

**Informe Final:**

**Desarrollo de Sistema de Gestión de Datos**

Angel Rubén Ruidiaz | CTEDA | 25 junio 2024

Informe Final: Desarrollo de Sistema de Gestión de Datos

**Nombre del Autor:** Angel Rubén Ruidiaz **Materia:** Complejidad Temporal y Estructura de Datos **Fecha:** 25 junio 2024

**1. Introducción**

La empresa informática está desarrollando un sistema de búsqueda de máximas ocurrencias con el objetivo de analizar conjuntos de datos almacenados en archivos CSV. Este sistema permitirá a los usuarios identificar los elementos que tienen la mayor cantidad de apariciones en esos conjuntos de datos.

El objetivo principal del sistema es proporcionar a los usuarios una herramienta eficiente para analizar grandes volúmenes de datos y determinar cuáles son los elementos más frecuentes en esos conjuntos de datos. Esto puede ser útil en diversas aplicaciones, como análisis de mercado, procesamiento de datos de encuestas, detección de anomalías, entre otros.

Funcionalidades del Sistema

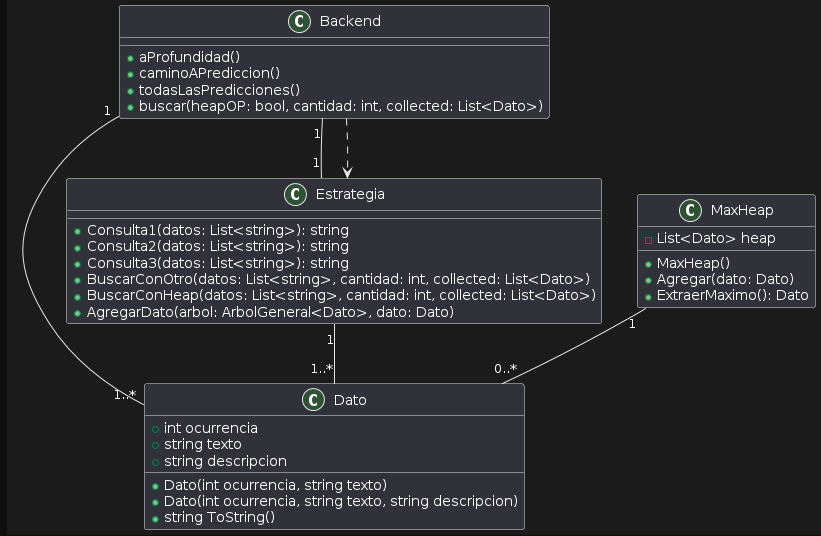
* **Carga de datos desde archivos CSV.**
* **Búsqueda de máximas ocurrencias.**
* **Consulta de resultados.**
* **Visualización de datos.**

A lo largo del desarrollo del sistema de búsqueda de máximas ocurrencias, se enfrentaron diversos desafíos que pusieron a prueba la capacidad de manipulación y análisis de datos del equipo de desarrollo. Estos desafíos abarcaron desde la optimización de algoritmos para la eficiencia en la búsqueda y consulta de datos hasta la gestión adecuada de la estructura de datos tipo heap.

Uno de los desafíos más importantes fue la implementación eficiente de la estructura de datos para asegurar un rendimiento óptimo durante las operaciones de búsqueda y consulta. Asimismo, se encontraron retos relacionados con la gestión de grandes volúmenes de datos y la necesidad de presentar resultados claros y comprensibles para el usuario final.

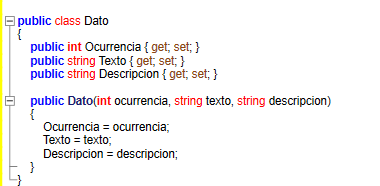
Para superar estos desafíos, se aplicaron diversas estrategias y técnicas de optimización en el desarrollo del sistema. Se realizaron ajustes en los algoritmos de búsqueda y consulta, se optimizó el uso de recursos de hardware y se implementaron soluciones innovadoras para mejorar la experiencia del usuario y la eficiencia del sistema en general.

**2. Detalles de Implementación**

****

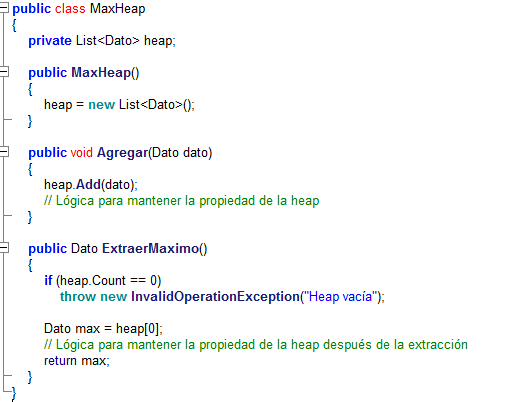
**Clase Dato**

La clase Dato representa un dato individual con tres atributos: Ocurrencia, Texto, y Descripcion.



**Clase MaxHeap**

La clase MaxHeap implementa una heap máxima utilizando una lista de objetos Dato. Proporciona métodos para agregar elementos a la heap y extraer el elemento máximo.



**Clase Estrategia**

La clase Estrategia contiene los métodos principales que interactúan con la heap y los datos.

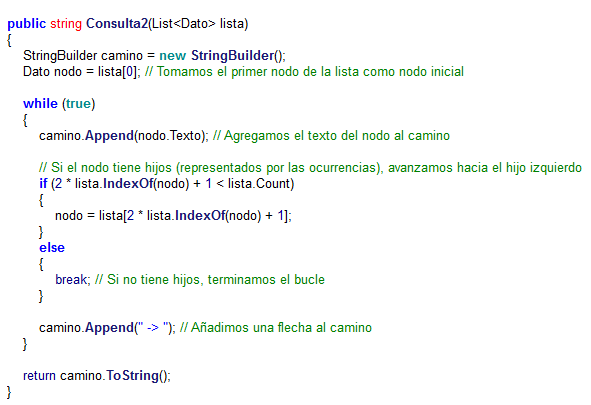
**Método BuscarConHeap**

Este método cuenta las ocurrencias de cada cadena en la lista de datos, las agrega a una heap máxima y devuelve una cadena con los resultados.



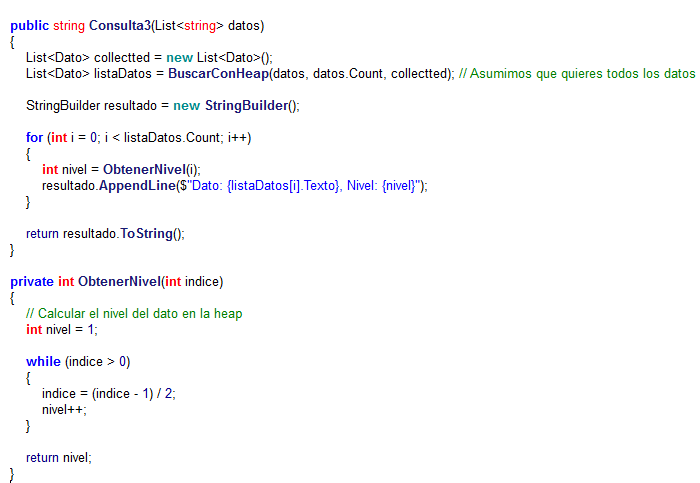
**Método Consulta2**

Este método recibe una lista de Dato y retorna una cadena que describe el camino a la hoja más izquierda en la estructura

. 

**Método Consulta3**

Este método construye un texto que contiene los datos de la heap y los niveles en los que se encuentran ubicados.



**3. Condiciones de Ejecución**

**Requisitos:**

* .NET Framework o .NET Core.
* Un entorno de desarrollo como SharpDevelop.

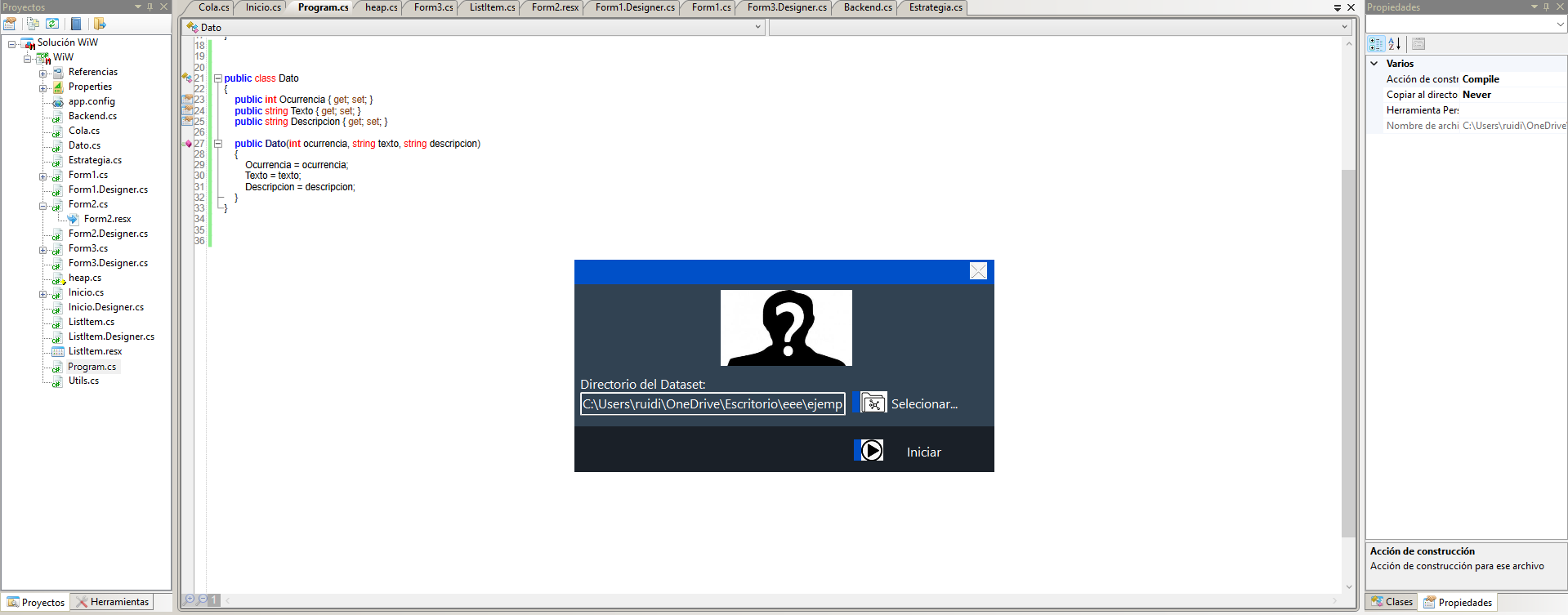
**Instrucciones para Compilar y Ejecutar:**

* Clonar o descargar el proyecto en tu entorno local.
* Abrir el proyecto con SharpDevelop.
* Compilar el proyecto.
* Adjuntar archivo ejemplo CSV ‘’Ejemplo’’.
* Ejecutar el programa y observar los resultados en la consola.

**4. Análisis de Resultados**

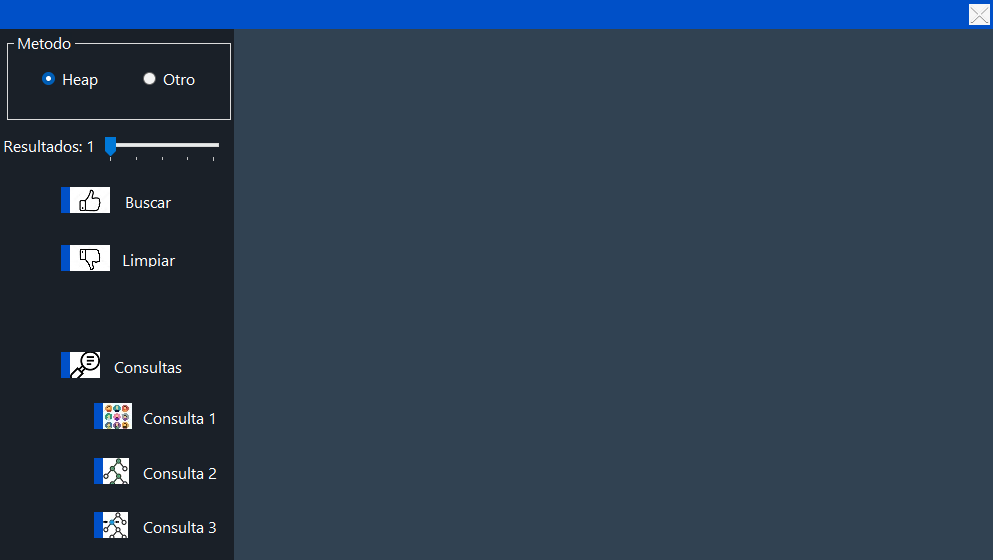
* **Compilacion e inicio de la Aplicación:**

Primero comenzamos por compilar el programa y ejecutarlo. Una vez que inicialice el sistema nos permitirá seleccionar el archivo CSV con el que deseemos trabajar. Una vez que seleccionemos un archivo con que vamos a trabajar, presionamos el botón iniciar.

****

* **Pantalla Principal y sus Operaciones**

En esta parte vamos a poder observar la pantalla inicial del sistema, donde vamos a poder observar las operaciones de búsqueda y consulta a las que podemos acceder. La misma va a contar con 2 tipos de métodos de búsqueda (heap y otro), y 3 tipos de consultas diferentes que más adelante vamos a detallar y un slider que nos permite seleccionar la cantidad de datos que necesitas mostrar en pantalla. Cabe aclarar que las consultas están atadas a los resultados obtenidos del tipo de búsqueda se que utilizo.

****

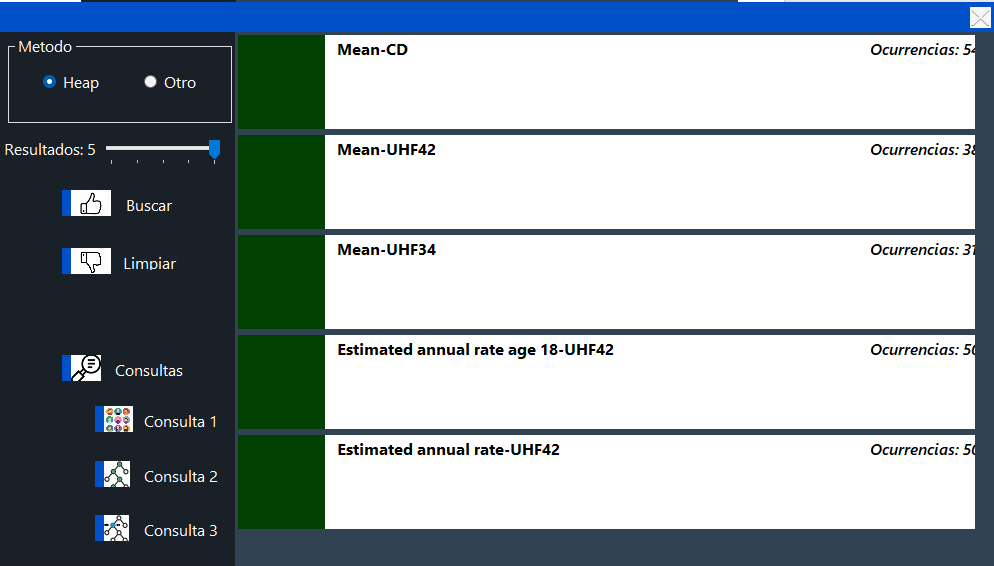
* **Búsqueda Método Heap**

En esta primera operaciones, llevamos a cabo la búsqueda de los primeros 5 datos del archivo de datos que seleccionamos al principio usando el método Heap. Esta primera búsqueda tiene como fin que el sistema devuelva los datos con mayor ocurrencia, y demostrarlo de forma descendente, es decir, mostrarlo desde mayor a menor.

Esta operación consta principalmente del método ***BuscarConHeap()***, que a partir de diferentes instrucciones y usos de otros métodos auxiliares realiza las siguientes operaciones:

* **Inicialización de Variables:**
  + Se crea un diccionario llamado ocurrencias para almacenar la cantidad de veces que aparece cada dato en la lista datos.
  + Se crea un heap máximo llamado heap para ordenar los datos según su ocurrencia.
* **Conteo de Ocurrencias:**
  + Se itera sobre cada cadena en la lista datos.
  + Se verifica si la cadena ya existe como clave en el diccionario ocurrencias. Si existe, se incrementa su valor en 1; de lo contrario, se agrega la cadena al diccionario con un valor inicial de 1.
* **Construcción del Heap:**
  + Se recorre el diccionario ocurrencias para obtener cada par clave-valor (texto-ocurrencia).
  + Se crea un nuevo objeto Dato con el texto, la ocurrencia y una descripción vacía, y se agrega al heap heap mediante el método Agregar.
* **Extracción de Datos del Heap:**
  + Se realiza un bucle for para extraer la cantidad especificada (cantidad) de datos máximos del heap.
  + En cada iteración, se extrae el dato máximo del heap utilizando el método Extraer Máximo y se agrega a la lista collected.
* **Retorno de Resultados:**
  + Se crea una copia de la lista collected en la variable ***aux***.
  + Se retorna la lista collected que contiene los datos con la mayor ocurrencia según el heap.

En resumen, este algoritmo busca y recopila los datos más frecuentes de una lista de cadenas utilizando un heap máximo para ordenarlos según su ocurrencia, y luego devuelve los datos con la mayor ocurrencia en forma de una lista de objetos Dato.

****

* **Busqueda Metodo “Otro”**

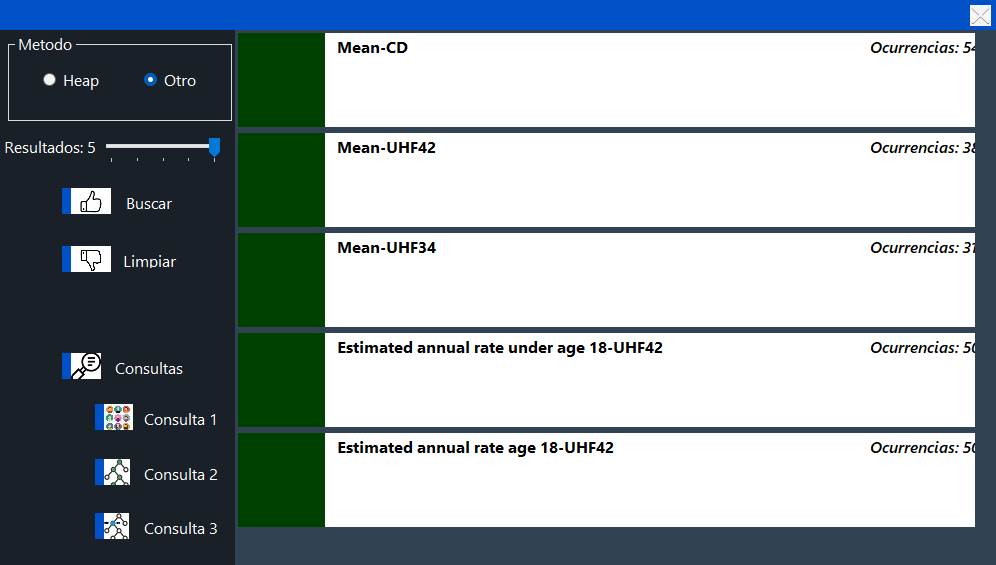
Este algoritmo representa un método llamado BuscarConOtro que toma tres parámetros: una lista de cadenas (List<string> datos), un entero (int cantidad), y una lista de objetos de tipo Dato (List<Dato> collected). El propósito de este método es buscar y recopilar datos de la lista de cadenas en base a ciertas reglas de agrupación y ordenamiento, y luego agregar los resultados a la lista collected.

Aquí está el detalle del algoritmo paso a paso:

* **Agrupación de datos por ocurrencia:**
  + Se utiliza el método GroupBy de LINQ para agrupar los elementos de la lista datos por sus valores.
  + Para cada grupo de datos, se crea un objeto Dato nuevo donde:
    - ocurrencia se establece como el número de elementos en ese grupo (obtenido mediante grupo.Count()).
    - texto se establece como la clave del grupo (grupo.Key), es decir, el valor del dato.
    - descripcion se inicializa como una cadena vacía ("") en este caso.
* **Ordenamiento de datos agrupados:**
  + Los datos agrupados se ordenan primero por su ocurrencia de mayor a menor usando OrderByDescending.
  + Luego, si hay datos con la misma ocurrencia, se ordenan por su descripción (aunque en el código proporcionado, la descripción siempre está vacía, por lo que este paso no tendrá efecto real).
* **Selección de datos y agregación a la lista de resultados:**
  + Se utiliza Take(cantidad) para seleccionar solo los primeros cantidad elementos de la lista ordenada.
  + Estos datos seleccionados se agregan uno por uno a la lista collected mediante un bucle foreach.

El algoritmo toma una lista de cadenas, cuenta cuántas veces aparece cada cadena, las ordena por número de ocurrencias de mayor a menor (y luego por descripción), y finalmente selecciona un número determinado de elementos de esa lista ordenada para agregarlos a otra lista de objetos Dato que se mostrarán en pantalla.

Como puede ver, al fin y al cabo, el resultado es el mismo, solo que en ambos casos se intenta demostrar los resultados obtenidos a partir de 2 diferentes tipos de búsqueda.



* **Consulta 1**

**Declaración de listas de resultados:**

Se crean dos listas vacías (resultadoHeap y resultadoOtro) para almacenar los resultados de las búsquedas.

**Medición del tiempo de ejecución con el método BuscarConHeap:**

Se utiliza un objeto Stopwatch para medir el tiempo de ejecución del método BuscarConHeap, que recibe la lista de datos, un número entero (en este caso, 5) y la lista resultadoHeap como parámetros. El método BuscarConHeap realiza una búsqueda específica en la lista de datos y guarda los resultados en resultadoHeap.

**Medición del tiempo de ejecución con el método BuscarConOtro:**

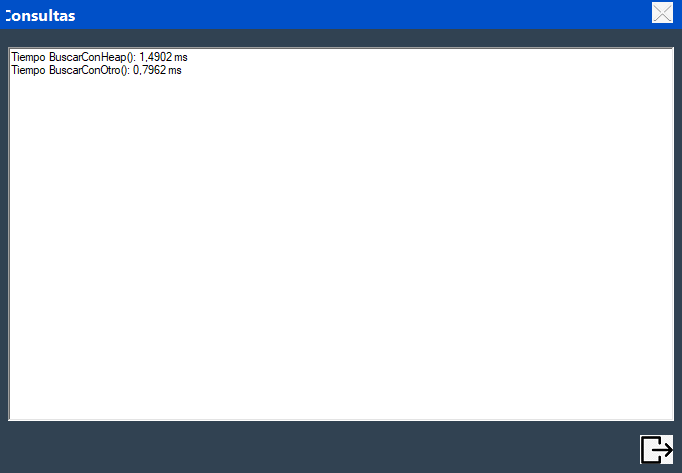
Similar al paso anterior, se utiliza otro objeto Stopwatch para medir el tiempo de ejecución del método ***BuscarConOtro***, que también recibe la lista de datos, un número entero (5) y la lista resultadoOtro como parámetros. Por otra parte el método ***BuscarConOtro*** realiza una búsqueda diferente en la lista de datos y guarda los resultados en resultadoOtro.

**Construccion de Resultado a mostrar:**

Se crea un string llamado textoTiempo que contiene información sobre los tiempos de ejecución de las búsquedas con el método BuscarConHeap y BuscarConOtro. Este texto incluye los tiempos en milisegundos de cada búsqueda, para mostrar cuánto tiempo tomó ejecutar cada método de búsqueda.

**Retorno del resultado:**

Finalmente, el método devuelve el texto textoTiempo, que contiene los tiempos de ejecución de ambas búsquedas.

****

* **Consulta 2:**

En esta operación, el sistema nos permite mostrar por pantalla el camino a la hoja más izquierda de la heap obtenida en el método BuscarConHeap().

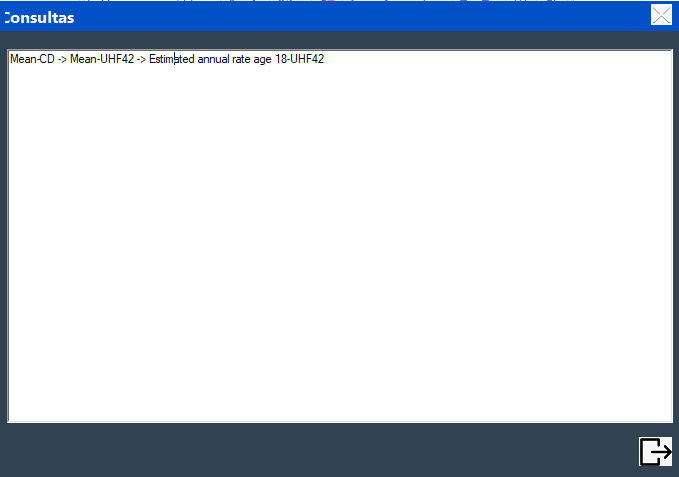
**Comprobación de lista vacía:** La primera parte del método verifica si la lista aux está vacía. Si es así, devuelve el mensaje "La lista está vacía" y termina la ejecución del método.

**Recorrido del árbol:** Se inicia un bucle while que se ejecutará hasta que se alcance una hoja en el árbol. Dentro del bucle, se calcula el índice del hijo izquierdo del nodo actual en el árbol.

**Construcción del camino:** Si el índice del hijo izquierdo está dentro de los límites de la lista aux, se actualiza nodoActual al hijo izquierdo y se agrega el texto de ese nodo al StringBuilder camino, seguido de una flecha (->) para indicar la relación entre nodos.

**Finalización del bucle:** Cuando se alcanza una hoja en el árbol (es decir, cuando el índice del hijo izquierdo supera el tamaño de la lista aux), se sale del bucle while.

**Camino:** Finalmente, se devuelve el StringBuilder convertido a una cadena de texto con el método ToString(), que representa el camino recorrido en el árbol, mostrando los textos de los nodos desde la raíz hasta la hoja más a la izquierda.

****

* **Consulta 3:**

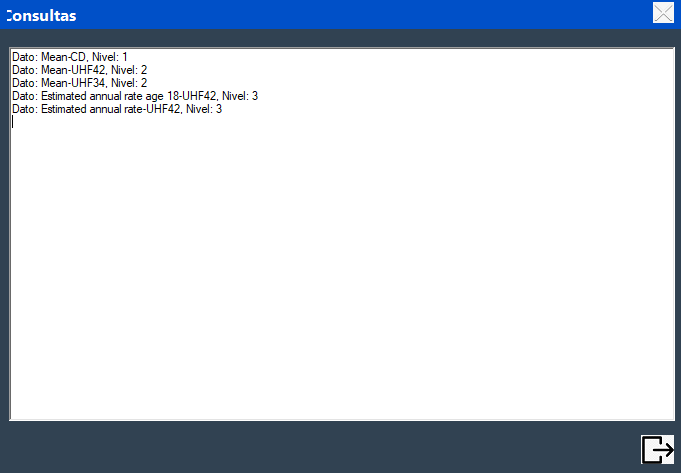
Esta operación retorna un texto que contiene los datos de la Heap que se construye a partir de los datos de entrada cuando se utiliza el método BuscarConHeap(), explicitando en el texto resultado los niveles en los que se encuentran ubicados cada uno de los datos.

**Obtención del Nivel de Cada Dato**: Para cada dato en la lista, se llama a la función privada ObtenerNivel pasando el dato actual y la lista completa de datos. Esta función calcula el nivel del dato en una estructura de heap (montículo), que es una estructura de datos jerárquica similar a un árbol binario, donde el nodo padre siempre es mayor o igual que sus nodos hijos (en un montículo máximo).

**Cálculo del Nivel**: La función ObtenerNivel calcula el nivel de un dato en la heap utilizando su índice en la lista de datos. Comienza con un nivel de 1 y se va moviendo hacia arriba en la estructura de heap hasta llegar al nodo raíz.

**Construcción del Resultado**: Dentro del bucle foreach, se construye una cadena de texto para cada dato que incluye su texto y el nivel calculado.

**Resultado**: Finalmente, el método devuelve el resultado completo como una cadena de texto utilizando el método ToString() del objeto StringBuilder.

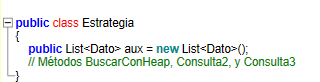
****

**5. Problemas Encontrados y Soluciones**

**NullReferenceException:**

**Problema:** Se lanzó una excepción cuando se intentaba acceder a un objeto no inicializado.

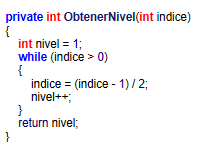
**Solución:** Asegurarse de inicializar todas las listas y objetos antes de usarlos. Esto incluye inicializar la lista aux en la clase Estrategia.



**Organización en Niveles de la Heap:**

**Problema:** Determinar en qué nivel se encuentra cada elemento en la heap.

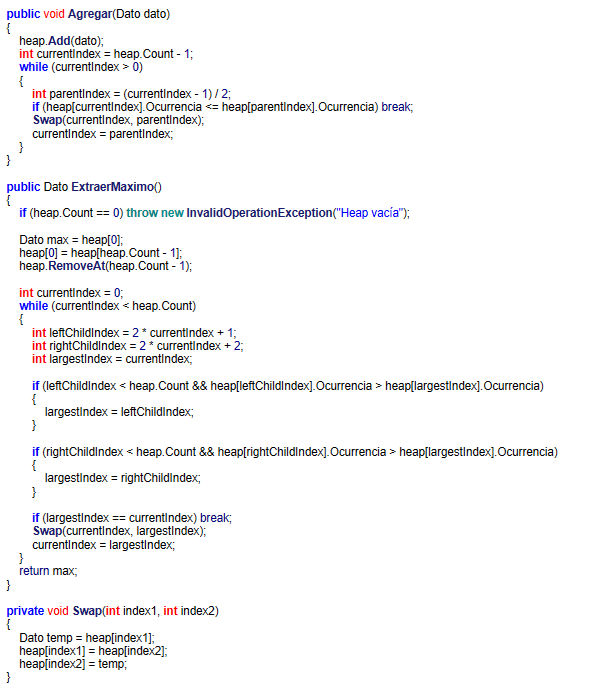
**Solución:** Implementar un método auxiliar ObtenerNivel que calcula el nivel de un elemento en base a su índice en la lista.



**Construcción y Mantenimiento de la Heap:**

**Problema:** Mantener la propiedad de la heap (montículo máximo) durante las operaciones de inserción y extracción.

**Solución:** Implementar la lógica para reorganizar la heap después de agregar un nuevo elemento o extraer el máximo. Aquí se presenta una implementación básica para agregar y extraer elementos manteniendo la propiedad de la heap:



**6. Mejoras y Sugerencias**

**Interfaz Gráfica Intuitiva:**

Desarrollar una interfaz gráfica más intuitiva utilizando tecnologías como Windows Forms o WPF para mejorar la experiencia del usuario. En este sentido, mejorar este aspecto nos va a permitir quizás la opción de poder visualizar los recorridos que se realizan en la operación ***Consulta 3,*** para su mejor comprensión.

**Optimización de la Lógica de la Heap:**

Implementar una lógica más eficiente para la manipulación de la heap, como una heap binaria, para aumentar la eficiencia en la gestión de datos.

La heap binaria simplifica las operaciones fundamentales como la inserción, eliminación y búsqueda de elementos. Estas operaciones tienen una complejidad de tiempo logarítmica O(log n), donde n es el número de elementos en la heap. En comparación, una heap general puede tener una complejidad peor, especialmente en casos donde la estructura de datos no está optimizada.

Esto nos permitirá tener un sistema escalable y confiable, ya que a partir de cierta cantidad de consultas o de procesamiento de datos, la estructura de heap binaria nos permitirá tener la mayor eficiencia posible, sin tener que alterar recursos computacionales o físicos.

**7. Reflexión Final**

Después de completar el desarrollo y la implementación de cada uno de los requerimientos establecidos en el trabajo final, puedo afirmar que, a pesar de las diversas dificultades encontradas al manejar los resultados de las operaciones, busqué alcanzar la solución óptima para cada problema. En ocasiones, opté por estrategias simplificadoras, mientras que, en otros casos, empleé enfoques totalmente novedosos que no había utilizado previamente.

Al iniciar el desarrollo del código de las búsquedas y consultas (1 y 2), me enfrenté a dificultades en la gestión de datos, así como a la interdependencia entre métodos que provocaba errores en la visualización de resultados o interrumpía la ejecución del código. Estas problemáticas me motivaron a explorar posibles soluciones que, lejos de simplificar el problema, lo complicaron aún más. Después de un análisis exhaustivo de cada clase y método, pude comprender la naturaleza del inconveniente y, a partir de ese entendimiento, desarrollé una solución que me permitió realizar cada operación de manera efectiva.