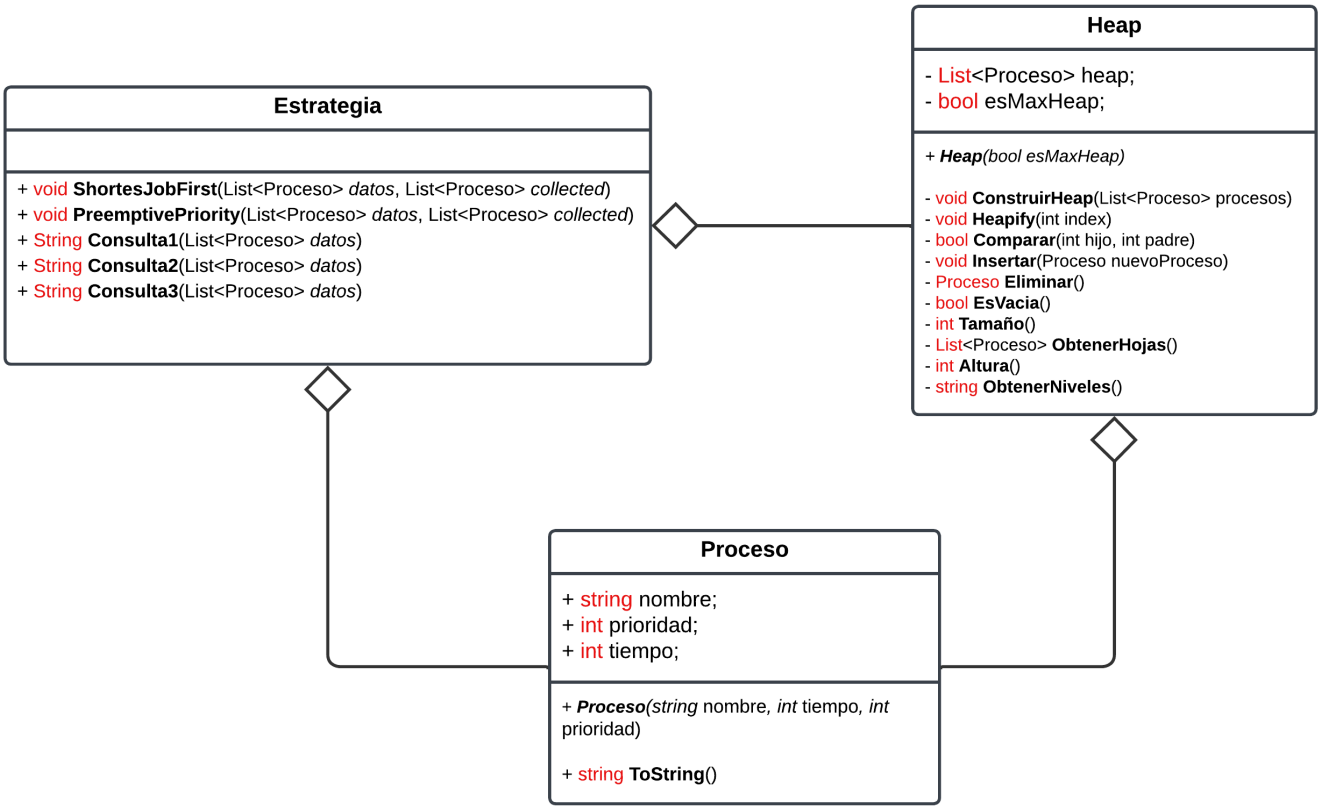
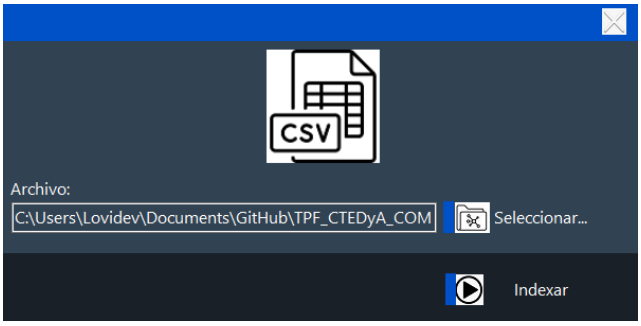
Universidad Nacional ARTURO JAURETCHE	**Complejidad temporal, estructura de datos y Algoritmos.** **TRABAJO FINAL**	**ALUMNO**: **LOVISOTTO** SANTIAGO **COMISIÓN**: 01 **PROFESOR**: Pablo José **IULIANO**

Diagrama UML:

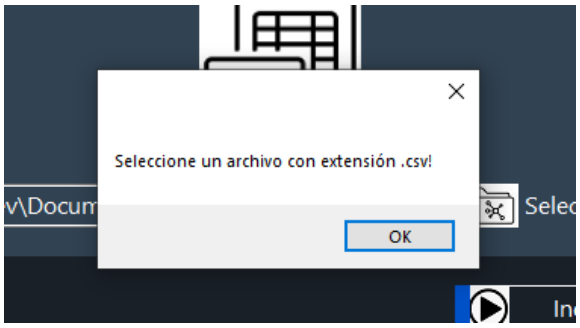


Pantallas del programa (Forms):

Ni bien ejecutamos el programa nos encontramos con una pantalla inicial que nos permite seleccionar el archivo dataset en formato .csv, en caso de seleccionar un dato invalido nos sale otra pantalla de alerta:

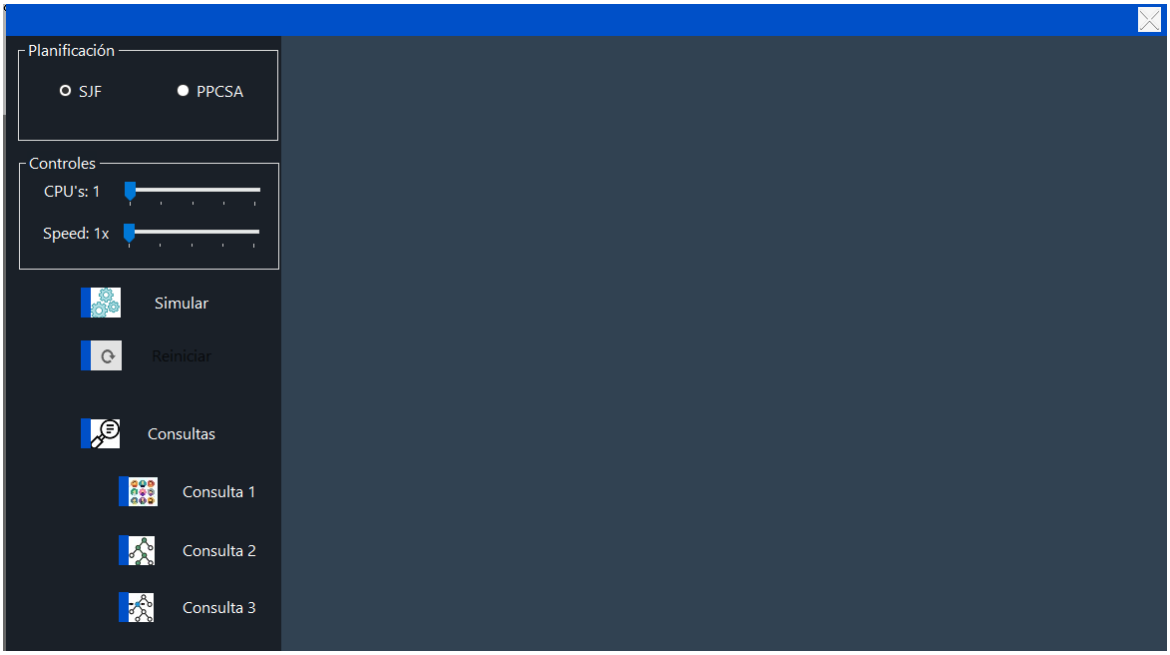


Pantalla de selección de archivo



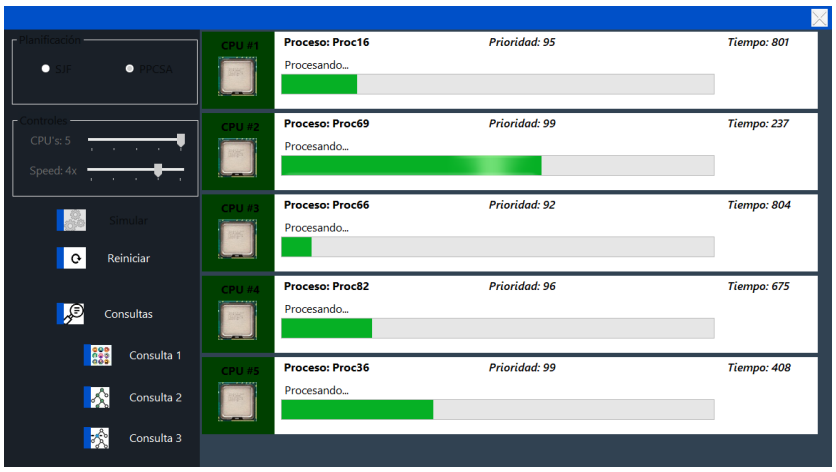
Pantalla de error por I/O

Luego, entramos a la pantalla principal donde nos permite elegir una de las opciones entre Consulta 1, Consulta 2, Consulta 3 y realizar una simulación eligiendo SJF o PPCSA:

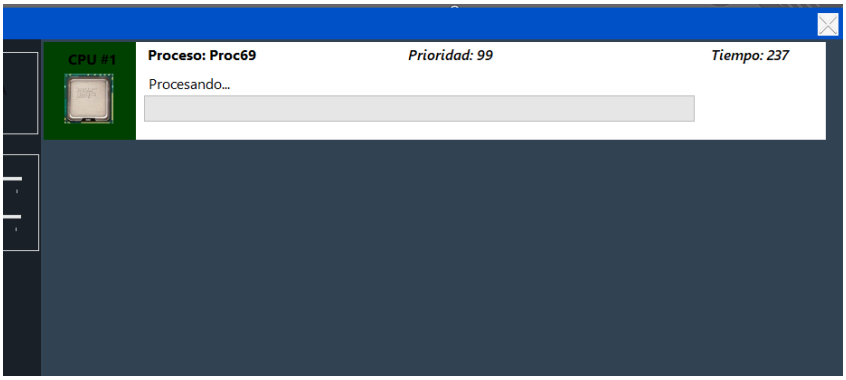


Pantalla de Inicio

Dependiendo las opciones de nuestra simulación vamos a ver 1 CPU o 5 en ejecución:

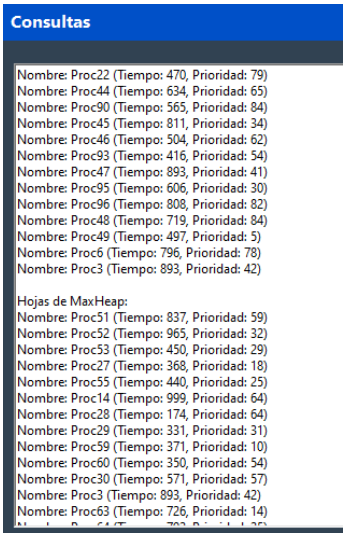


Ejecución con 5 CPU 's

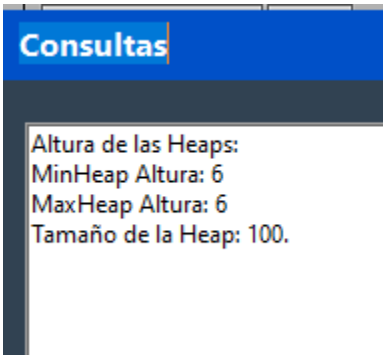


Ejecución con 1 CPU

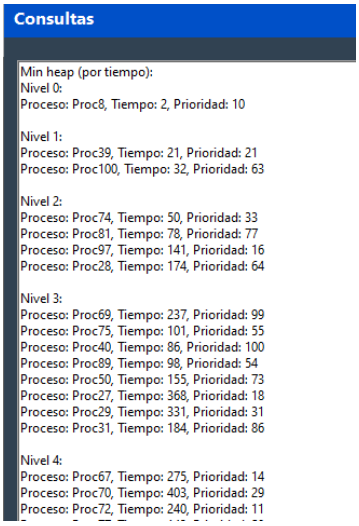
Luego, veremos una pantalla distinta por cada Consulta:



Consulta 1

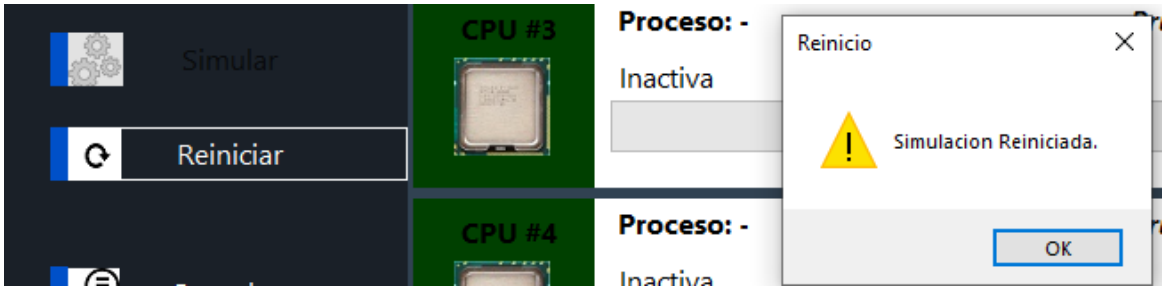


Consulta 2



Consulta 3

Además, se incluye un botón para reiniciar la simulación:



Pantalla de reinicio

***Análisis previo a el desarrollo*:**

El proyecto nos proporciona métodos para gestionar la información contenida en el archivo CSV. Contamos con una interfaz que permite seleccionar un archivo y, a partir de esta información, trabajar bajo el supuesto de que el dataset sigue el formato "nombre del proceso; tiempo de uso de la CPU; prioridad". Este análisis es crucial, ya que los métodos **ShortestJobFirst** y **PreemptivePriority** requieren un entendimiento claro de la estructura del dataset para construir las **Heaps** en función de un atributo específico.

Además, es importante determinar con que índice vamos a trabajar, yo elegí trabajar con la raíz en índice 1 para seguir con la implementación que proporcionó la cátedra:

Seguimiento del desarrollo:

1) Implementación de MinHeap:

Se desarrolló una “MinHeap” para el método **ShortestJobFirst** con los siguientes métodos: “ConstruirHeap”, “MinHeapify”, “Insertar” (no es utilizado, pero está implementado para tener una Heap más completa con utilidad a futuro), “Eliminar”, “EsVacia” y “Tamaño”. El parámetro principal considerado es el tiempo de ejecución en CPU de un proceso.

- **ConstruirHeap:** Inicializa la lista de procesos en formato heap y realiza una operación de "heapificación" desde el último nodo no hoja hacia la raíz, garantizando que se cumplan las propiedades del MinHeap.

- **MinHeapify:** Se encarga de mantener la propiedad del MinHeap al comparar un nodo con sus hijos, intercambiándolos si es necesario para que el nodo menor quede en la raíz.
- **Insertar:** Añade un nuevo proceso al final del heap y ajusta su posición subiendo el nodo hasta que se respete la propiedad de MinHeap.
- **Eliminar:** Elimina y devuelve el proceso en la raíz del heap (el proceso con menor tiempo de CPU), luego mueve el último nodo a la raíz y ajusta el heap mediante MinHeapify.
- **EsVacia:** Verifica si el heap está vacío.
- **Tamaño:** Devuelve el número de elementos en el heap.

Esta estructura fue diseñada para garantizar una eficiencia O(log N) en las operaciones críticas, aprovechando las propiedades de los árboles binarios completos. Posteriormente, la clase MinHeap fue fusionada en una clase genérica Heap, para reducir redundancias con la clase MaxHeap.

2) Implementación de MaxHeap:

Se implementó una "MaxHeap**"** para el método **PreemptivePriority**, que incluye métodos como "ConstruirHeap", "MaxHeapify", "Insertar", "Eliminar", "EsVacia" y "Tamaño", donde el parámetro principal es la prioridad de cada proceso.

Sigue una estructura similar a la MinHeap, la principal diferencia clave es qué atributo de los procesos se utiliza para organizar el heap:

- En **MinHeap**, se ordena por el **tiempo de CPU** (el menor en la raíz).
- En **MaxHeap**, se ordena por la **prioridad** (el mayor en la raíz).

Por este motivo, decidí utilizar un booleano a modo de "bandera", el cuál es pasado como parámetro en el constructor de la Heap para determinar si estamos tratanto con una min o una max Heap.

3) Primer problema encontrado:

Al observar que las clases" MinHeap" y "MaxHeap" compartían una lógica similar, con una diferencia clave en el parámetro principal (tiempo de ejecución para "MinHeap" y prioridad para MaxHeap), decidí crear una clase genérica llamada "Heap" y borrar las anteriores (*las cuales están subidas en el commit de git, por lo tanto, si se desea recuperar esa información no habría problema*).

Esta clase recibe un parámetro booleano en su constructor: si es true, la "Heap" actúa como MaxHeap; si es false, funciona como "MinHeap".

Para implementar esta lógica, se desarrolló un método privado llamado **Comparar**(hijo, padre), que utiliza el valor del booleano para determinar qué lógica aplicar en cada caso. Éste era el mismo código que se usaba individualmente en cada clase, solamente que ahora está simplificado en un método solo, permitiendo evitar la redundancia de código y una mejor implementación a futuro.

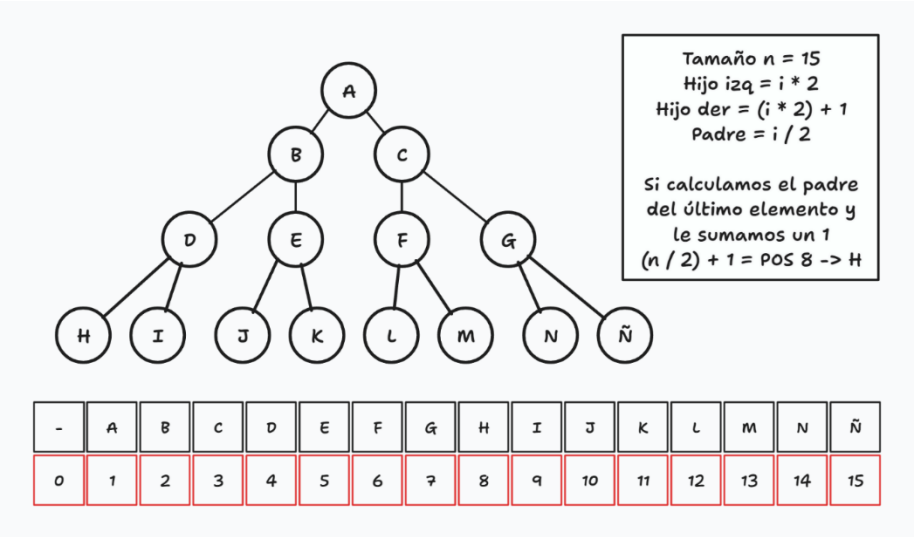
```
private bool Comparar(int hijo, int padre) {
    if (esMaxHeap)
        return heap[hijo].prioridad > heap[padre].prioridad; //! prioridad (punto 2)
    else
        return heap[hijo].tiempo < heap[padre].tiempo; //! tiempo (punto 1)
}
```

*Demostración del método **Comparar***

4) Consulta 1: Hojas de las Heaps.

Se implementó un método para devolver un texto con las hojas de las Heaps utilizadas en los métodos anteriores. Este método identifica las hojas de la "MinHeap" y la "MaxHeap", y genera un resultado que presenta la información de cada hoja de forma clara y estructurada. La presentación de los resultados se realizó utilizando concatenación de strings, mostrando Nombre, Tiempo y Prioridad de la hoja.

Voy a hacer uso de la herramienta "TL-Draw" para hacer un gráfico:



Heap con 15 elementos

Como vemos en este caso, al calcular el padre del último elemento se puede observar que todos los elementos desde el índice 8 hasta el 15 son hojas. Esto se determina utilizando la fórmula “n/2 + 1”, donde “n” es el total de nodos en la Heap. En este ejemplo, dado que la Heap tiene 15 elementos, los nodos que son hojas se encuentran en los índices 8 a 15.

La consulta se implementó siguiendo esta lógica:

- Se construyeron las Heaps correspondientes a partir de los datos de entrada utilizando los métodos “ConstruirHeap” para la “MinHeap” y la “MaxHeap”.
- Se identificaron los índices de los nodos hoja, comenzando desde el índice “n/2 + 1”.
- Se recopilaron los datos de cada nodo hoja y se formatearon en un string que incluye el nombre del proceso, el tiempo de ejecución en CPU y la prioridad.
- Finalmente, el resultado se devuelve como un texto estructurado que facilita la visualización de las hojas de las Heaps.

5) <u>Consulta 2</u>: Altura de las Heaps.

Un Heap es una estructura de datos que se organiza como un **árbol binario completo**, lo que significa que cada nivel del árbol está completamente lleno, excepto posiblemente el último nivel, que se llena de izquierda a derecha.

Dado que es un árbol binario, cada nodo puede tener como máximo dos hijos. El número total de nodos "*n*" en un árbol binario completo de altura "*h*" es como máximo:

$$N = 2^{h+1} - 1$$

Esto implica que en el nivel 0 hay 1 nodo, en el nivel 1 hay 2 nodos, en el nivel 2 hay 4 nodos, y así sucesivamente. Esta propiedad de crecimiento exponencial es lo que explica que la altura de un árbol binario sea proporcional al logaritmo base 2 del número total de nodos. Es decir, la altura "*h*" se puede expresar como:

$$h = \log_2(N)$$

El logaritmo base 2 refleja cómo el número de nodos se duplica con cada nivel adicional. Por lo tanto, para un Heap con *"n"* nodos, su altura será aproximadamente **O(log N)**, lo cual es crucial para la eficiencia en operaciones como insertar o eliminar elementos, ya que ambas dependen directamente de la altura del Heap.

En la implementación de nuestro proyecto, utilizamos un índice basado en 1 para organizar los nodos en la Heap, lo cual ya se tiene en cuenta con el método Tamaño() de nuestra clase Heap.

C# permite usar “Math.Log2” para realizar el cálculo de logaritmos (documentación adjunta en la sección de bibliografía). Sin embargo, vamos a usar una implementación distinta, utilizando operadores de desplazamiento (voy a tocar este tema al final de esta sección).

Según la documentación oficial “*Los operadores de desplazamiento bit a bit son el operador de desplazamiento a la derecha (>>), que mueve los bits de una expresión de tipo entero o de enumeración a la derecha, y el operador de desplazamiento izquierdo (<<), que mueve los bits a la izquierda.*”

A continuación, se presenta el código utilizado para calcular la altura de un Heap utilizando el operador de desplazamiento a la derecha:

```
public int Altura() {
    int n = this.Tamaño();
    int altura = 0;

    while (n > 1) {
        n >>= 1;
        altura++;
    }

    return altura;
}
```

Implementación de logaritmo utilizando operadores de desplazamiento

Consideremos que el tamaño del Heap es 100 (son 101 elementos, pero el 0 no lo tenemos en cuenta). El número **100** en binario es ***1100100***. Al aplicar el operador de desplazamiento bit a bit, el número se va dividiendo sucesivamente por 2:

1. n = 100 → binario: *1100100*
   Desplazamiento → *110010* (50 en decimal)
   Altura = 1

2. n = 50 → binario: *110010*
   Desplazamiento → *11001* (25 en decimal)
   Altura = 2

3. n = 25 → binario: *11001*
   Desplazamiento → *1100* (12 en decimal)
   Altura = 3

4. n = 12 → binario: *1100*
   Desplazamiento → 110 (6 en decimal)
   Altura = 4

5. n = 6 → binario: *110*
   Desplazamiento → 11 (3 en decimal)
   Altura = 5

6. n = 3 → binario: *11*
   Desplazamiento → 1 (1 en decimal)
   Altura = 6

El ciclo se detiene cuando n llega a 1. Por lo tanto, la altura del Heap con 100 nodos es 6, que es el logaritmo base 2 de 100, redondeado hacia abajo.

¿Tiene sentido este resultado?
Anteriormente analizamos que el número total de nodos se calcula con la fórmula:

$$N = 2^{h+1} - 1$$

Esto nos permite estimar los posibles valores de "h" dado un tamaño de Heap "n". Si tenemos n= 100 y h= 6, buscamos el rango en el que se encuentra:

$$N = 2^{h+1} - 1 = 2^{6+1} - 1 = 128 - 1 = 127$$

Esto nos dice que el máximo número de nodos en un Heap de altura 6 es 127. Un Heap de tamaño 100 puede entrar en un Heap con altura 6 porque está dentro de este límite.

Para un h=5 y mismo n:

$$N = 2^{h+1} - 1 = 2^{5+1} - 1 = 64 - 1 = 63$$

El máximo número de nodos para un Heap de altura 5 es 63, la cual no es suficiente para contener 100 nodos.
Por lo tanto, un Heap con 100 nodos tiene una altura de **6** porque 100 está entre **64 y 127**. Es decir:

- El mínimo número de nodos para una altura 6 es 64, y el máximo es 127.
- Dado que el tamaño del Heap es 100, la altura adecuada es 6, ya que se encuentra en ese intervalo de posibles tamaños.

Esta es una verificación clara de que, para 100 nodos, la altura logarítmica resultante es 6.

6) <u>Consulta 3</u>: Niveles de las Heaps.

En esta consulta, se busca obtener un texto que contenga los datos de los procesos organizados por niveles dentro de la estructura de la Heap.

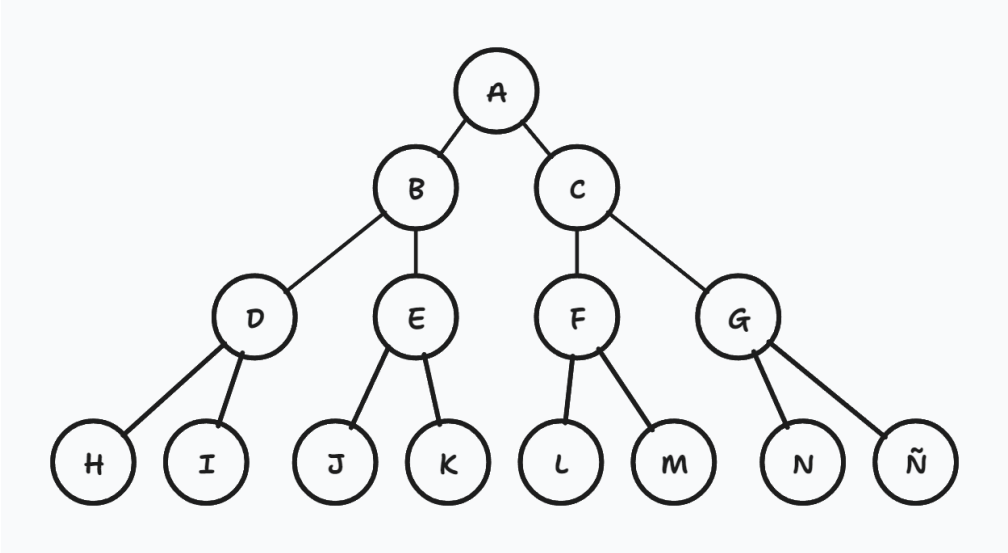
La lógica utilizada es similar a la empleada en la Consulta 2, donde calculamos la altura de la Heap utilizando operaciones de desplazamiento, que son equivalentes al cálculo de logaritmos.

Cada nivel en un Heap binario se puede determinar al dividir el índice de cada proceso hasta que este llegue a 1. Este proceso de división es análogo a calcular el logaritmo base 2 del índice.

<u>Explicación del código</u>:

1. **Recorrido de Nodos**: Iteramos sobre cada nodo de la Heap desde el índice 1 hasta el tamaño de la Heap.
2. **Cálculo de Nivel**: Para cada índice, se calcula su nivel mediante desplazamientos a la derecha en un bucle while, que continúa hasta que el índice sea 1. Cada desplazamiento cuenta como un incremento en el nivel.
3. **Agrupación por Niveles**: Si se detecta un cambio en el nivel (es decir, si el nivel actual es diferente del nivel previo), se agrega un encabezado para ese nivel en el resultado.
4. **Salida Formateada**: Finalmente, se recopila la información de cada proceso y se añade al resultado, que es devuelto como una cadena de texto.

Tomemos de ejemplo una Heap con 15 elementos:



Heap con 15 elementos

Proceso de Cálculo de Niveles

1. **Índice 1** (a):
    - índice = 1 (binario: 1)
    - Nivel = 0 (no se desplaza).

2. **Índice 2** (b):
    - índice = 2 (binario: 10)
    - Desplazamos: 2 >> 1 = 1 (nivel = 1).

3. **Índice 3** (c):
    - índice = 3 (binario: 11)
    - Desplazamos: 3 >> 1 = 1 (nivel = 1).

4. **Índice 4** (d):
    - índice = 4 (binario: 100)
    - Desplazamos: 4 >> 1 = 2 → 2 >> 1 = 1 (nivel = 2).

5. **Índice 5** (e):
    - índice = 5 (binario: 101)
    - Desplazamos: 5 >> 1 = 2 → 2 >> 1 = 1 (nivel = 2).

6. **Índice 6** (f):
    - índice = 6 (binario: 110)
    - Desplazamos: 6 >> 1 = 3 → 3 >> 1 = 1 (nivel = 2).

7. **Índice 7** (g):
    - índice = 7 (binario: 111)
    - Desplazamos: 7 >> 1 = 3 → 3 >> 1 = 1 (nivel = 2).

8. **Índice 8** (h):
    - índice = 8 (binario: 1000)
    - Desplazamos: 8 >> 1 = 4 → 4 >> 1 = 2 → 2 >> 1 = 1 (nivel = 3).

9. **Índice 9** (i):
    - índice = 9 (binario: 1001)
    - Desplazamos: 9 >> 1 = 4 → 4 >> 1 = 2 → 2 >> 1 = 1 (nivel = 3).

10. **Índice 10** (j):
    - índice = 10 (binario: 1010)
    - Desplazamos: 10 >> 1 = 5 → 5 >> 1 = 2 → 2 >> 1 = 1 (nivel = 3).

11. **Índice 11** (k):
    - índice = 11 (binario: 1011)
    - Desplazamos: 11 >> 1 = 5 → 5 >> 1 = 2 → 2 >> 1 = 1 (nivel = 3).

12. **Índice 12** (l):
    - índice = 12 (binario: 1100)
    - Desplazamos: 12 >> 1 = 6 → 6 >> 1 = 3 → 3 >> 1 = 1 (nivel = 3).

13. **Índice 13** (m):
    - índice = 13 (binario: 1101)
    - Desplazamos: 13 >> 1 = 6 → 6 >> 1 = 3 → 3 >> 1 = 1 (nivel = 3).

14. **Índice 14** (n):
    - índice = 14 (binario: 1110)
    - Desplazamos: 14 >> 1 = 7 → 7 >> 1 = 3 → 3 >> 1 = 1 (nivel = 3).

15. **Índice 15** (ñ):
    - índice = 15 (binario: 1111)
    - Desplazamos: 15 >> 1 = 7 → 7 >> 1 = 3 → 3 >> 1 = 1 (nivel = 3).

Agrupando por niveles:

- **Nivel 0**: Índice 1 (a)
- **Nivel 1**: Índices 2 (b), 3 (c)
- **Nivel 2**: Índices 4 (d), 5 (e), 6 (f), 7 (g)
- **Nivel 3**: Índices 8 (h), 9 (i), 10 (j), 11 (k), 12 (l), 13 (m), 14 (n), 15 (ñ)

Así es como se calcula el nivel de cada nodo en la Heap utilizando desplazamientos en binario. Cada desplazamiento a la derecha divide el índice por 2, lo que nos permite encontrar la profundidad en el árbol binario.

¿Por qué usé desplazamiento en lugar de Math.log?

Dado que el propósito de este trabajo final es ***aprender***, al implementar el cálculo de la altura utilizando desplazamientos, se obtiene un entendimiento más profundo de cómo funcionan los números en su representación binaria.

Además, encontré otras ventajas:

- **Eficiencia Computacional**: Los operadores de desplazamiento son generalmente más rápidos que las operaciones matemáticas complejas. Por ejemplo, los desplazamientos se realizan en un solo ciclo de CPU, mientras que calcular logaritmos puede implicar operaciones más complejas que tardan más tiempo. Una curiosidad es este estudio de "www.dotnetperls.com" que mostró que la operación de desplazamiento a la derecha es más rápida que la división por 4 (0.8 ns frente a 0.95 ns, enlace en la parte de bibliografía).
- **Eliminación de Dependencias Externas**: Al evitar el uso de bibliotecas externas como *Math.Log*, se reduce la dependencia del código en funciones predefinidas, lo que puede llevar a una mayor portabilidad y comprensión del código, teniendo la posibilidad de llevar el código a otro lenguaje.

- **Simplicidad y Claridad**: Al usar desplazamientos, el código se mantiene más simple y fácil de entender, sin depender de funciones externas como Math.Log. Esto hace que el código sea más transparente y comprensible, ya que no depende de fórmulas matemáticas predefinidas. Además, al implementar la lógica por mi cuenta, me permitió aprender más sobre cómo funcionan los números en su formato binario, algo que antes no tenía en cuenta.

Posibles mejoras a futuro:

Como se mencionó anteriormente, implementé una clase general Heap para evitar la repetición de código. Sin embargo, en las primeras etapas, cuando aún existían las clases MaxHeap y MinHeap por separado, consideré la posibilidad de permitir al usuario elegir qué tipo de Heap utilizar desde la pantalla de inicio.

Interfaz de selección:

Al llegar a la pantalla de selección, además de elegir el archivo CSV, el usuario tiene la opción de seleccionar la estrategia de Heap que desea usar:

1. **MinHeap** (para Shortest Job First)
2. **MaxHeap** (para Preemptive Priority)

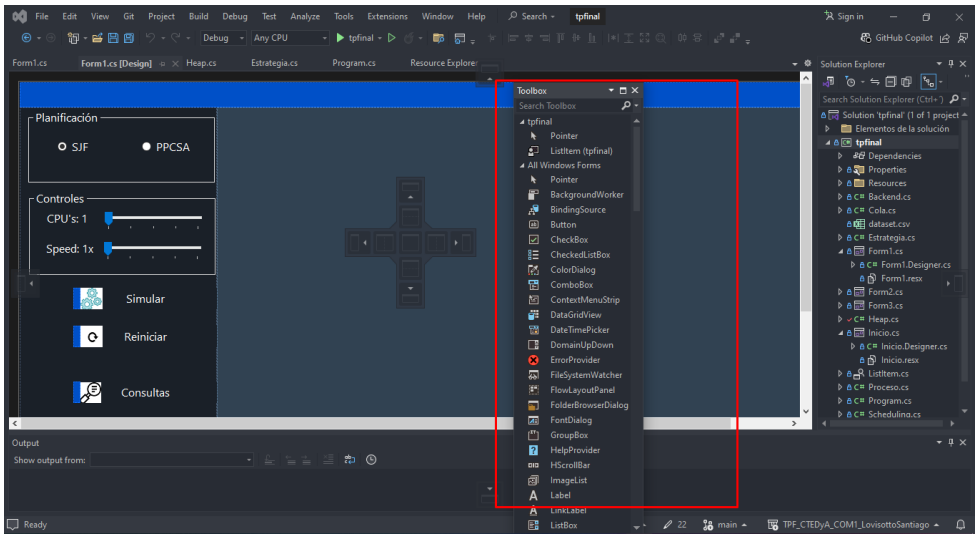
Procesamiento de la elección:

Según la opción seleccionada por el usuario, se inicializa la Heap correspondiente:

- Si se selecciona **MinHeap**, se aplicará la lógica de SJF.
- Si se selecciona **MaxHeap**, se aplicará la lógica de PPCSA.

Esto puede implementarse fácilmente utilizando **Forms**, añadiendo controles como RadioButton en la pantalla de selección, lo que permitirá al usuario elegir la estrategia de Heap antes de cargar el archivo CSV y continuar con la simulación.

Forms nos dejaría arrastrar objetos en pantalla mediante una interfaz como la siguiente:



Otra implementación a futuro sería implementar un sistema de simulación paso a paso, que permita visualizar el estado de la Heap en cada inserción.

Conclusión:

Este trabajo final me resultó muy interesante y, sobre todo, un gran aprendizaje. Lo más importante del proyecto fue estructurar bien la clase Heap, ya que todo el sistema se basa en ella. Lograr que la clase Heap estuviera optimizada fue clave para que los algoritmos de planificación, como Shortest Job First (SJF) y Preemptive Priority Scheduling (PPCSA), funcionaran de manera eficiente. Esto me hizo dar cuenta de lo crucial que es elegir y organizar correctamente las estructuras de datos para que el sistema tenga un buen rendimiento y que el código sea legible para poder implementar funciones en el futuro.

Este proyecto no solo me sirvió para afianzar conocimientos, sino que también me hizo pensar en cómo voy a abordar la implementación de estructuras de datos en proyectos futuros (e incluso en los que estoy trabajando ahora en GitHub).

Pienso que todo lo que aprendí acá va a tener un gran impacto en la manera en que desarrollo mis proyectos actuales y futuros, especialmente en cuanto a la eficiencia y la organización del código.

Bibliografía y documentación (además de la cátedra)**:**

- "https://publish.obsidian.md/algoritmos/Estructuras+de+datos/Heap" documentación sobre Heaps.
- "https://learn.microsoft.com/en-us/dotnet/api/system.math.log2?view=net-8.0" logaritmos en C#.
- "https://learn.microsoft.com/es-es/cpp/cpp/left-shift-and-right-shift-operators-input-and-output?view=msvc-170" operadores de desplazamiento.
- "https://www.geeksforgeeks.org/c-sharp-math-log-method/" otro artículo sobre logaritmos en C#.
- "https://www.luisllamas.es/csharp-operadores-bitwise/" otro artículo sobre operadores de desplazamiento en C#.
- "https://www.dotnetperls.com/divide-powers-two" estudio de DotNetPerls.