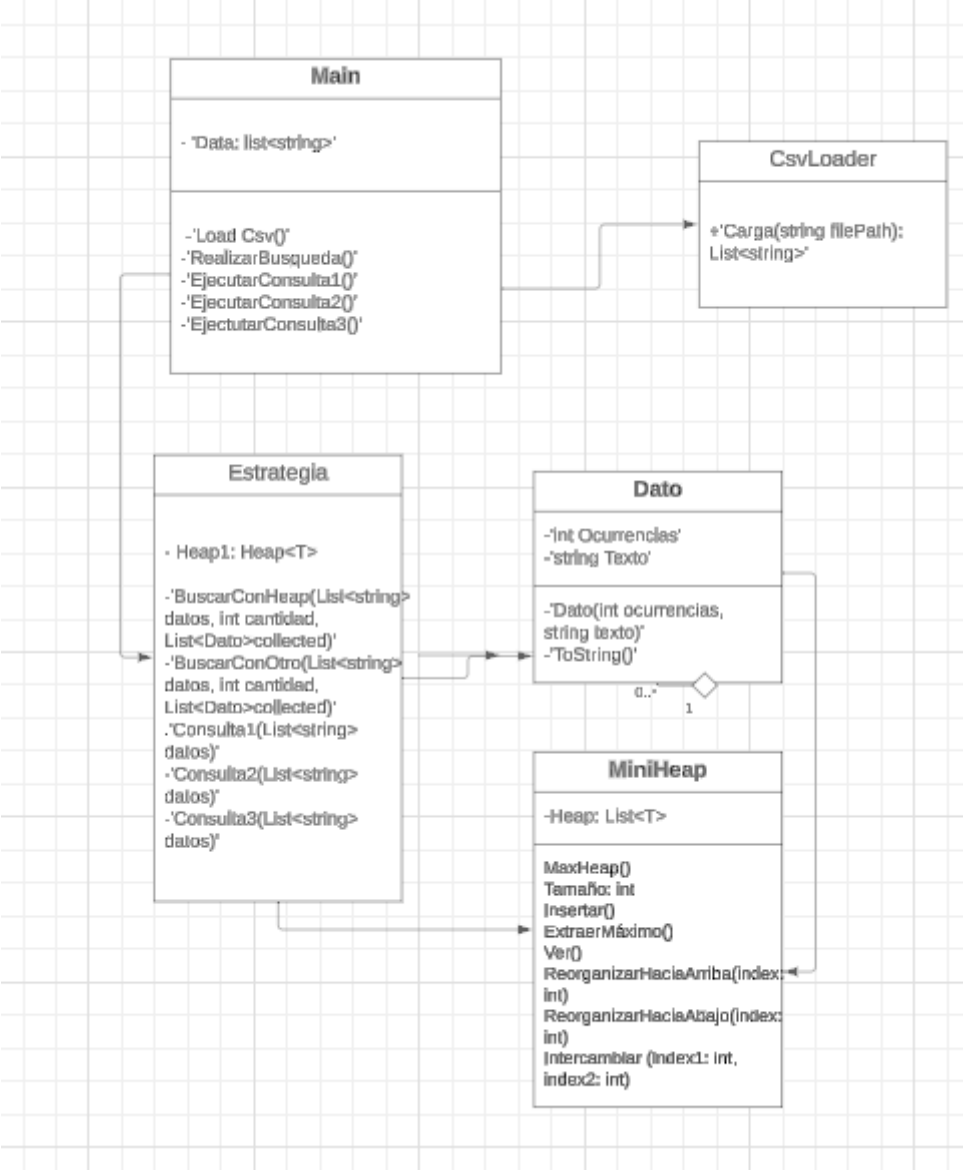
Complejidad Temporal, Estructuras de Datos y Algoritmos

Trabajo Final

Castro Juan Pablo, comisión 2.

DNI: 44256239

-UML



UML del proyecto, con las clases, estructura, relaciones, atributos y métodos.

-Problemas a la hora de la implementación:

El desarrollo del proyecto dio comienzo con un inconveniente en el momento de lectura de archivo csv que fue solucionado cambiando un carácter de la clase “Forms 2”.

Se presentaron conflictos de pasaje de valores entre dato, estrategia y Heap, ya que se debían utilizar las 3 clases para resolver cuestiones como por ejemplo, la implementación de la clase BuscarConHeap, qué debía sacar un archivo de tipo Dato, cargado en la lista datos, contar sus ocurrencias y luego agregarlo a la Heap.

Se logró mostrar el resultado con más ocurrencias pero al querer ver los 5 resultados, se muestra el mismo 5 veces, la resolución fue una variable que se declaraba erróneamente dentro del bucle cada vez que se añadía un dato a la lista Collected.

Luego de resolver la impresión de resultados, muestra los 5 con más ocurrencias pero los datos de Collected de manera desordenada, la solución era añadir una nueva lista temporal que los ordene para su correcta impresión.

Problema surgido con Consulta 2, se debía utilizar la Heap creada de otro método, para el cual se nombró una heap global de la clase estrategia para poder ser utilizada.

-Observaciones e implementos:

-Para comparar los tiempos de ejecución de BuscarConHeap y BuscarConOtro deberíamos de usar una clase utilitaria externa que calcule en tiempo real con el reloj del sistema.

Comenzamos la creación del Heap, con estructura de MiniHeap y con las funciones básicas: Agregar, eliminar objeto, ver cantidad, eliminar raíz y ver raíz, por otra parte las funciones privadas; el filtrado hacia arriba o hacia a la hora de alterar la Heap de alguna manera y el Swap que se comunica con estas mismas.

```
public class Heap<T>
{
    IList<T> Arbol = new List<T>();
    Comparer<T> comparar;

    public Heap(Comparer<T> comparar)
    {
        this.comparar=comparar;
    }

    public Heap(Comparison<T> comparacion)
    {
        this.comparar=Comparer<T>.Create(comparacion);
    }

    public Heap()
    {
        this.comparar=Comparer<T>.Default;
    }

    public int Cantidad()
    {
        int cantidad=Arbol.Count -1;
        return cantidad;
    }
    public void Agregar(T value)
    {
        Arbol.Add(value);
        FiltradoHaciaArriba(Arbol.Count -1);
    }

    public T Ver(int posicion)
    {
        return Arbol[posicion];
    }

    public T VerRaiz()
    {
        return Arbol[0];
    }

    public T BorrarRaiz()
    {
        var resultado = VerRaiz();
        Swap(0, Arbol.Count -1);
        Arbol.RemoveAt(Arbol.Count - 1);
        FiltradoHaciaAbajo(0);
        return resultado;
    }
    public void Borrar(T value)
    {
        if (Arbol[Arbol.Count-1].Equals(value))
        {
            Arbol.RemoveAt(Arbol.Count -1);
            return;
        }
        var Indice = Arbol.IndexOf(value);
        Swap(Indice, Arbol.Count-1);
        Arbol.RemoveAt(Arbol.Count -1);

        int segundoIndice = (Indice - 1) /2;
        if (comparar.Compare(Arbol[Indice], Arbol[segundoIndice])>0)
        {
            FiltradoHaciaArriba(Indice);
        }
        else
        {
            FiltradoHaciaAbajo(Indice);
        }
    }
```

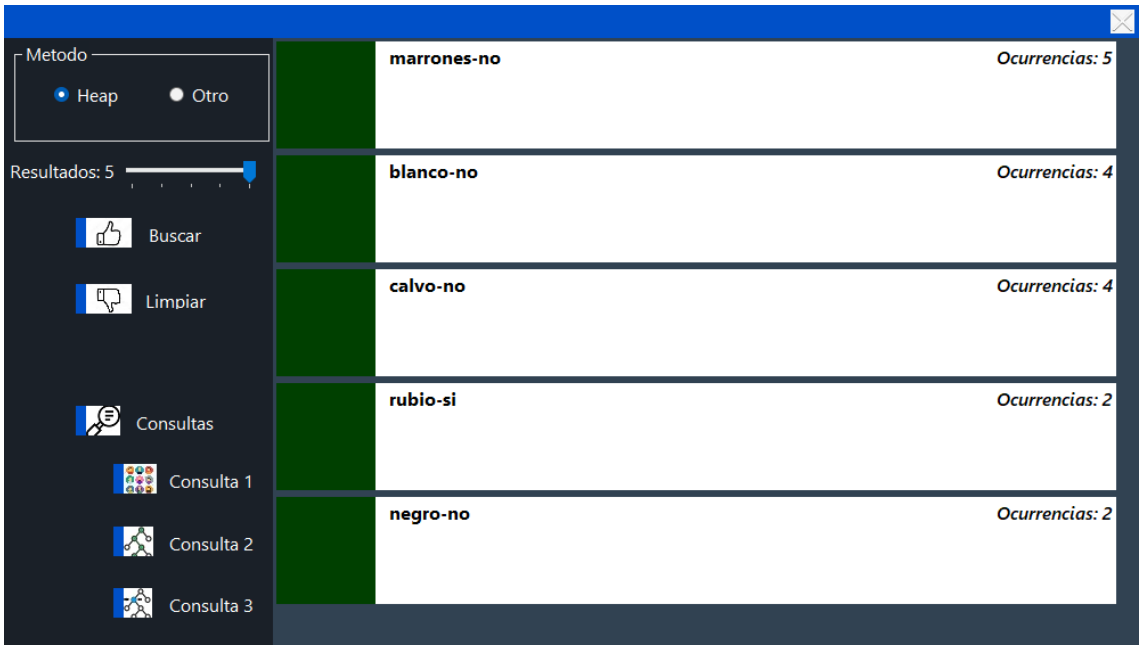
```
        private void FiltradoHaciaArriba(int indice)
        {
            while (indice > 0)
            {
                int indiceDos = (indice - 1) / 2;
                if (comparar.Compare(Arbol[indice], Arbol[indiceDos])>0)
                {
                    Swap (indice, indiceDos);
                    indice = indiceDos;
                }
                else
                {
                    break;
                }
            }
        }
        private void FiltradoHaciaAbajo(int indice)
        {
            while (true)
            {
                var hijoIzqIndice = indice * 2 + 1;
                if (hijoIzqIndice >= Arbol.Count)
                {
                    return;
                }
                var hijoActualIndice = hijoIzqIndice;
                var hijoDerechoIndice = indice * 2 + 2;
                if (hijoDerechoIndice < Arbol.Count)
                {
                    if (comparar.Compare(Arbol[hijoDerechoIndice], Arbol[hijoIzqIndice])>0)
                    {
                        hijoActualIndice = hijoDerechoIndice;
                    }
                }
                if (comparar.Compare(Arbol[hijoActualIndice], Arbol[indice]) > 0)
                    {
                        Swap(hijoActualIndice, indice);
                        indice = hijoActualIndice;
                    }
                else
                {
                    break;
                }
            }
        }
        private void Swap(int i, int j)
        {
            var t = Arbol[i];
            Arbol[i] = Arbol[j];
            Arbol[j] = t;
        }
    }
}
```

Implemento de la Heap.

Una vez desarrollada la Heap, podemos utilizarla para la búsqueda, se utiliza la clase de diccionario, para almacenar un string junto con un int, se cuentan las ocurrencias, se crea la heap y el objeto de tipo diccionario se convierte en tipo Dato (que también está conformado por un string y un int) que luego se agrega a la Heap. Posteriormente se agrega a una lista temporal, para ser agregado a la lista collected, afín de una impresión ordenada.

```
public void BuscarConHeap(List<string> datos, int cantidad, List<Dato> collected)
 {
 Dictionary<string, int> ocurrencias = new Dictionary<string, int>();
 foreach (var dato in datos)
 {
     if (ocurrencias.ContainsKey(dato))
     {
         ocurrencias[dato]++;
     }
     else
     {
         ocurrencias[dato] = 1;
     }
 }
     var comparador = Comparer<Dato>.Create((x, y) => y.ocurrencia.CompareTo(x.ocurrencia));
     Heap<Dato> heap1= new Heap<Dato>(comparador);
     foreach (KeyValuePair<string,int> par in ocurrencias)
     {
     string Texto = par.Key;
     int Ocurrencias = par.Value;
     Dato dato1 = new Dato(Ocurrencias, Texto);
     heap1.Agregar(dato1);
     }
     this.Heap1=heap1;
     int hastaCinco = 5;
     int cantidadDatos = heap1.Cantidad();
     List<Dato> tempCollected = new List<Dato>();
     while (hastaCinco >0)
     {
         Dato maxOcurrencias = heap1.Ver(cantidadDatos);
         tempCollected.Add(maxOcurrencias);
         cantidadDatos--;
         hastaCinco--;
     }

     tempCollected.Sort((x, y) => y.ocurrencia.CompareTo(x.ocurrencia));
     int puntero = 0;
     while (cantidad >0)
     {
         collected.Add(tempCollected[puntero]);
         puntero++;
         cantidad--;
     }
     return;
 }


 }
```

Implemento de BuscarConHeap.

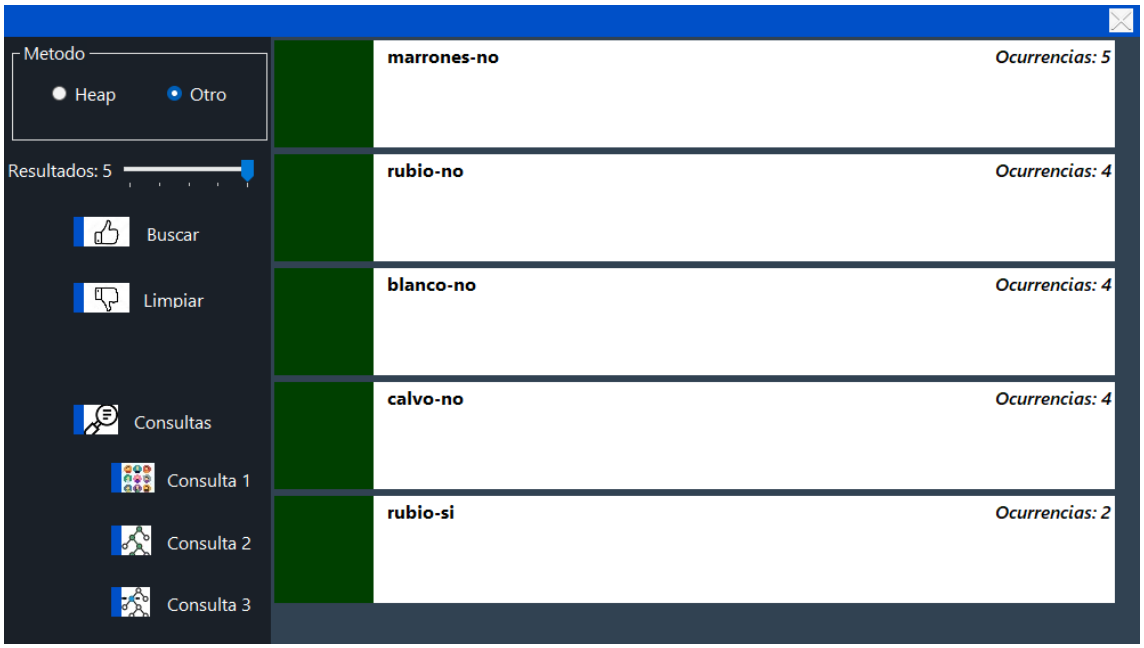


Resultados de BuscarConHeap, utilizando el archivo llamado “preguntas.csv”

El proyecto continúa con la creación de “BuscarConOtro”, para contar las ocurrencias seguimos con el tipo diccionario, esta vez utilizando una lista de datos para almacenar las ocurrencias y palabras provenientes de la lista datos. Luego almacenar en otra lista de manera ordenada según las ocurrencias con el método “.OrderByDescending” y devolver esa lista a collected para su impresión.

```
public void BuscarConOtro(List<string> datos, int cantidad, List<Dato> collected)
{

Dictionary<string, int> ocurrencias = new Dictionary<string, int>();

foreach (var dato in datos)
{
    if (ocurrencias.ContainsKey(dato))
    {
        ocurrencias[dato]++;
    }
    else
    {
        ocurrencias[dato] = 1;
    }
}

List<Dato> listaDatos = ocurrencias.Select(kvp => new Dato(kvp.Value, kvp.Key)).ToList();

List<Dato> datosOrdenados = listaDatos.OrderByDescending(d => d.ocurrencia).Take(cantidad).ToList();

collected.AddRange(datosOrdenados);

}
```

Implementación de “BuscarConOtro”



Resultados de BuscarConHeap, utilizando el archivo llamado “preguntas.csv”

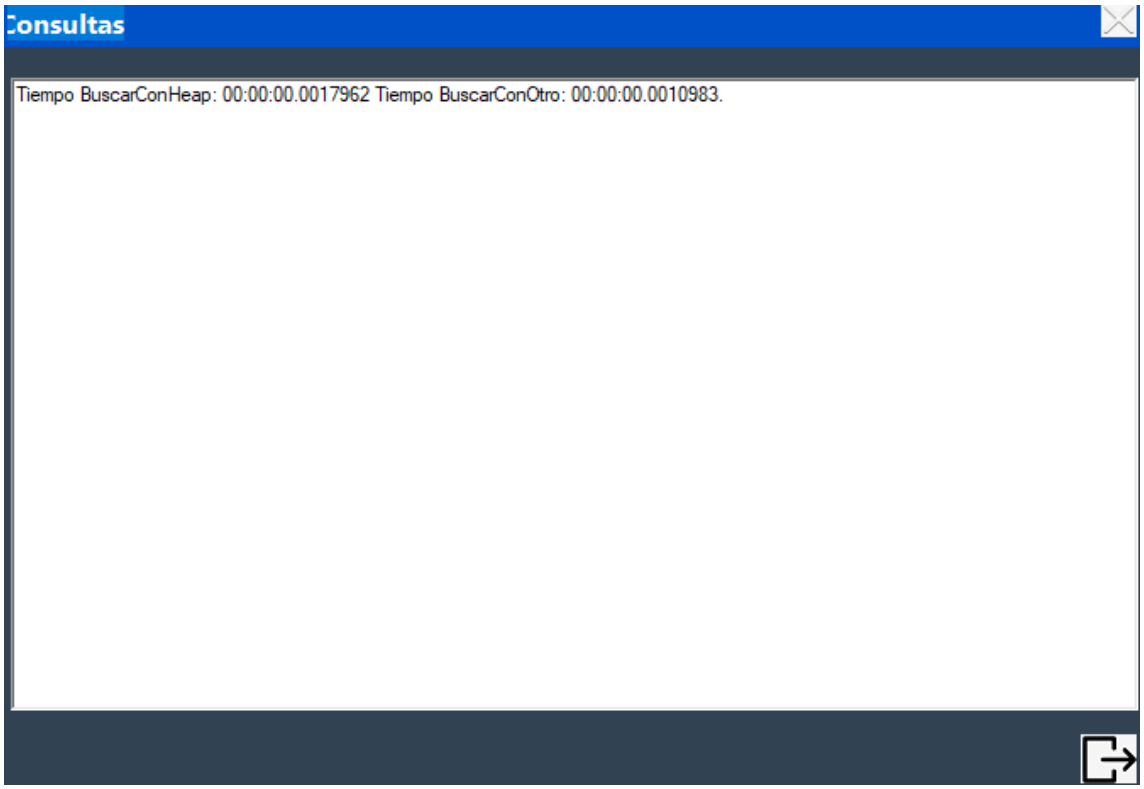
Una vez creados los dos métodos de búsqueda proseguiremos con la creación de Consulta1 y Consulta2.

Para Consulta1 implementamos la clase Stopwatch, invocamos las funciones y contamos el tiempo de su ejecución.

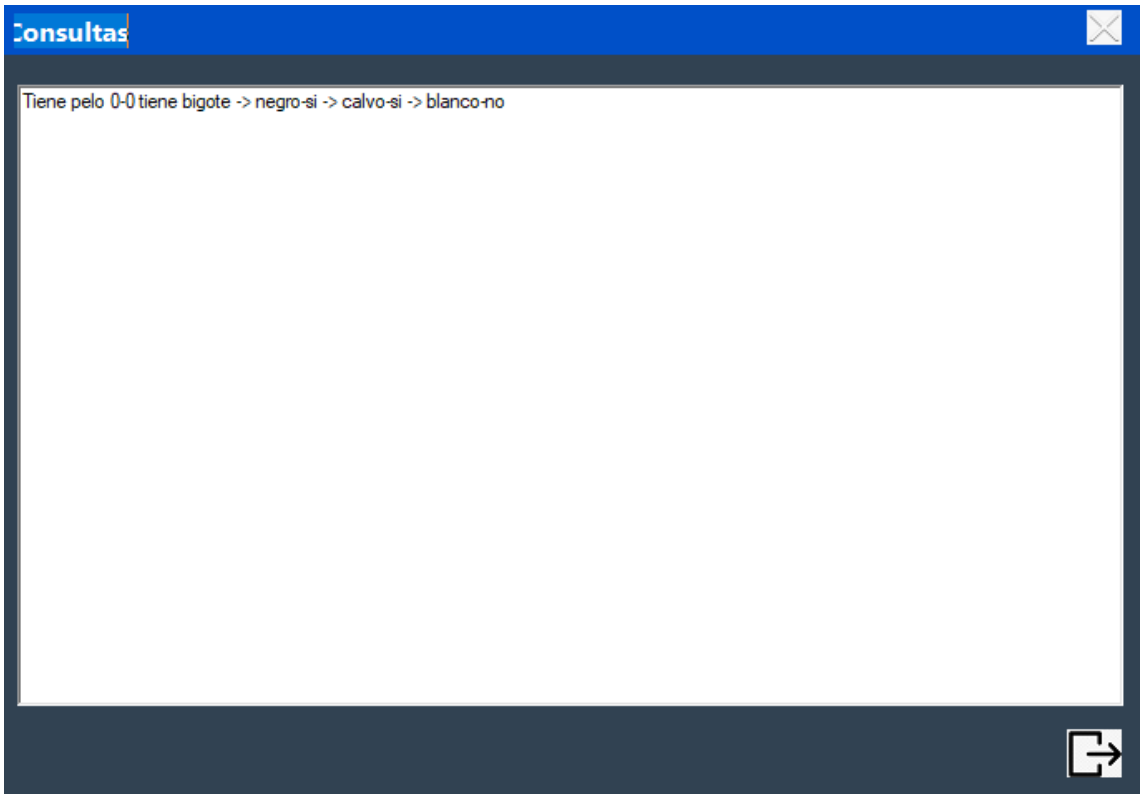
En Consulta2 necesitamos de acceder al Heap creado por la fusión BuscarConHeap, para luego, junto con un índice, ir visitando el hijo izquierdo de los datos.

```
public class Estrategia
{
    private Heap<Dato> Heap1;

    public String Consulta1(List<string> datos)
    {
    List<Dato> listaD = new List<Dato>();
    Stopwatch stopwatchHeap = new Stopwatch();
    Stopwatch stopwatchOtro = new Stopwatch();
    stopwatchHeap.Start();
    BuscarConHeap(datos, 5,listaD);
    stopwatchHeap.Stop();
    stopwatchOtro.Start();
    BuscarConOtro(datos, 5, listaD);
    stopwatchOtro.Stop();
    string resultado = "Tiempo BuscarConHeap: " + stopwatchHeap.Elapsed+ " Tiempo BuscarConOtro: "+ stopwatchOtro.Elapsed + ".";
    return resultado;
    }


    public String Consulta2(List<string> datos)
    {
    List<Dato> listaD2 = new List<Dato>();
    BuscarConHeap(datos, 5, listaD2);
    int indice = 0;
    string camino = Heap1.VerRaiz().texto;
while (indice * 2 + 1 < Heap1.Cantidad())
{
    indice = indice * 2 + 1;
    camino += " -> " + Heap1.Ver(indice).texto;
}

     return camino;

    }
```

Implementación de Consulta 1 y Consulta 2.



Resultado de la Consulta 1 utilizando el archivo llamado “preguntas.csv”.

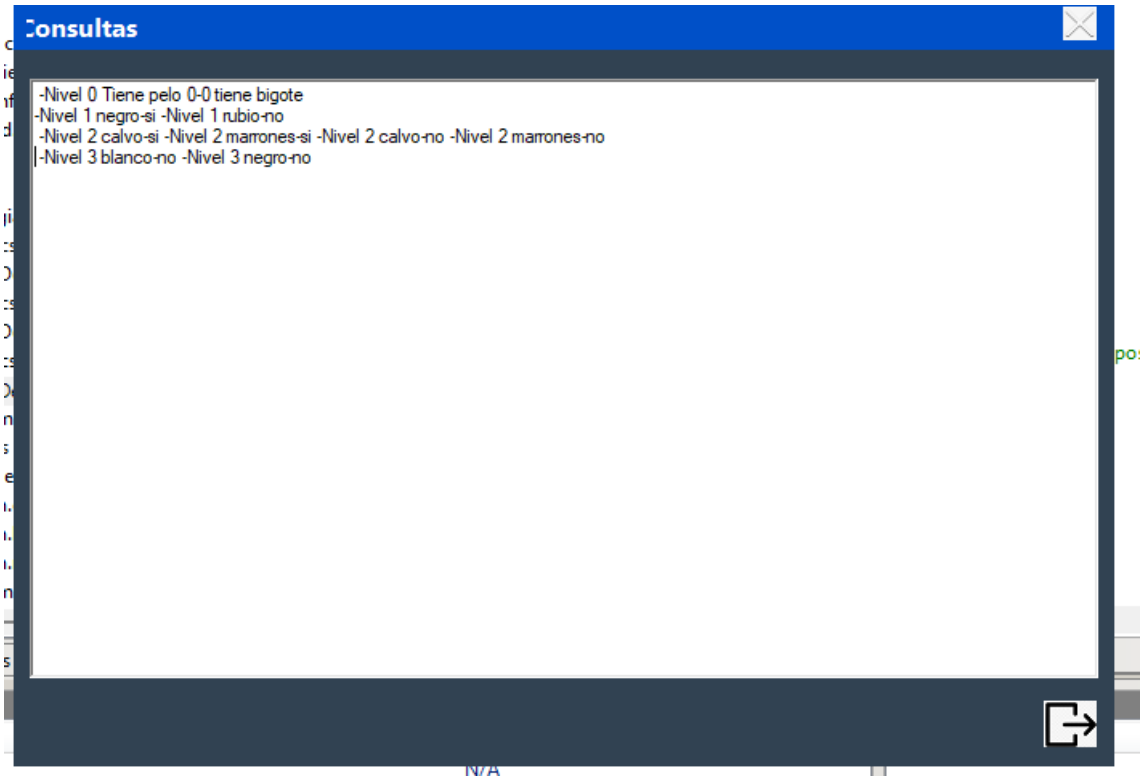


Resultado de la Consulta 2 utilizando el archivo llamado “preguntas.csv”.

Por último, para la implementación de Consulta3, se ha requerido usar el tipo Queue (Fila). La fila “Indices” se usa como soporte para enfilar los indíces y realizar una operación con los mismos para acceder por posición a la Heap.

```
public String Consulta3(List<string> datos)
{
List<Dato> listaD3 = new List<Dato>();
BuscarConHeap(datos, 5, listaD3);

int nivel = 0;
int itemsNivelActual = 1;
int itemsNivelSiguiente = 0;
int Indice = 0;
string result= "";
Queue<int> Indices = new Queue<int>();
Indices.Enqueue(Indice);
while (Indices.Count > 0)
{
    Indice = Indices.Dequeue();
    Dato datoActual= Heap1.Ver(Indice);
    result += " -Nivel " + nivel + " " +atoActual.texto;
    if (Indice * 2 + 1< Heap1.Cantidad())
    {
        Indices.Enqueue(Indice * 2 + 1);
        itemsNivelSiguiente++;
    }
    if (Indice * 2 + 2< Heap1.Cantidad())
    {
        Indices.Enqueue(Indice * 2 + 2);
        itemsNivelSiguiente++;
    }
    itemsNivelActual--;
    if (itemsNivelActual == 0)
    {
        nivel++;
        itemsNivelActual= itemsNivelSiguiente;
        itemsNivelSiguiente=0;
    }
}
return result.TrimEnd();
}
```

Implementación de la Consulta3.



Resultado de la consulta 3 utilizando el archivo llamado “preguntas.csv”.

Para mejorar el proyecto, se podrían implementar manejo de errores y validaciones, así como agregar validaciones para manejar listas vacías o nulas o Incluir manejo de excepciones para situaciones inesperadas. O Cambiar el tipo de datos de List<string> a List<T> donde T es genérico, lo que permite usar el código con cualquier tipo de dato.

Durante la realización de este proyecto, tuve la oportunidad de aplicar y consolidar varios conceptos clave en programación y estructuras de datos, lo cual ha sido una experiencia enriquecedora tanto en términos académicos como prácticos.

La implementación de una Heap y el manejo de operaciones básicas como el Swap o el filtrado hacia arriba ha profundizado mi comprensión de las estructuras de datos avanzadas y su importancia en la optimización de algoritmos.

Al momento de medir el tiempo de ejecución de diferentes algoritmos me ha enseñado a ser consciente de la eficiencia del código. La optimización del rendimiento no solo mejora la velocidad de ejecución a escala global, sino que también mejora la escalabilidad de las aplicaciones.

Este proyecto ha sido una excelente oportunidad para integrar conocimientos teóricos con aplicaciones prácticas. La experiencia adquirida me ha permitido ver la conexión entre los conceptos estudiados en la clase y su implementación real.

Además, trabajar en este proyecto me ha enseñado a abordar problemas de manera estructurada y a pensar en múltiples soluciones antes de seleccionar la más adecuada. La capacidad de evaluar y comparar diferentes enfoques es una habilidad valiosa en ingeniería, donde a menudo hay múltiples caminos para resolver un problema.