 4.1. Código de la Consulta2.

- **Consulta 3:** Retorna el texto de cada nivel de los datos mostrados en la heap. (Imagen 5).

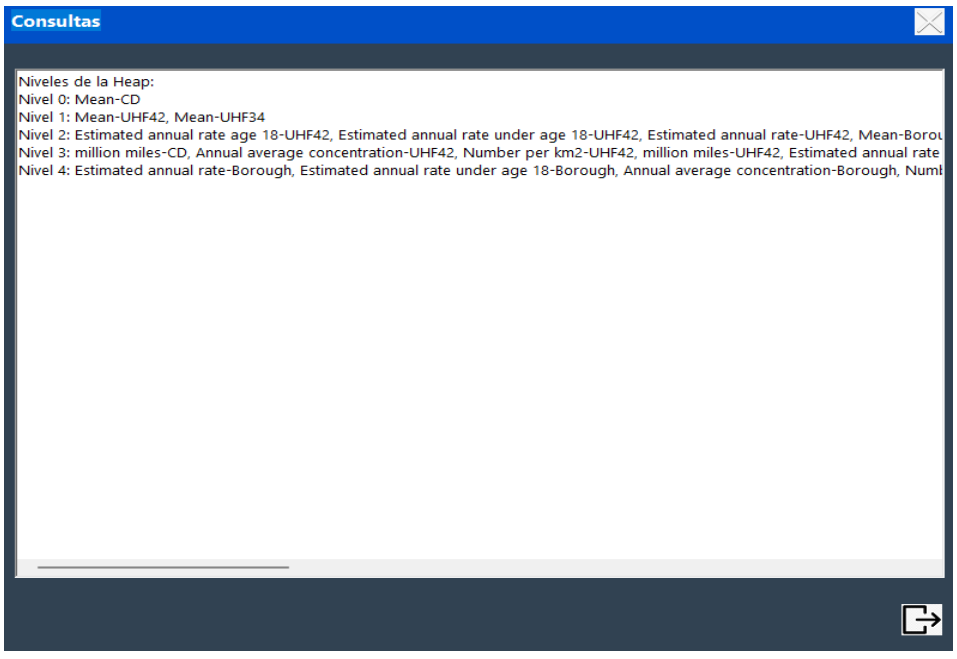

Imagen 5. Textos almacenados por niveles.

```
1 referencia
public String Consulta3(List<string> datos)
{
    //Se crea una heap, una cola y se llama al metodo buscarConHeap
    List<Dato> heap = new List<Dato>();
    BuscarConHeap(datos, datos.Count, heap);
    Cola<(Dato, int)> cola = new Cola<(Dato, int)>();
    cola.encolar((heap[0], 0));
    int nivelActual = 0;

    string resultado = "Niveles de la Heap:\nNivel 0: ";

    //Mientras la cola no este vacia va ir imprimiendo y buscando elementos por niveles en la heap.
    while (!cola.esVacia())
    {
        (Dato dato,int nivel) = cola.desencolar();

        if (nivel != nivelActual)
        {
            resultado += "\nNivel " + nivel + ": ";
            nivelActual = nivel;
        }
        else if (nivel > 0)
        {
            resultado += ", ";
        }
        resultado += dato.texto;

        int tamaño = heap.IndexOf(dato);
        int tamIzq = 2 * tamaño + 1;
        int tamDer = 2 * tamaño + 2;

        if (tamIzq < heap.Count)
        {
            cola.encolar((heap[tamIzq], nivel + 1));
        }
        if (tamDer < heap.Count)
        {
            cola.encolar((heap[tamDer], nivel + 1));
        }
    }

    return resultado;

}
```

Imagen 5.1. Código de la Consulta3.

- **BuscarConHeap:** Retornara los elementos con la mayor ocurrencia utilizando una Heap como estructura de datos. El número de elementos a retornar es indicado por el parámetro. (Imagen 6)

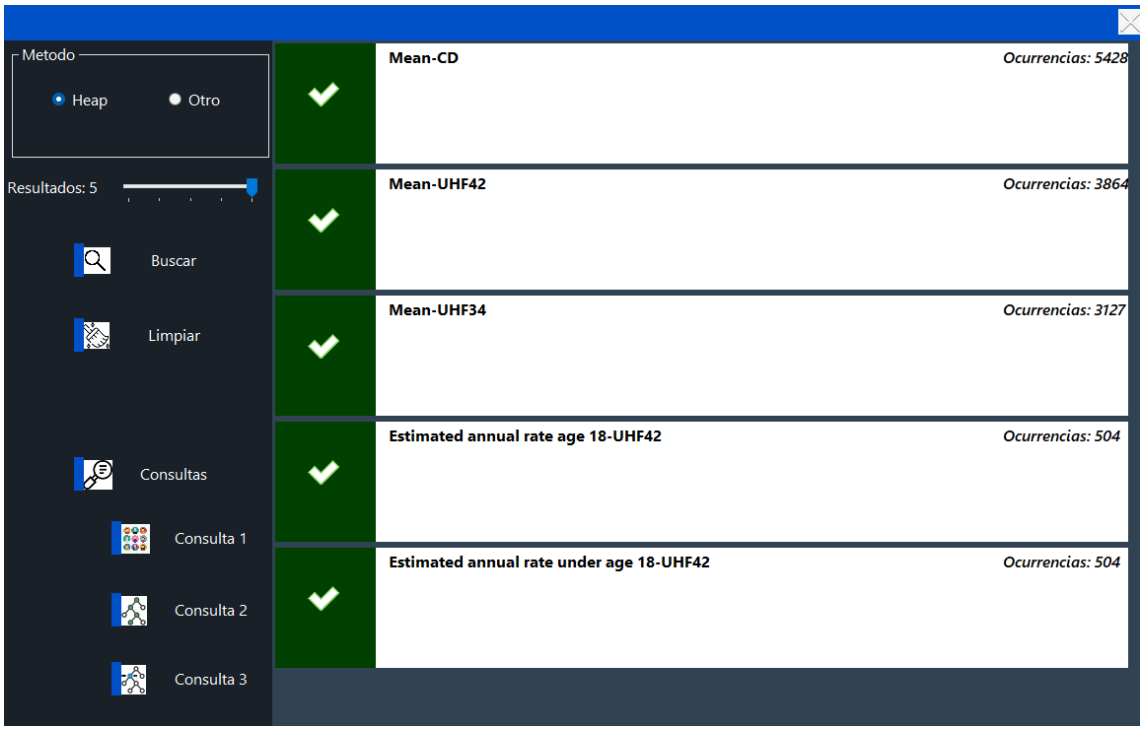

Imagen 6. Ejemplo de BuscarConHeap.

```
4 referencias
public void BuscarConHeap(List<string> datos, int cantidad, List<Dato> collected)
{
    List<Dato> heap = new List<Dato>(); //Creacion de la Heap

    foreach (var texto in datos)
    {
        bool encontrado = false;
        for (int i = 0; i < heap.Count; i++)
        {
            if (heap[i].texto == texto)
            {
                heap[i].ocurrencia++;
                encontrado = true;
                break;
            }
        }
        if (!encontrado)
        {
            heap.Add(new Dato(1, texto));
        }
    }

    for (int i = (heap.Count / 2) - 1; i >= 0; i--) //for para construir la heap
    {
        int index = i;
        while (true)
        {
            int tamaño = index;
            int hijoIzq = 2 * index + 1;
            int hijoDer = 2 * index + 2;

            if (hijoIzq < heap.Count && heap[hijoIzq].ocurrencia > heap[tamaño].ocurrencia)
            {
                tamaño = hijoIzq;
            }

            if (hijoDer < heap.Count && heap[hijoDer].ocurrencia > heap[tamaño].ocurrencia)
            {
                tamaño = hijoDer;
            }

            if (tamaño != index)
            {
                Dato temp = heap[index];
                heap[index] = heap[tamaño];
                heap[tamaño] = temp;
                index = tamaño;
            }
            else
            {
                break;
            }
        }
    }
```

Imagen 6.1. Código de BuscarConHeap (Parte1).

```
    for (int i = 0; i < cantidad && heap.Count > 0; i++) //for para extraer los elementos con mayor ocurrencia
    {
        collected.Add(heap[0]);

        // Mover el último elemento al tope y reducir el tamaño de la heap
        heap[0] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);

        // Restaurar la propiedad de la heap
        int index = 0;
        while (true)
        {
            int tamaño = index;
            int hijoIzq = 2 * index + 1;
            int hijoDer = 2 * index + 2;

            if (hijoIzq < heap.Count && heap[hijoIzq].ocurrencia > heap[tamaño].ocurrencia)
            {
                tamaño = hijoIzq;
            }

            if (hijoDer < heap.Count && heap[hijoDer].ocurrencia > heap[tamaño].ocurrencia)
            {
                tamaño = hijoDer;
            }

            if (tamaño != index)
            {
                Dato temp = heap[index];
                heap[index] = heap[tamaño];
                heap[tamaño] = temp;
                index = tamaño;
            }
            else
            {
                break;
            }
        }
    }
}
```

Imagen 6.2. Código de BuscarConHeap (Parte2).

- **BuscarConOtro:** funciona igual que el método de buscar con heap, en este caso se usará una lista como estructura de datos. (Imagen 7).

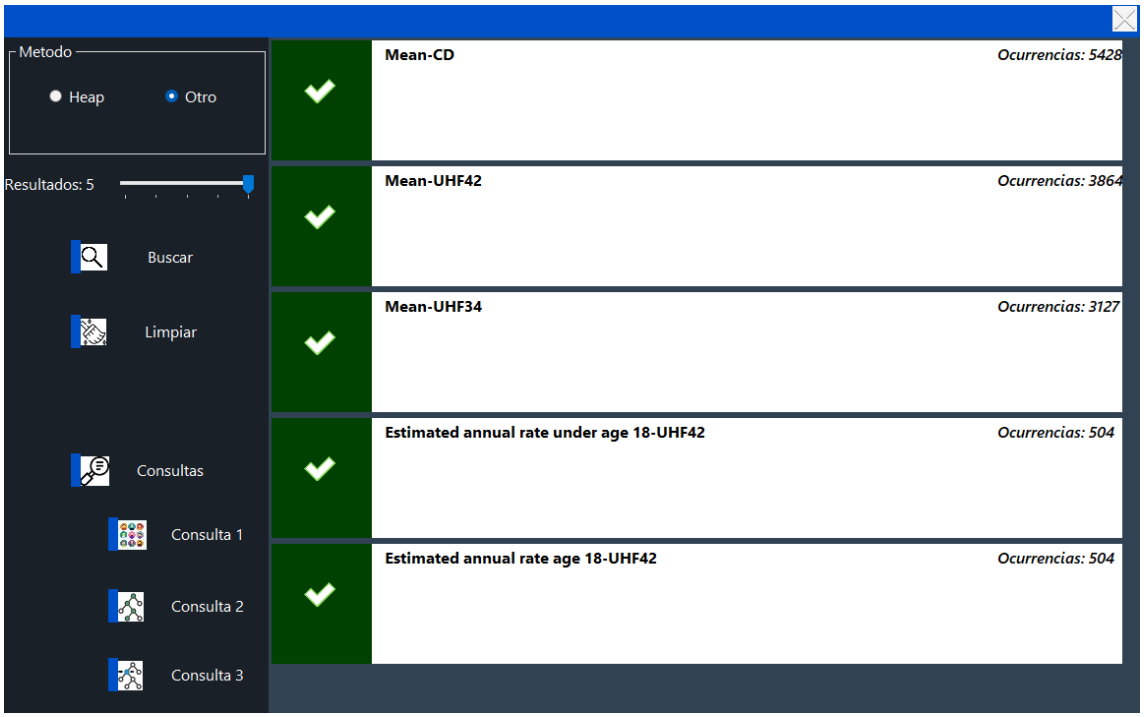

Imagen 7. Ejemplo de BuscarConOtro.

```
2 referencias
public void BuscarConOtro(List<string> datos, int cantidad, List<Dato> collected)
{
    List<Dato> listaDatos = new List<Dato>(); //Se crea la lista
    foreach (var texto in datos)
    {
        bool encontrado = false;
        for (int i = 0; i < listaDatos.Count; i++)
        {
            if (listaDatos[i].texto == texto)
            {
                listaDatos[i].ocurrencia++;
                encontrado = true;
                break;
            }
        }
        if (!encontrado)
        {
            listaDatos.Add(new Dato(1, texto));
        }
    }

    // Ordenar la lista de mayor a menor ocurrencias
    for (int i = 0; i < listaDatos.Count - 1; i++)
    {
        for (int j = 0; j < listaDatos.Count - i - 1; j++)
        {
            if (listaDatos[j].ocurrencia < listaDatos[j + 1].ocurrencia)
            {
                Dato temp = listaDatos[j];
                listaDatos[j] = listaDatos[j + 1];
                listaDatos[j + 1] = temp;
            }
        }
    }

    for (int i = 0; i < cantidad && i < listaDatos.Count; i++)
    {
        collected.Add(listaDatos[i]);
    }

}
```

Imagen 7.1. Código de BuscarConOtro.

Problemas encontrados y mejoras

Tras probar varias veces el sistema fueron encontrados problemas que hacen referencia a la comprensión de la consigna e implementación en código. El armado de la heap me dificulto mucho ya que es la primera vez que trabajo con este tipo de estructura. Mantener el orden de la heap, luego ordenar las ocurrencias fueron

los primeros problemas encontrados, pero lo logre resolver a prueba y error, buscando información adicional y con ayuda de compañeros. Las consultas paso de manera similar.

Finalmente, en torno a la eficiencia para mejorar el sistema se podría usar una estructura mucho más eficaz como los árboles binarios de búsqueda u otra estructura mas avanzada. En torno al usuario incluiría una ayuda para el uso del sistema, implementaría un historial de búsqueda.

Conclusión

En conclusión, el trabajo me pareció muy bueno y enriquecedor aportando más conceptos similares a los vistos en la materia. Es la primera vez que me toca hacer un trabajo con muchas clases y muchas líneas de código, al principio me costó mucho entender las clases y como era el funcionamiento, pero practicando logre entender la mayoría del programa. Gracias a este trabajo final pude reforzar conceptos principales de la materia. Como Heap, arboles y listas. Fue desafiante ya que me costó mucho comprender el enunciado y arrancar a trabajar, ya que en la catedra no se muestra como se arma una Heap en código. Mas allá de todo esto me quedo con que aprendí a trabajar con esta estructura de datos, escribir código eficaz y mantenible, documentar y armar una planificación previa antes de armar el sistema.