Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2022-2

27 de Octubre de 2022
Actividad Sumativa

Actividad Sumativa 4

Estructuras Nodales I y II

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AS4/`
- **Hora del *push*:** 16:40

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción: D(ul)CCe o Truco

Se acerca el final de Octubre y con él una semana de disfraces, dulces y otras oportunidades. Para aprovechar al máximo estas fiestas, decides crear un programa que te ayude a modelar distintas situaciones a través de estructuras nodales, de manera que estés preparado para lo que se venga.

Estructura del programa

Esta actividad consta de tres partes, en las cuales se te pedirá que implementes funciones que trabajen con un grafo, un árbol binario y una lista ligada.

Parte 1

Le has pedido a uno de los profesores del ramo que te preste un disfraz que puedas usar. El profesor te responde que no hay problema, pero que el único disfraz que tenía se le perdió mientras visitaba una **Mansión Embrujada**, por lo que deberás recorrer cada una de sus habitaciones para buscar las partes del disfraz entre sus objetos.

Tu objetivo corresponde a implementar los métodos faltantes de la clase `Mapa` para poder recorrer el grafo y revisar sus nodos.

Archivos:

- mansion.py: Se encarga de definir las clases **MansionEmbrujada**, **Habitacion**, **Mapa** y **Explorador**

Clases:

- No modificar `class Habitacion`: Representa una habitación del mapa, es decir, un nodo del grafo.
  - No modificar `def __init__(self, id:int, nombre:str, id_vecinos:list, objetos:list) -> None:`

    Inicializa cada habitación con los siguientes atributos:
    - `self.id:` id de la habitación.
    - `self.nombre:` nombre de la habitación.
    - `self.objetos:` lista con los objetos, en formato `string`, que se encuentran dentro de la habitación.
    - `self.id_vecinos:` lista que contiene los id's de las habitaciones vecinas.
    - `self.conexiones:` lista que contiene las instancias de `Habitacion` que rodean al nodo actual. Esta lista parte vacía.
    - `self.explorador:` Un `bool` que indica si la habitación ya fue explorada.
  - No modificar `def __str__(self) -> str:` Retorna el nombre de la Habitación.

- No modificar `class MansionEmbrujada`: Guarda los datos físicos de la mansión para poder registrarlos en el Mapa.
  - No modificar `def explorar(self, id) -> tuple`: Recibe el id de una `Habitacion` y retorna todos los datos necesarios para instanciarla.

- Modificar `class Mapa`: Clase que representa el mapa del explorador, es decir, el grafo. Se le van añadiendo Habitaciones a medida que se explora la mansión.

- No modificar `def __init__(self, punto_de_partida: Habitacion) -> None`: Inicializa el Mapa con una Habitacion como punto de partida. Crea una lista con las instancias de las Habitaciones ya creadas.
- Modificar `def crear_habitacion(self, id, nombre, id_vecinos, objetos) -> Habitacion:` Si el id recibido coincide con alguna de las instancias contenidas en `self.habitaciones_creadas`, retorna la instancia existente, de lo contrario, crea una nueva instancia de `Habitacion` con los argumentos entregados, la agrega a la lista de Habitaciones creadas y retorna **la instancia de creada**.
- Modificar `def registrar_vecino(self, habitacion: Habitacion, vecino: Habitacion) -> None:`

  Agrega la instancia de "vecino" a las conexiones de "habitacion" y viceversa, si es que no se encuentra ya en la lista.
- Modificar `def descartar_habitacion(self, habitacion) -> None`: Elimina la instancia de `Habitacion` de las conexiones de todos sus vecinos.

Parte 2

Luego de encontrar tu supertraje, es momento de salir a brillar. Todos saben que las casas con los mejores dulces son también aquellas con el mejor espíritu festivo, es por esto que decides ir a la casa con la mejor decoración, aunque preferirías no caminar tanto ~~por que eres muy flojo~~ para no ensuciar tu traje.

En esta parte, deberás encontrar la casa que mejor se adecue a tus preferencias, para lo cual posees un `Mapa` que representa perfectamente el `Condominio` en el que vives. Particularmente, este posee la forma de un árbol binario 🤯 como se describe a continuación:

- El vecindario se puede acceder sólo a través de una casa en la esquina del vecindario, que representa la raíz.
- Cada casa puede conectar con hasta 2 otras casas, que representan a los hijos de un nodo.
- Cada casa se puede acceder a través de sólo una casa (nodo padre), a excepción de la casa de la esquina (raíz)

Archivos:

- condominio.py: Se encarga de definir las clases **Condominio** y **Casa**.
- protagonista_disfrazado.py: Se encarga de definir las clases **ProtagonistaDisfrazado**.

Clases

- No modificar `class Casa`: Clase que representa a un nodo del árbol binario.
  - No modificar `def __init__(self, id:int, id_padre:int,distancia:int, nivel_decoracion:int, posicion:int) -> None:`

    Inicializa cada casa con los siguientes atributos:
    - `self.id`: id del nodo dentro del árbol.

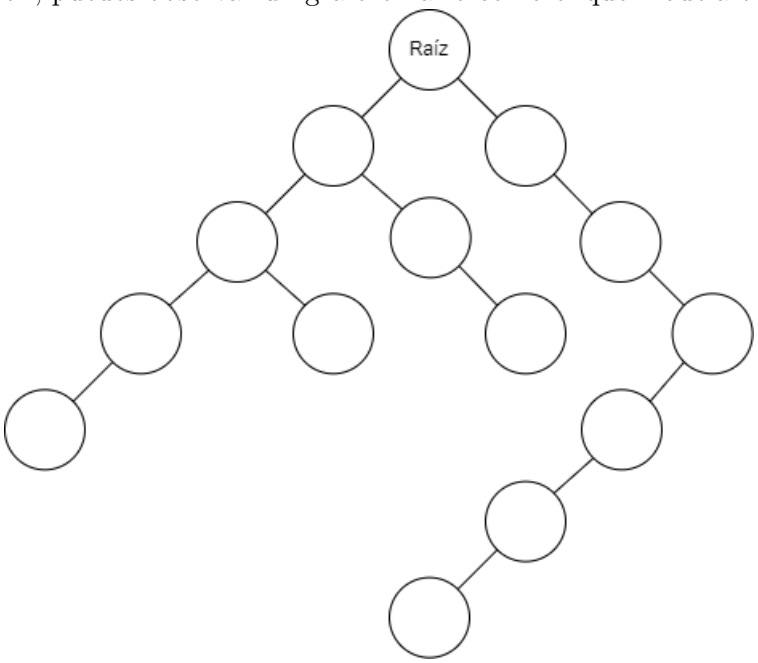
- `self.id_padre`: id del padre de esta casa, es decir, la casa en la que está el protagonista antes de realizar el trayecto.
- `self.decoracion`: valor representativo de que tan decorada se encuentra la casa considerando que es Halloween.
- `self.distancia`: valor que representa la distancia desde el nodo **padre** hasta **este** nodo.
- `self.hijo_izquierdo`: instancia de `Casa` ubicada "abajo" a la izquierda de esta casa. Es `None` si es que no existe.
- `self.hijo_derecho`: instancia de `Casa` ubicada "abajo" a la derecha de esta casa. Es `None` si es que no existe.
- `self.posicion`: representa si este nodo es un hijo izquierdo(0) o derecho(1) de la casa padre.

- No modificar `def actualizar_cola(self, protagonista) -> None`: Genera el flujo y/o movimiento de la cola, con el fin de que eventualmente atiendan al protagonista. Mientras atienden al protagonista existe la posibilidad que personajes se cuelen a la fila.

- No modificar `class Condominio`: Clase que representa al árbol Binario.
  - No modificar `def __init__(self, dict_casas: dict) -> None`: Inicializa el condominio con los siguientes atributos:
    - `self.casa_protagonista`: Inicia con `None`, pero una vez que se identifica al protagonista, corresponde a la instancia de `Casa` del protagonista, la cual corresponde a la **raíz** del árbol.
    - `self.dict_casas`: diccionario que contiene toda la información de las casas a recorrer.
    - `self.poblado`: `False` si el árbol no está poblado, `True` una vez que ya este poblado.
    - `self.dict_casas_final`: diccionario que tiene como key el id de la casa y la instancia de Casa como valor.
    - `self.protagonista`: `False` si el protagonista, junto con su casa, no ha sido identificado, `True` si ya fue identificado.
  - No modificar `def identificar_protagonista(self):-> None`: Identifica al protagonista, quien tendrá id = 0. Inicializa su instancia de Casa y la guarda en `self.casa_protagonista`. Una vez identificado, `self.protagonista = True`.
  - No modificar `def poblar_condominio(self):-> None`: Para poblar el condominio, el protagonista y su casa debe haber sido identificada. Hace la inicialización de cada Casa, con sus respectivos hijos. Utiliza `self.dict_casas` para tener la información para la inicialización.
  - No modificar `def asignar_ubicacion(self, nodo_padre: Casa, nodo_hijo:Casa):-> None`: Dependiendo de `self.posicion` del `nodo_hijo`, su instancia se guarda en la instancia de `nodo_padre`, en `nodo_padre.hijo_derecho` como hijo derecho o en `nodo_padre.hijo_izquierdo` como hijo izquierdo.
  - No modificar `def mostrar_arbol(self):-> None`: El árbol ya debe estar poblado para mostrarlo. Un nodo podrá tener ningún nodo hijo, un nodo hijo o dos nodos hijos. Para mostrar el árbol, deberás fijarte que sus nodos hijos sean distintos de `None`. Para cada nodo deberás imprimir su id y el de sus hijos que sean distintos de `None`. Puedes basarte en el siguiente código (que es para un nodo padre con ambos hijos distintos de `None`):

```python
print(f'''
Casa de id:{casa.id}
--> hijo izquierdo Casa de id:{casa.hijo_izquierdo.id}
--> hijo derecho Casa de id:{casa.hijo_derecho.id}
''')
```

A continuación, puedes observar un grafo binario como el que modela el condominio



PD: el condominio **no** es exactamente igual a este grafo, este es sólo una referencia.

- Modificar `class ProtagonistaDisfrazado`: Clase que representa al protagonista durante la segunda parte de la actividad, este iterará por todas las casas registrándolas junto a su distancia respecto a su casa (raíz) para luego filtrar estas y elegir la mejor.
  - No modificar `def __init__(self, nombre: str)`: Inicializa al protagonista con los siguientes atributos:
    - `self.nombre`: guarda el nombre del protagonista.
  - Modificar `def recuperar_casas_x`: Deberás definir **una** de las siguientes funciones que almacene en una **lista**, todas las casas del condominio junto con la distancia acumulada desde la casa del protagonista(raíz). Cada elemento de la **lista** debe ser un **diccionario** de la forma: `{"casa": casa_actual, "distancia": distancia_actual}`

    Donde: `casa_actual` es una instancia de la clase `Casa` y `distancia_actual` es un `int` con la distancia acumulada.

    Cualquiera de las 3 funciones escogidas que se describirán a continuación deberá retornar la lista creada siguiendo las instrucciones anteriores.

- `def recuperar_casas_DFS_recursivo(self, casa_actual, distancia_actual: int) -> list:`

  Deberás implementar un algoritmo de búsqueda del tipo **DFS** que cumpla con lo pedido anteriormente de forma **recursiva**

- `def recuperar_casas_DFS_iterativo(self, casa_inicial) -> list`: Deberás implementar un algoritmo de búsqueda del tipo **DFS** que cumpla con lo pedido anteriormente de forma **iterativa**.

- `def recuperar_casas_BFS_iterativo(self, casa_inicial) -> list`: Deberás implementar un algoritmo de búsqueda del tipo **BFS** que cumpla con lo pedido anteriormente de forma **iterativa**.

- No modificar `def filtrar_casas(self, listado_casas) -> list`: Recibe el listado de diccionarios, donde cada diccionario tiene una instancia de `Casa` y la distancia desde la raíz hasta el nodo en cuestión, manteniendo solo aquellos que cumplen con ser mayor que `EXPECTATIVAS_DECORACION` para añadir a una tupla esta instancia de clase y un puntaje asignado con la fórmula:

  `puntaje = PONDERADOR_DECORACION*casa.decoracion-PONDERADOR_DISTANCIA*distancia_casa`

  Para finalmente retornar una lista de tuplas como la descrita anteriormente.

- No modificar `def elegir_mejor_casa(self, lista_casas)`: Esta función itera por la lista generada en la función anterior, de forma que retorna la casa con mayor puntaje.

Parte 3

Siendo una persona precavida decides prepararte para lo peor. Sabiendo que no puedes ser el único en busca de los mejores dulces, piensas que probablemente ya exista una cola de personas esperando su oportunidad para sacar su parte antes de que sea demasiado tarde.

Deberás implementar los métodos de una **Lista Ligada** para poder modelar el correcto funcionamiento de una fila y poder predecir el tiempo que tendrás que esperar antes de llegar adelante.

Archivos:

- cola_dulces.py: Se encarga de definir las clases **TrickOrTreater** y **ColaDulces**.
- casa.py: Se encarga de definir la clase **DCCasa**.

Clases

- No modificar `class DCCasa`: Clase que contiene a la cola y simula el paso del protagonista por esta.
  - No modificar `def __init__(self, cola: ColaDulce) -> None`:
    Inicializa la cola con los siguientes atributos:
    - `self.cola`: lista ligada que representa la cola actual de la casa.
    - `self.colones`: lista con los nombres de las personas que posiblemente se podrían colar en la fila.
  - No modificar `def simular_cola(self) -> None`:
    Simula el funcionamiento de la cola a través de un ciclo, haciendo avanzar una persona por

cada iteración.

- No modificar `class TrickOrTreater`: Clase abreviada como tot. Representa a una persona esperando a recibir dulces en la cola.
  - No modificar `def __init__(self, nombre:str)`: Inicializador de la la clase.
    - `self.nombre`: nombre asignado del tot.
    - `self.protagonista`: `True` si este tot es el protagonista `False` en otro caso,
    - `self.siguiente`: instancia del tot que le sigue en la cola.
- Modificar `class ColaDulces`: Lista ligada que representa una cola de personas.
  - No modificar `def __init__(self) -> None`:
    Inicializador de la clase. -
    - `self.primero`: instancia del primer tot en la cola, se inicia como `None`.
    - `self.ultimo`: instancia del último tot en la cola, se inicia como `None`.
  - No modificar `def tot_llega(self, nombre) -> None`:
    Agrega una persona al final de la cola.
  - No modificar `def obtener_tot(self, posicion) -> TrickOrTreater`:
    Recibe una posición como argumento y retorna el tot en esa posición de la cola.
  - Modificar `def obtener_posicion_protagonista(self) -> int`:
    Busca al protagonista en la fila para finalmente retornar su posición en la cola. Si no se encuentra en la fila, retorna -1.
  - Modificar `def tot_se_cola(self, nombre, posicion) -> None`:
    Crea un `TrickOrTreater` con el nombre recibido y lo inserta en la cola en la posición recibida. Recuerda que considerar los casos especiales y que si deseas añadirlo en la posición `posicion`, debe ubicarse **después** del tot en la posición `posicion-1`.
  - No modificar `def tot_se_va(self, posicion) -> None`:
    Elimina a la persona de la posición recibida de la cola.
  - Modificar `def atender_tot(self) -> TrickOrTreater`:
    Elimina a la primera persona de la cola, define al nuevo primero y finalmente retorna al tot que se encontraba primero en la lista.
  - Modificar `def obtener_largo(self) -> int`:
    Obtiene el largo de la cola(cantidad de personas desde el primero al último) para luego retornarlo.
  - No modificar `def __str__(self) -> str`:
    Retorna una representación de la fila con los nombres de cada uno de sus participantes.

Requerimientos

- (2.00 pts) Parte 1:
  - (1.00 pts) define correctamente `crear_habitacion`
  - (0.50 pts) define correctamente `registrar_vecino`

  - (0.50 pts) define correctamente `descartar_habitacion`
- (2.00 pts) Parte 2:
  - (2.00 pts) define correctamente `recuperar_casas_x`
- (2.00 pts) Parte 3:
  - (0.50 pts) define correctamente `obtener_posicion_protagonista`
  - (0.50 pts) define correctamente `tot_se_cola`
  - (0.50 pts) define correctamente `atender_tot`
  - (0.50 pts) define correctamente `obtener_largo`