Estructuras de Datos 2022-1
Práctica 3: Pilas y Colas

Pedro Ulises Cervantes González
confundeme@ciencias.unam.mx

Emmanuel Cruz Hernández
emmanuel_cruzh@ciencias.unam.mx

Yessica Janeth Pablo Martínez
yessica_j_pablo@ciencias.unam.mx

América Montserrat García Coronado
ame_coronado@ciencias.unam.mx

Adrián Felipe Vélez Rivera
adrianf_velez@ciencias.unam.mx

Fecha límite de entrega: 31 de octubre de 2021
Hora límite de entrega: 23:59:59 hrs

1. Objetivo

Aplicar la técnica de *Backtraking* para encontrar la solución a un problema, con apoyo de una Pila o Stack. En particular, resolver un laberinto, buscando caminos y retrocediendo a estados válidos en caso de no encontrar salida en alguno de los recorridos.

2. Actividad

La compañía diseñadora de juegos *Game Gaming* te ha contratado para desarrollar un programa que permita saber si un laberinto tiene solución o no. En caso de tener solución, se debe mostrar el recorrido que lo resuelve. Para lograrlo, se solicitó utilizar la técnica de *Backtraking*.

2.1. Stack *(2.5 puntos)*

Implementa la interfaz *TDAStack* en una clase llamada *Stack*. Implementación basada en referencias.

2.2. Queue *(2.5 puntos)*

Implementa la interfaz *TDAQueue* en una clase llamada *Queue*. Implementación basada en referencias.

2.3. Clase casilla *(1 punto)*

7 Crea una clase llamada *Box* que cuente con los siguientes atributos:

- *boolean wall*: true si la casilla representa una pared, false si representa una casilla disponible para buscar.
- *boolean visited*: true si la casilla ya fue visitada. Sólo se pueden visitar las casillas que no han sido visitadas y que no son paredes.
- *Queue neighbors*: una cola que almacena los números de 0 a 3, insertados aleatoriamente. Cada número representará el orden en que se recorrerá al siguiente vecino si es posible (que no sea pared y no esté visitado)

1. Arriba
2. Derecha
3. Abajo
4. Izquierda

Puedes agregar todos los atributos que consideres convenientes para representar una casilla. Además, debes implementar los siguientes métodos:

- isWall: permite saber si una casilla es pared o no.
- isVisited: permite saber si una casilla está visitada o no.
- visit: visita la casilla

```
1   public boolean isWall(){
2       ...
3   }
4
5   public boolean isVisited(){
6       ...
7   }
8
9   public void visit(){
10      ...
11  }
```

2.4. Clase laberinto *(3 puntos)*

Crea una clase *Maze* que representa un laberinto. Esta clase cuenta con los siguientes atributos.

- Un arreglo de *nxm* que almacena casillas. Esta es la representación del tablero de laberinto.
- La casilla de inicio. Este objeto de tipo *Box* determina el inicio de recorrido del laberinto, por lo que debe ser una casilla que no es pared.
- La casilla de fin de recorrido. Este objeto de tipo *Box* determina la casilla objetivo o final del laberinto, cuando se encuentra un recorrido desde el inicio hasta esta casilla entonces el laberinto tiene solución.
- La casilla actual del recorrido. Este objeto de tipo *Box* corresponde a la casilla en el recorrido actual. Cuando se extiende un estado *s*, esta casilla es la que se actualiza, modificando el valor de la casilla actual después del cambio para *s'*.

Puedes agregar todos los atributos extra que consideres necesarios.

Para crear un nuevo laberinto, se lee la información desde un archivo de texto, donde los dos primeros renglones corresponden a la cantidad de renglones y columnas, respectivamente. El resto de renglones corresponde a un par separado por una coma, que representa el índice de las casillas que son camino. **Puedes apoyarte de la clase *ArrayReader***, ajustando el código del método `readMaze(String)` a tu diseño.

Por ejemplo, el archivo *LaberintoA.txt*

```
21
21
1,1
1,2
1,3
1,4
1,5
1,7
1,8
1,9
1,11
1,12
1,13
1,14
1,15
1,17
...
```

corresponde al laberinto siguiente:

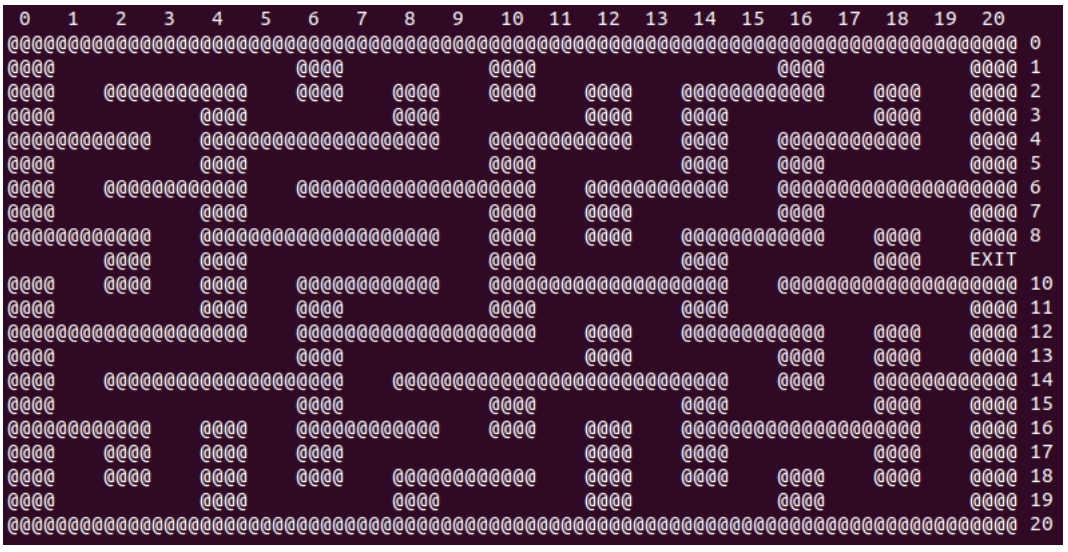

Figura 1: Laberinto asociado al archivo LaberintoA.txt

Por otra parte, se deben implementar los métodos siguientes en la clase *Maze*:

- isSolution: regresa true si el laberinto está resuelto.
- isExtensible: regresa true si la casilla actual tiene vecinos por los cuales extender.
- extend: mueve la casilla actual a una casilla vecina que no sea pared y no haya sido visitada. Esta casilla se elige a partir del método `dequeue()` del atributo *neighbors* de la casilla actual.
- solve: encuentra una solución al laberinto, si existe. En caso de existir regresa la solución en una estructura de datos (TDAList, TDAStack, TDAQueue) con el recorrido hecho del inicio a fin, en caso de no existir se regresa la estructura vacía (Dar la solución representada en pantalla es opcional).
- toString: muestra la representación de un tablero. **La representación del tablero resuelto es opcional.**

```java
public boolean isSolution(){
    ...
}

public boolean isExtensible(){
    ...
}

public void extend(){
    ...
}

// Versión con listas
public TDAList<Box> solve(){
    ...
}

// Versión con pilas
public TDAStack<Box> solve(){
    ...
}

// Versión con colas
public TDAQueue<Box> solve(){
    ...
}

@Override
public String toString(){
    ...
}
```

2.5. Método main *(1 punto)*

Crea un menú en el método *main* de la clase *Maze*, que ponga a prueba en consola el funcionamiento del buscador de soluciones a un laberinto.

Las operaciones que se podrán hacer en el laberinto son:

- Resolver laberinto
    1. Escribir el nombre del archivo donde está la representación del laberinto.
    2. Mostrar el laberinto
    3. Seleccionar la casilla de inicio
    4. Seleccionar la casilla de fin
    5. Resolver laberinto y mostrar la solución (en casillas o de forma gráfica)
- Cerrar

3. Recursos de apoyo

- Backtraking: `http://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-11.pdf`
- Escritura de ficheros: `https://youtu.be/E0H4OzW2_1Y`

- Lectura de ficheros: `https://youtu.be/etQN4EfYN7k`
- Ejemplo de lectura de ficheros: `https://github.com/EmmanuelCruz/Material-ICC-2021-1/tree/master/9.%20Entrada%20y%20Salida`

4. Reglas Importantes

- No se recibirán prácticas en las que estén involucrados más de dos integrantes.
- Cumple con los lineamientos de entrega.
- Todos los archivos deberán contener nombre y número de cuenta.
- Tu código debe estar comentado. Esto abarca clases, atributos, métodos y comentarios extra.
- Utiliza correctamente las convenciones para nombrar variables, constantes, clases y métodos.
- El programa debe ser robusto.
- En caso de no cumplirse alguna de las reglas especificadas, se restará 0.5 puntos en tu calificación obtenida.
- **Queda estrictamente prohibido usar una implementación de listas, pilas o colas hecha por Java, de lo contrario la actividad se evaluará sobre 0**.
- **Esta práctica no tendrá prórroga**.