Universidad de San Carlos de Guatemala

Centro Universitario de Occidente

División de Ciencias de la Ingeniería

Estructuras de Datos

Laboratorio

Proyecto 1

Juego de “Laberintos”

Manual Técnico

Javier Eduardo Ixcolín Orozco

202132163

1. **HERRAMIENTAS UTILIZADAS**

C++:

C++ es un lenguaje de programación de propósito general, orientado a objetos y de alto rendimiento, desarrollado como una extensión del lenguaje C. Es ampliamente utilizado en el desarrollo de software de sistemas, aplicaciones de escritorio, videojuegos, y aplicaciones que requieren un control preciso sobre los recursos del sistema. C++ combina la eficiencia y el control de bajo nivel de C con características modernas como clases, herencia, polimorfismo y plantillas (templates), lo que lo hace ideal para proyectos que demandan tanto rendimiento como flexibilidad. Su sintaxis es poderosa y expresiva, permitiendo a los desarrolladores crear código modular, reutilizable y mantenible.

Visual Studio Code:

Visual Studio Code es un editor de código fuente desarrollado por Microsoft que es altamente personalizable y admite una amplia gama de lenguajes de programación. Ofrece características como resaltado de sintaxis, finalización de código, depuración y control de versiones integrado. Visual Studio Code es conocido por su rendimiento y su amplia gama de extensiones.

Git:

Git es un sistema de control de versiones distribuido ampliamente utilizado para el seguimiento de cambios en el código fuente durante el desarrollo de software. Permite a los desarrolladores colaborar en proyectos, realizar un seguimiento de las modificaciones, fusionar ramas y revertir cambios. Git es conocido por su velocidad, escalabilidad y capacidad para manejar proyectos grandes.

GitHub:

GitHub es una plataforma de desarrollo colaborativo que utiliza Git para el control de versiones. Permite a los desarrolladores alojar y revisar el código, gestionar proyectos, realizar seguimiento de problemas y colaborar en equipos distribuidos. GitHub es conocido por su amplia comunidad, su integración con otras herramientas y su soporte para proyectos de código abierto.

Archivo makefile:

Los Makefile son unos archivos que se incluyen en la carpeta raíz de un proyecto que le dicen a un programa, make, que se ejecuta desde la terminal, qué hacer con cada uno de los archivos de código para compilarlos. En este caso, es utilizado para crear un archivo ejecutable de todo el código realizado.

2. ESTRUCTURAS CREADAS

Lista Doble Enlazada

Todos los archivos mencionados a continuación se encuentran en la carpeta doubleLinkedList. La clase Node representa un nodo en una lista doble enlazada. Su función principal es almacenar punteros de tipo SparseMatrix, la cuál es una clase que representa una matriz dispersa, y mantener referencias al nodo siguiente (next) y al nodo anterior (prev) en la lista. Esta estructura permite la navegación bidireccional, lo que facilita operaciones como la inserción, eliminación y recorrido en ambas direcciones. En este caso se escogió dicha estructura para facilitar el recorrido de las matrices dispersas, ya que de este modo no es necesario recorrer toda la lista en caso el jugador se encuentre entre las últimas matrices dispersas de la lista.

Atributos:

- **matrix:** Almacena un objeto de tipo SparseMatrix, que representa la información contenida en el nodo.
- **next:** Puntero al siguiente nodo en la lista. Si el nodo es el último, este puntero será nullptr.
- **prev:** Puntero al nodo anterior en la lista. Si el nodo es el primero, este puntero será nullptr.

Métodos y funciones:

- **DoubleLinkedList():** Constructor por defecto de la lista doble enlazada.
- **~DoubleLinkedList():** Liberador de memoria de la lista doble enlazada.
- **Int length() const:** Retorna el tamaño de la lista.
- **SparseMatrix *getNext(int iterator) const:** Retorna un nodo en la lista dependiendo de la posición en la que se encuentre el jugador al momento de iniciar el juego.
- **SparseMatrix *getPrev(int iterator) const:** Retorna un nodo en la lista dependiendo de la posición en la que se encuentre el jugador al momento de iniciar el juego.
- **SparseMatrix *getFirst() const:** Retorna el primer nodo de la lista.
- **void insertNext(SparseMatrix *sparseMatrix):** Inserta un nodo en la lista doble enlazada, dependiendo de la posición en la que se desee.
- **void insertPrev(SparseMatrix *sparseMatrix):** Inserta un nodo en la lista doble enlazada. Dependiendo de la posición en la que se desee.
- **void append(SparseMatrix *sparseMatrix):** Inserta un nodo en la lista doble enlazada. Este método es utilizado principalmente cuando se está llenando la lista.

Colas

Todos los archivos mencionados a continuación se encuentran en la carpeta queue. En el proyecto se implementaron dos colas. Los archivos de la primer cola son Queue y QueueNode, mientras que de la segunda cola son GameQueue y GameNode. La primer cola es utilizada para la lectura del archivo .csv, mientras que la segunda cola es utilizada para el registro de movimientos del

jugador durante el juego. Las clases GameNode y QueueNode representan nodos en sus respectivas colas, mientras que las clases GameQueue y Queue representan las colas.

Matriz Dispersa

Todas las clases mencionadas a continuación se encuentran en la carpeta sparseMatrix. La clase CellNode representa un nodo-celda dentro de la matriz dispersa, mientras que la clase Header-Node representa un nodo-cabecera dentro de la matriz dispersa. Ambos tipos de nodos almacenan tipos de datos string y sus respectivos punteros hacia las 4 direcciones. La clase HeadersList representa una lista de los nodos cabecera dentro de la matriz dispersa. La clase SparseMatrix es la matriz dispersa, en la cuál se terminan manejando el comportamiento de todos los nodos, junto con algunos métodos o funciones.

3. COMPLEJIDAD DE LAS ESTRUCTURAS CREADAS

DoubleLinkedList

- **Constructor**
  - Complejidad: O(1)
  - Se inicializan los punteros head y tail a nullptr y el tamaño a 0; es una asignación directa sin bucles.
- **Destructor**
  - Complejidad: O(n)
  - Se recorre la lista (desde head hasta tail) para liberar cada nodo. En el peor caso, se realiza una iteración para cada nodo de la lista, por lo que la complejidad depende linealmente del número de nodos (n).
- **Método length():**
  - Complejidad: O(1)
  - Devuelve el valor de size, que se mantiene actualizado en cada operación de modificación. No implica ninguna iteración.
- **Método getNext(int iterator):**
  - Complejidad: O(n) (en el peor caso)
  - Se parte del nodo head y se recorre la lista avanzando (iterator) veces mediante un bucle for.
  - En el peor de los casos (cuando iterator es cercano al tamaño de la lista), se recorre gran parte o la totalidad de la lista.
- **Método getPrev(int iterator):**
  - Complejidad: O(n) (en el peor caso)
  - Se inicia en el nodo tail y se retrocede iterator - 1 veces mediante un bucle for.
  - Similar a getNext, si el valor de iterator es grande, se recorrerá una cantidad lineal de nodos.
- **Método getFirst():**
  - Complejidad: O(1)
  - Simplemente devuelve el dato del nodo apuntado por head, sin necesidad de realizar bucles o búsqueda.

- **Método append(SparseMatrix sparseMatrix)**
  - Complejidad: O(1)
  - Si la lista está vacía, se asignan los punteros head y tail al nuevo nodo.
  - Si ya existen nodos, se actauliza el puntero next del nodo tail y el puntero prev del nuevo nodo, y se mueve el puntero tail al nuevo nodo.
  - Estas operaciones se realizan en tiempo constante, sin importar el tamaño de la lista.

Queue

- **Constructor**
  - Complejidad: O(1)
  - Inicializa el puntero front a nullptr y el tamaño a 0. Operaciones de asignación directa.
- **Destructor**
  - Complejidad: O(n)
  - Llama a clear(), que recorre y elimina todos los nodos de la cola.
- **Método enqueue(const string &data)**
  - Complejidad: O(n)
  - Crea un nuevo nodo en O(1)
  - Si la cola está vacía, asinga fornt = newNode en O(1)
  - Si no, recorre la lista desde front hasta encontrar el último nodo (puntero next nulo), lo cual en el peor caso visita todos los n nodos, por tanto O(n).
  - Actualiza size en O(1)
- **Método dequeue()**
  - Complejidad: O(1)
  - Comprueba si la cola está vacía en O(1)
  - Guarda el nodo frontal, actualiza front = fornt -> getNext(), lee el dato, elimina el nodo y decrementa size en O(1).
- **Método length()**
  - Complejidad: O(1)
  - Devuelve el valor de size
- **Método clear()**
  - Complejidad: O(n)
  - Recorre la cola desde front eliminando cada nodo hasta que front sea nullptr. En el peor caso elimina n nodos.
- **Método print()**
  - Complejidad: O(n)
  - Recorre la cola dese front hasta el final, imprimiendo cada dato. Visita cada uno de los n nodos.
- **Método isEmpty()**
  - Complejidad: O(1)
  - Comprueba si size == 0.

SparsMatrix

- **Constructor**
  - Complejidad: O(1)
  - Inicializa los encabezados de filas y columnas; no recorre estructuras.
- **Destructor**
  - Complejidad: O(r + c + k)
  - r = número de filas con datos (nodos HeaderNode en rows)
  - c = número de columnas con datos (nodos HeaderNode en columns)
  - k = número total de celdas (nodos CellNode)
  - Recorre cada fila: para cada HeaderNode de fila (r), recorre su lista de CellNode (sumando k) en O(r + k).
  - Luego recorre la lista de HeaderNode de columnas (c) en O(c)
- **Método insert(int x, int y, string value)**
  - Obtener o crear encabezados: Rows.get(x) y columns.get(y) recorren listas de HeaderNode en O(r) y O(c) respectivamente.
  - Obtener o crear encabezados: Si no existe, rows.add y columns.add añaden al final en O(1) (asumiendo puntero tail).
  - Insertar en la fila: Si la fila está vacía, O(1)
  - Insertar en la fila: Si no, recorre la lista de CellNode de la fila hasta la posición correcta en O(m) en el peor caso.
  - Insertar en la columna: Recorre la lista de la columna en O(n)
- **Método editValueRD(int x, int y, string value)**
  - Complejidad; O(r + m)
  - r = tiempo de rows.get(x)
  - M = número de CellNode en la fila x hasta encontrar la columna y
  - Busca el headerNode de la fila en O(r)
  - Recorre sus CellNode a la derecha hasta encontrar la posición y o llegar al final en O(m)
- **Método editValueLU(int x, int y, string value)**
  - Complejidad: O(c + n)
  - c = tiempo de columns.get(y)
  - n = número de CellNode en la columna y hasta encontrar la fila x
  - Busca el HeaderNode de columna en O(c)
  - Recorre sus CellNode hacia abajo hasta encontrar la posición x o llegar al final en O(n)
- **Método search(int x, int y) const**
  - Complejidad: O(r + m)
  - Similar a editValueRD, donde re es para rows.get(y) y me es el número de celdas recorridas en la fila y
- **Método printMatrix()**
  - Complejidad: O(R * C * (r + m))
  - R = rows.length() número total de filas (posiciones posibles)
  - C = columns.length() número total de columnas
  - Cada llamada a search(i, j) es O(r + m)

- Dos bucles anidados que recorren todas las posiciones (R * C)
- En cada posición llaman a search(i, j), con coste O(r + m)