**Desafio#1 Informática 2. Álvaro Gómez, Juan David Jiménez.**

**Puntos para tener en cuenta**

**A).** Análisis del problema y consideraciones para la alternativa de solución propuesta.

**B).** Esquema donde describa las tareas que usted definió en el desarrollo de los algoritmos.

**C).** Algoritmos implementados.

**D.)** Problemas de desarrollo que afrontó. e. Evolución de la solución y consideraciones para tener en cuenta en la implementación.

**Modulo 1.**

#### **A). Análisis del Problema:**

El problema consiste en desarrollar un módulo que permita crear estructuras de datos de tamaño variable, con espacio en blanco en el centro, como se indica en el código y en las especificaciones del main proporcionado.

#### Consideraciones Iniciales:

**Creación de la Estructura M:** Se debe implementar una función **crearEstructuraM(int tamano)** que reciba como parámetro el tamaño deseado para la estructura de datos y devuelva una instancia de la estructura **EstructuraM** con las características especificadas.

**Tamaño Impar:** Se debe asegurar que el tamaño de la estructura **EstructuraM** sea impar, aumentándolo en 1 si es par, para cumplir con la condición de tener espacio en blanco en el centro.

**Memoria Dinámica:** La matriz de enteros dentro de la estructura **EstructuraM** se debe crear utilizando asignación de memoria dinámica.

**Rellenar la Matriz:** Se debe llenar la matriz con valores -1 (espacios en blanco) en todas las posiciones excepto en el centro, donde se llenará con números del 1 al tamaño^2 excluyendo el centro.

#### Alternativa de Solución Propuesta:

Para abordar este problema, se propone la siguiente alternativa de solución:

**Función crearEstructuraM(int tamano):**

Verificar si el tamaño es par y aumentarlo en 1 si es necesario para asegurar que sea impar.

Crear la estructura **EstructuraM** y asignar memoria dinámica para la matriz de enteros.

Llenar la matriz con valores -1 en todas las posiciones.

Rellenar el centro de la matriz con números del 1 al tamaño^2 excluyendo el centro.

**Consideraciones de Eficiencia:**

Utilizar la asignación de memoria dinámica de manera eficiente para evitar problemas de fragmentación de memoria.

Implementar un algoritmo eficiente para rellenar la matriz y asegurar que cumpla con las condiciones requeridas.

**Gestión de Memoria:**

Implementar la función **liberarEstructuraM(EstructuraM &estructura)** para liberar la memoria asignada a la estructura **EstructuraM** cuando ya no sea necesaria.

**Modularidad y Reutilización:**

Mantener la separación de archivos (estructuraM.h y estructuraM.cpp) para una estructura modular y facilitar la reutilización del código.

Proporcionar una función **imprimirEstructuraM(EstructuraM &estructura)** para mostrar por pantalla la estructura creada.

Con esta alternativa de solución, se busca crear un módulo eficiente y modular que permita generar estructuras de datos de tamaño variable, con espacio en blanco en el centro, cumpliendo con las especificaciones indicadas en el problema. Además, se presta atención a la gestión de memoria y se garantiza la modularidad y reutilización del código para un desarrollo más organizado y mantenible.

**B). +-------------------------------+**

**| Inicio del Programa |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Solicitar Tamaño de la M |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Verificar si Tamaño es Par |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Ajustar Tamaño si es Par |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Crear EstructuraM Vacía |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Rellenar Centro con -1 |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Rellenar Resto con Números |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Devolver EstructuraM |**

**+-------------------------------+**

**|**

**V**

**+-------------------------------+**

**| Fin del Programa |**

**+-------------------------------+**

**C).** **Algoritmos implementados:**

**1.** crearEstructuraM

**2.** liberarEstructuraM

**3.** imprimirEstructuraM

**D.) Verificación de Tamaño Par o Impar:**

Uno de los desafíos fue verificar si el tamaño proporcionado por el usuario era par o impar. Esto implicó el uso de una condición para incrementar el tamaño en 1 si era par, asegurando así que la estructura resultante tuviera un tamaño impar.

**Asignación de Memoria Dinámica:**

La asignación de memoria dinámica para la matriz de enteros dentro de la estructura **EstructuraM** fue un punto crítico. Se necesitó implementar correctamente la asignación de memoria para evitar fugas de memoria y problemas de fragmentación.

**Relleno de la Matriz:**

Otra dificultad fue llenar la matriz con valores -1 en todas las posiciones y luego rellenar el centro con números del 1 al tamaño^2 excluyendo el centro. Esto requirió el uso de bucles anidados y lógica adicional para asegurar que se cumplieran las condiciones.

**Gestión de Errores y Excepciones:**

Se tuvo que considerar la gestión de errores y excepciones, como, por ejemplo, el manejo de situaciones donde la asignación de memoria falla debido a la falta de memoria disponible en el sistema.

**Optimización de Eficiencia:**

Se planteó el desafío de optimizar la eficiencia del algoritmo utilizado para crear y llenar la matriz, especialmente en casos de tamaños grandes donde la complejidad podría afectar el rendimiento.

**Modularidad y Reutilización del Código:**

Para garantizar el modularidad y la reutilización del código, se debió separar la implementación en archivos header (estructuraM.h) y de implementación (estructuraM.cpp), asegurando así una estructura organizada y mantenible.

**E). Manejo de Tamaños Par e Impar:**

Es fundamental tener en cuenta que el tamaño de la estructura debe ser impar para garantizar que haya un espacio en blanco en el centro. Si el usuario da un tamaño par, requiere una lógica de verificación y ajuste del tamaño.

**Asignación de Memoria Dinámica:**

La asignación de memoria dinámica para la matriz debe ser cuidadosamente gestionada para evitar fugas de memoria y problemas de fragmentación. Se recomienda liberar la memoria correctamente una vez que la estructura ya no sea necesaria para evitar pérdidas de memoria.

**Eficiencia en el Relleno de la Matriz:**

Es importante optimizar el proceso de llenado de la matriz para garantizar un rendimiento eficiente, especialmente en casos de matrices grandes. Se pueden explorar técnicas como el uso de bucles eficientes y algoritmos optimizados para el llenado de la matriz.

**Gestión de Errores y Excepciones:**

Se deben implementar mecanismos robustos de manejo de errores y excepciones para abordar situaciones inesperadas, como la falta de memoria durante la asignación dinámica o errores en la entrada proporcionada por el usuario.

**Documentación y Comentarios:**

Es crucial documentar el código de manera clara y proporcionar comentarios significativos para facilitar la comprensión y el mantenimiento del código a lo largo del tiempo. Esto incluye explicar el propósito de cada función, los parámetros que recibe y los valores que devuelve.

**Modularidad y Reutilización del Código:**

Se debe seguir el principio de diseño modular para separar la implementación en archivos header (estructuraM.h) y de implementación (estructuraM.cpp). Esto facilita la reutilización del código y asegura una estructura organizada y mantenible.

**Pruebas Exhaustivas:**

Antes de considerar la implementación como completa, es esencial realizar pruebas exhaustivas para validar el correcto funcionamiento de todas las funciones en diferentes escenarios, incluyendo tamaños pequeños y grandes de la estructura.

**Optimización y Mejoras Continuas:**

Una vez que la implementación básica esté funcionando correctamente, se pueden explorar oportunidades de optimización y mejoras continuas para mejorar el rendimiento y la eficiencia del módulo.

**Modulo 2:**

**A.) Análisis del problema:**

El problema consiste en implementar un módulo que permita realizar rotaciones en una estructura de datos M en sentido antihorario un número específico de veces. Para ello, se cuenta con las siguientes piezas:

**Función rotarAntihorario(EstructuraM &estructura, int rotaciones):** Esta función recibe como parámetro una estructura de datos M y un entero que indica el número de rotaciones a realizar en sentido antihorario. La función realiza las rotaciones necesarias y actualiza la estructura de datos M con la matriz rotada.

**Función imprimirEstructuraM(EstructuraM &estructura):** Esta función se encarga de imprimir la estructura de datos M en la consola.

**Consideraciones para la alternativa de solución propuesta:**

**Validación de entrada:** Se debe asegurar que el número de rotaciones sea un valor positivo. Si se ingresa un valor negativo, se puede mostrar un mensaje de error y solicitar al usuario que ingrese un número válido.

**Optimización de memoria:** Al realizar múltiples rotaciones, es importante optimizar el uso de memoria para evitar fugas de memoria. Se debe asegurar la liberación adecuada de la memoria de las matrices temporales utilizadas durante las rotaciones.

**Eficiencia del algoritmo:** Se debe evaluar la eficiencia del algoritmo de rotación para garantizar un rendimiento óptimo, especialmente en estructuras de datos grandes o al realizar un gran número de rotaciones.

**Interacción con el usuario:** Es importante proporcionar mensajes claros y detallados al usuario durante la ejecución del programa, especialmente al solicitar el número de rotaciones y al mostrar la estructura de datos M rotada.

**Pruebas exhaustivas:** Se deben realizar pruebas exhaustivas del módulo para verificar su correcto funcionamiento bajo diferentes escenarios, incluyendo casos límite como el número mínimo y máximo de rotaciones permitidas, así como estructuras de datos M de diferentes tamaños.

**B).** +-----------------------------------+

| **Inicio del programa**  |

+-----------------------------------+

|

V

+-----------------------------------+

| **Solicitar cantidad de rotaciones |**

+-----------------------------------+

|

V

+-----------------------------------+

| **Rotar estructura M antihorario** |

+-----------------------------------+

|

V

+-----------------------------------+

| **Imprimir estructura M**  |

+-----------------------------------+

|

V

+-----------------------------------+

|  **Liberar memoria de estructura M**|

+-----------------------------------+

|

V

+-----------------------------------+

|  **Fin del programa** |

+-----------------------------------+

**C).**

**1.**rotarAntihorario

**2.**imprimirEstructuraM

**D). Gestión de memoria:** Al manipular matrices dinámicas y realizar rotaciones, es crucial gestionar adecuadamente la memoria para evitar fugas de memoria o errores de acceso a memoria no válida.

**Implementación correcta de las rotaciones:** Es necesario asegurarse de que el algoritmo de rotación antihorario funcione correctamente para el número especificado de rotaciones y en todas las posiciones de la matriz.

**Eficiencia:** Dependiendo del tamaño de la matriz y el número de rotaciones requeridas, podría surgir un problema de eficiencia si no se implementa de manera óptima el algoritmo de rotación.

**Manejo de casos especiales:** Se deben considerar casos especiales como matrices de tamaño 1x1, manejo de rotaciones negativas, o rotaciones que superen el tamaño de la matriz.

**Validación de entradas:** Es importante validar las entradas del usuario, como el número de rotaciones, para asegurarse de que sean valores válidos y evitar errores durante la ejecución.

**E). Optimización del algoritmo de rotación:** Se debe buscar un algoritmo eficiente para realizar las rotaciones en sentido antihorario. Una opción común es usar una técnica de rotación en lugar de reorganizar los elementos uno a uno, lo que puede mejorar significativamente el rendimiento, especialmente para matrices grandes y múltiples rotaciones.

**Gestión de memoria:** Es fundamental manejar adecuadamente la asignación y liberación de memoria para evitar fugas de memoria y errores de acceso. Se deben utilizar punteros de manera responsable y liberar la memoria correctamente después de cada rotación.

**Validación de entradas:** Se deben validar las entradas proporcionadas por el usuario, como el número de rotaciones, para asegurarse de que sean valores válidos y no causen comportamientos inesperados o errores en el programa.

**Manejo de casos especiales:** Es importante considerar casos especiales como matrices de tamaño 1x1, rotaciones negativas o rotaciones que excedan el tamaño de la matriz. Se deben implementar lógicas específicas para manejar estos casos de manera adecuada y evitar errores.

**Pruebas exhaustivas:** Antes de implementar las funciones en producción, es esencial realizar pruebas exhaustivas para asegurarse de que funcionen correctamente en una variedad de escenarios y casos de uso. Las pruebas deben incluir matrices de diferentes tamaños y números de rotaciones.

**Documentación y comentarios claros:** Es recomendable proporcionar una documentación clara de las funciones de rotación, incluyendo sus parámetros, comportamiento esperado y posibles casos especiales. Además, agregar comentarios en el código para explicar la lógica detrás de las operaciones realizadas puede facilitar la comprensión y mantenimiento del código en el futuro.

**A).**

**Modulo 3:** Configurar cerraduras de la cerradura X con estructuras M variables

El problema consiste en desarrollar un módulo que permita configurar cerraduras X compuestas por múltiples estructuras M de diferentes tamaños. La cantidad de estructuras M y sus dimensiones (filas y columnas) deben ser configurables por el usuario.

**Para abordar este problema, se propone la siguiente alternativa de solución:**

-Crear una estructura Cerradura que contenga un vector de EstructuraM.

-Implementar una función configureCerradura() que solicite al usuario la cantidad de estructuras M a incluir en la cerradura.

-Para cada estructura M, solicitar al usuario el tamaño (filas y columnas) y crear una instancia de EstructuraM utilizando la función crearEstructuraM(int tamano) del punto 1.

-Agregar cada EstructuraM creada al vector dentro de la estructura Cerradura.

-Devolver la Cerradura configurada con las estructuras M especificadas.

**Consideraciones:**

-Reutilizar la función crearEstructuraM(int tamano) del punto 1 para crear cada estructura M dentro de la cerradura.

-Permitir una configuración flexible de la cerradura, donde el usuario pueda especificar la cantidad y tamaño de las estructuras M.

**Modulo 4:** Validar una regla K sobre una configuración dada de la cerradura X

El problema consiste en implementar funciones que permitan validar si una regla K se cumple para una configuración dada de la cerradura X.

**Para abordar este problema, se propone la siguiente alternativa de solución:**

-Implementar una función validarRegla(const Cerradura& cerradura, const vector<int>& regla) que reciba una cerradura configurada y una regla K como parámetros.

-Dentro de la función, verificar si la regla K tiene un formato válido y si los índices de fila, columna y estructura están dentro de los límites permitidos.

-Iterar sobre cada comparación en la regla K y extraer los índices de fila, columna y estructuras M involucradas, así como el signo de comparación.

-Acceder a los valores de las celdas correspondientes en las estructuras M de la cerradura utilizando los índices extraídos.

-Realizar las comparaciones indicadas por la regla K entre los valores de las celdas y verificar si se cumple la condición.

-Devolver true si todas las comparaciones de la regla K se cumplen, o false en caso contrario.

**Consideraciones:**

-Utilizar las funciones implementadas en los puntos 1 y 2 para acceder y manipular las estructuras M dentro de la cerradura.

-Manejar adecuadamente los casos de error, como reglas K con formato incorrecto o índices fuera de rango.

b) Esquema donde se describen las tareas definidas en el desarrollo de los algoritmos.

**Modulo 3:** Configurar cerradura

**+-----------------------------------+**

**| configureCerradura() |**

**+-----------------------------------+**

**|**

**V**

**+----------------------------------------------+**

**| Solicitar cantidad de estructuras M |**

**+----------------------------------------------+**

**|**

**V**

**+-----------------------------------+**

**| Crear Cerradura vacía |**

**+-----------------------------------+**

**|**

**V**

**+--------------------------------------------+**

**| Repetir para cada estructura M |**

**+--------------------------------------------+**

**|**

**V**

**+-------------------------------------------+**

**| Solicitar tamaño de estructura M |**

**+---------------------------------- --------+**

**|**

**V**

**+--------------------------------------------------------------------+**

**| Crear EstructuraM con crearEstructuraM(tamano) |**

**+--------------------------------------------------------------------+**

**|**

**V**

**+------------------------------------------------+**

**| Agregar EstructuraM a la Cerradura |**

**+------------------------------------------------+**

**|**

**V**

**+---------------------------------+**

**| Devolver Cerradura |**

**+---------------------------------+**

**Modulo 4: Validar regla K**

+----------------------------------------------+

| validarRegla(cerradura, regla) |

+----------------------------------------------+

|

V

+---------------------------------------------+

| Verificar formato válido regla |

+---------------------------------------------+

|

V

+-----------------------------------------+

| Verificar índices dentro de |

| límites |

+-----------------------------------------+

|

V

+----------------------------------+

| Repetir para cada |

| comparación |

+----------------------------------+

|

V

+----------------------------------------------------+

| Extraer índices fila, columna, |

| estructuras y signo de comparación |

+----------------------------------------------------+

|

V

+--------------------------------------------------------+

| Acceder valores celdas correspondientes |

| en estructuras M |

+--------------------------------------------------------+

|

V

+---------------------------------------------+

| Realizar comparación indicada |

| por regla K |

+---------------------------------------------+

|

V

+------------------------------------+

| Si comparación falla, |

| devolver false |

+------------------------------------+

|

V

+-----------------------------------------+

| Si todas comparaciones OK, |

| devolver true |

+-----------------------------------------+

**c) Algoritmos implementados.**

**Modulo 3: Configurar cerradura**

Función configureCerradura():

Cerradura cerradura

Solicitar al usuario la cantidad de estructuras M

Para cada estructura M:

Solicitar al usuario el tamaño (filas y columnas)

EstructuraM estructura = crearEstructuraM(tamaño)

Agregar estructura a cerradura

Fin Para

Devolver cerradura

Fin Función

**Modulo 4: Validar regla K**

Función validarRegla(cerradura, regla):

Si formato de regla no es válido:

Devolver false

Para cada comparación en la regla:

Extraer índices de fila, columna, estructuras y signo de comparación

Si índices fuera de límites:

Devolver false

Obtener valores de celdas correspondientes en estructuras M

Si comparación no se cumple:

Devolver false

Fin Para

Devolver true

Fin Función

**d) Problemas de desarrollo que afrontó.**

**Modulo 3: Configurar cerradura**

Durante el desarrollo de la funcionalidad para configurar cerraduras, se enfrentaron los siguientes problemas:

**1. Manejo de entrada de usuario:** Fue necesario implementar una lógica adecuada para solicitar y validar la entrada del usuario al especificar la cantidad de estructuras M y los tamaños de cada una.

**2. Integración con el punto 1:** La función configureCerradura() debía reutilizar la función crearEstructuraM(int tamano) desarrollada en el punto 1. Esto implicó asegurarse de que las interfaces y los tipos de datos fueran compatibles entre ambos módulos.

**3. Gestión de memoria:** Al agregar múltiples estructuras M a la cerradura, se debió prestar atención a la gestión adecuada de la memoria asignada dinámicamente para cada estructura.

**Modulo 4: Validar regla K**

En el desarrollo de la funcionalidad para validar la regla K, se presentaron los siguientes problemas:

**1. Formato de la regla K:** Fue necesario implementar una lógica para verificar si el formato de la regla K proporcionada era válido y cumplía con los requisitos especificados.

**2. Manejo de índices:** Al extraer los índices de fila, columna y estructura desde la regla K, se debió asegurar que estos índices estuvieran dentro de los límites permitidos para cada estructura M dentro de la cerradura.

**3. Acceso a valores de celdas:** Para realizar las comparaciones indicadas por la regla K, se requirió acceder a los valores de las celdas correspondientes en las diferentes estructuras M de la cerradura, lo cual implicó un manejo cuidadoso de los índices y la iteración sobre las estructuras.

**4. Detección de errores:** Se implementaron mecanismos para detectar y manejar adecuadamente los casos de error, como reglas K con formato incorrecto o índices fuera de rango.

**e) Evolución de la solución y consideraciones para tener en cuenta en la implementación.**

**Modulo 3: Configurar cerradura**

La solución para configurar cerraduras evolucionó a medida que se analizaban los requisitos y se identificaban las interacciones necesarias con las funcionalidades desarrolladas en los puntos anteriores. Inicialmente, se planteó una solución básica que involucraba la creación de una estructura Cerradura y la adición de estructuras M a esta.

Sin embargo, se identificaron aspectos clave que requerían una atención especial, como la integración con la función crearEstructuraM(int tamano) del punto 1, el manejo de la entrada de usuario y la gestión adecuada de la memoria.

Para abordar estos desafíos, se implementaron las siguientes consideraciones:

**1. Reutilización de código:** Se reutilizó la función crearEstructuraM(int tamano) del punto 1 para crear cada estructura M dentro de la cerradura. Esto permitió aprovechar el trabajo previo y mantener una consistencia en la implementación.

**2. Manejo de entrada de usuario:** Se implementó una lógica para solicitar al usuario la cantidad de estructuras M y el tamaño de cada una. Se incluyeron validaciones para asegurarse de que los tamaños ingresados cumplieran con los requisitos (número de filas impar y igual al número de columnas).

**3. Gestión de memoria:** Se prestó atención a la gestión adecuada de la memoria asignada dinámicamente para cada estructura M dentro de la cerradura. Esto se logró mediante la reutilización de la función liberarEstructuraM(EstructuraM &estructura) desarrollada en el punto 1.

**Modulo 4: Validar regla K**

La solución para validar la regla K evolucionó a medida que se analizaban los requisitos y se identificaban los desafíos inherentes al problema. Inicialmente, se planteó una solución básica que involucraba la comparación de los valores de las celdas indicadas por la regla K entre las estructuras M correspondientes.

Sin embargo, se identificaron aspectos clave que requerían una atención especial, como la verificación del formato de la regla K, el manejo de índices fuera de rango y la detección de errores.

Para abordar estos desafíos, se implementaron las siguientes consideraciones:

**1. Verificación del formato de la regla K:** Se implementó una lógica para verificar si el formato de la regla K proporcionada era válido y cumplía con los requisitos especificados. Esto incluyó la validación del número de elementos en la regla y el rango de valores permitidos.

**2. Manejo de índices:** Se implementaron mecanismos para verificar que los índices de fila, columna y estructura extraídos de la regla K estuvieran dentro de los límites permitidos para cada estructura M dentro de la cerradura. Esto ayudó a evitar accesos fuera de rango y comportamientos indefinidos.

**3. Acceso a valores de celdas:** Se desarrolló una lógica para acceder a los valores de las celdas correspondientes en las diferentes estructuras M de la cerradura, utilizando los índices extraídos de la regla K. Esto implicó un manejo cuidadoso de los índices y la iteración sobre las estructuras.

**4. Detección de errores:** Se implementaron mecanismos para detectar y manejar adecuadamente los casos de error, como reglas K con formato incorrecto o índices fuera de rango. Esto incluyó la devolución de valores adecuados y la generación de mensajes de error descriptivos.

En resumen, la evolución de la solución para los puntos 3 y 4 se centró en abordar los desafíos específicos identificados durante el desarrollo, como la integración con el código existente, el manejo de la entrada de usuario, la gestión de memoria, la verificación del formato de la regla K, el manejo de índices fuera de rango y la detección de errores. Estas consideraciones permitieron obtener una solución robusta y funcional que cumple con los requisitos establecidos.