**Análisis de Algoritmos**

**Clase Paciente:**

**La clase Paciente gestiona la información de un paciente, incluyendo su historial de enfermedades, alergias, y métodos para actualizar y mostrar los atributos. A continuación se describen las operaciones más relevantes.**

**Métodos Principales de la Clase Paciente:**

* **Inicialización (\_\_init\_\_):**
  + **Tiempo: O(1). La inicialización de los atributos se hace en tiempo constante, ya que se asignan valores fijos a los atributos.**
  + **Espacio: O(1). Solo se almacenan los atributos proporcionados.**
* **Método \_\_str\_\_:**
  + **Tiempo: O(k), donde k es el número de elementos en el historial del paciente y las alergias. En el peor caso, si ambos atributos contienen muchos elementos, la complejidad sería proporcional a la cantidad de información a imprimir.**
  + **Espacio: O(k). Se genera una cadena que contiene toda la información del paciente, lo cual consume espacio proporcional a la longitud de la cadena.**
* **Métodos de Actualización (e.g., actualizar\_nombres, actualizar\_edad, etc.):**
  + **Tiempo: O(1) en la mayoría de los casos. Los métodos de actualización solo comparan y asignan nuevos valores si es necesario.**
  + **Espacio: O(1), ya que se almacenan datos individuales.**
* **Método agregar\_enfermedad:**
  + **Tiempo: O(1). La operación de agregar una enfermedad al historial se realiza de forma constante, ya que solo se inserta un par clave-valor en el diccionario.**
  + **Espacio: O(1) para cada nueva entrada, ya que solo se agregan elementos al diccionario.**
* **Métodos de Búsqueda y Eliminación (e.g., eliminar\_enfermedad, mostrar\_historial\_paciente):**
  + **Tiempo: O(n), donde n es el número de enfermedades en el historial. Las búsquedas, como mostrar\_historial\_paciente, requieren iterar sobre el historial del paciente.**
  + **Espacio: O(n) en el peor caso, ya que se crea una lista con las enfermedades del historial que deben ser almacenadas temporalmente.**

**Función busqueda\_enfermedad\_recurrente:**

**La función busqueda\_enfermedad\_recurrente es una búsqueda recursiva que encuentra todas las ocurrencias de una clave (enfermedad o medicamento) en el historial del paciente.**

**Análisis de la función busqueda\_enfermedad\_recurrente:**

* **Tiempo:**
  + **La función recursiva visita cada par clave-valor del historial una vez, lo que da como resultado una complejidad de O(n), donde n es el número de enfermedades en el historial. En cada paso, la función compara la clave con el nombre de la enfermedad y el medicamento, lo que requiere tiempo constante.**
  + **La función recursiva termina cuando todos los elementos han sido procesados.**
* **Espacio:**
  + **O(n) en el peor caso, debido a la pila de recursión utilizada. Cada llamada recursiva consume espacio en la pila, y el número máximo de llamadas recursivas es igual a la cantidad de elementos en el historial.**
  + **La lista resultado también puede llegar a tener un tamaño proporcional al número de coincidencias encontradas.**

**Resumen de Complejidad de Tiempo y Espacio para la Clase Paciente y la Función busqueda\_enfermedad\_recurrente:**

| **Operación** | **Complejidad en Tiempo** | **Complejidad en Espacio** |
| --- | --- | --- |
| **Inicialización del paciente** | **O(1)** | **O(1)** |
| **Método \_\_str\_\_ (imprimir paciente)** | **O(k) (k: historial + alergias)** | **O(k)** |
| **Método agregar\_enfermedad** | **O(1)** | **O(1)** |
| **Método mostrar\_historial\_paciente** | **O(n) (n: número de enfermedades)** | **O(n)** |
| **Método eliminar\_enfermedad** | **O(1)** | **O(1)** |
| **Función busqueda\_enfermedad\_recurrente** | **O(n)** | **O(n) (debido a recursión y almacenamiento temporal de resultados)** |

**Impacto en el Rendimiento:**

* **Uso de Memoria:**
  + **La estructura historial\_paciente es un diccionario, lo que permite una búsqueda eficiente en O(1) para cada operación de lectura o escritura. Sin embargo, si el historial se vuelve muy grande, el consumo de memoria también aumenta.**
  + **La pila de recursión en la función busqueda\_enfermedad\_recurrente es lineal en el peor caso, lo que podría convertirse en un problema si se tienen muchos elementos en el historial.**
  + **Los métodos que generan cadenas (como \_\_str\_\_) también pueden ser costosos en términos de espacio si se imprimen muchos detalles del paciente.**
* **Rendimiento:**
  + **Las operaciones de actualización y adición de enfermedades tienen un rendimiento constante O(1) en la mayoría de los casos.**
  + **Las operaciones de búsqueda y eliminación de enfermedades tienen un rendimiento lineal O(n), lo que es adecuado para un número moderado de enfermedades.**
  + **La función recursiva de búsqueda es eficiente para un número relativamente pequeño de elementos, pero en grandes historiales podría volverse costosa debido a la recursión.**

**Árbol Binario de Búsqueda (ABB)**

**Insertar un Paciente:**

* **Método utilizado**: \_insertar\_recursivo(nodo, valor).
* **Complejidad en tiempo**: En un árbol binario de búsqueda **balanceado**, la complejidad de inserción es **O(log n)**, donde **n** es el número de nodos en el árbol. Esto es porque en cada paso se descarta la mitad de los nodos, ya sea a la izquierda o a la derecha del árbol.
  + En el peor caso, si el árbol está desbalanceado (esencialmente una lista enlazada), la complejidad de la inserción sería **O(n)**, ya que el árbol degeneraría a una estructura lineal.
* **Complejidad en espacio**: La complejidad espacial es **O(n)**, porque en el peor caso, el árbol podría necesitar almacenar todos los pacientes.

**Buscar un Paciente:**

* **Método utilizado**: \_buscar\_recursivo(nodo, dni).
* **Complejidad en tiempo**: Similar a la inserción, la complejidad de búsqueda es **O(log n)** en el caso promedio, si el árbol está balanceado. En el peor caso, **O(n)**, si el árbol está desbalanceado.
* **Complejidad en espacio**: La complejidad espacial es **O(h)**, donde **h** es la altura del árbol. En el peor caso, **O(n)** (cuando el árbol es degenerado).

**Eliminar un Paciente:**

* **Método utilizado**: \_eliminar\_recursivo(nodo, dni).
* **Complejidad en tiempo**: La eliminación de un nodo también sigue una estrategia de búsqueda, por lo que la complejidad es **O(log n)** en el caso promedio, y **O(n)** en el peor caso (si el árbol está desbalanceado). Después de encontrar el nodo, si tiene dos hijos, se requiere encontrar el mínimo en el subárbol derecho, lo que añade una complejidad de **O(h)** para encontrar el mínimo.
* **Complejidad en espacio**: **O(h)**, similar a la búsqueda.

**Imprimir el árbol (in-order):**

* **Método utilizado**: \_imprimir\_inorder\_recursivo(nodo).
* **Complejidad en tiempo**: **O(n)**, porque debemos recorrer todos los nodos en el árbol, una vez en el caso promedio y peor.
* **Complejidad en espacio**: **O(h)**, ya que la recursión utiliza espacio de pila proporcional a la altura del árbol.

**Mostrar historial de un paciente:**

* **Método utilizado**: mostrar\_historial(dni).
* **Complejidad en tiempo**: La complejidad de búsqueda de un paciente es **O(log n)** en el caso promedio, y **O(n)** en el peor caso, y luego mostrar el historial es **O(1)** si la operación de búsqueda fue exitosa.
* **Complejidad en espacio**: **O(1)** adicional para mostrar el historial.

**Listar pacientes por localidad:**

* **Método utilizado**: \_listar\_por\_localidad\_recursivo(nodo, localidad, pacientes).
* **Complejidad en tiempo**: **O(n)**, porque en el peor caso necesitamos recorrer todo el árbol para encontrar todos los pacientes de una localidad.
* **Complejidad en espacio**: **O(n)** para almacenar todos los pacientes que coincidan con la localidad.

**Cola de Prioridad (Usando heapq):**

**Insertar un Paciente:**

* **Método utilizado**: heapq.heappush(self.cola, (paciente.prioridad, paciente)).
* **Complejidad en tiempo**: La inserción en una cola de prioridad basada en un heap es **O(log n)**, donde **n** es el número de elementos en la cola.
* **Complejidad en espacio**: **O(n)**, ya que estamos almacenando los pacientes en el heap.

**Extraer un Paciente:**

* **Método utilizado**: heapq.heappop(self.cola).
* **Complejidad en tiempo**: La extracción de un elemento de la cola de prioridad también tiene una complejidad de **O(log n)**, ya que es necesario reorganizar el heap después de la extracción.
* **Complejidad en espacio**: **O(n)** para almacenar los pacientes en el heap.

**Mostrar los pacientes:**

* **Método utilizado**: mostrar(self).
* **Complejidad en tiempo**: **O(n)**, porque iteramos sobre todos los pacientes en la cola.
* **Complejidad en espacio**: **O(n)**, ya que no se requiere espacio adicional aparte del almacenaje en el heap.

**Verificación si la cola está vacía:**

* **Método utilizado**: esta\_vacia().
* **Complejidad en tiempo**: **O(1)**, ya que solo se verifica el tamaño de la cola.
* **Complejidad en espacio**: **O(1)**.

**Resumen de Complejidad de Tiempo y Espacio**

| **Operación** | **Complejidad en tiempo** | **Complejidad en espacio** |
| --- | --- | --- |
| **Insertar en ABB** | **O(log n)** (promedio), **O(n)** (peor caso) | **O(n)** |
| **Buscar en ABB** | **O(log n)** (promedio), **O(n)** (peor caso) | **O(h)** |
| **Eliminar en ABB** | **O(log n)** (promedio), **O(n)** (peor caso) | **O(h)** |
| **Imprimir ABB en orden** | **O(n)** | **O(h)** |
| **Mostrar historial en ABB** | **O(log n)** (buscar) + **O(1)** (mostrar) | **O(1)** |
| **Listar pacientes por localidad** | **O(n)** | **O(n)** |
| **Insertar en Cola de Prioridad** | **O(log n)** | **O(n)** |
| **Extraer de Cola de Prioridad** | **O(log n)** | **O(n)** |
| **Mostrar Cola de Prioridad** | **O(n)** | **O(n)** |
| **Verificar si Cola está vacía** | **O(1)** | **O(1)** |

**Conclusión**

* **En términos de eficiencia en tiempo**, la implementación es bastante eficiente para la mayoría de las operaciones, con **O(log n)** en promedio para inserciones, búsquedas y extracciones de la cola de prioridad. Las operaciones más costosas en el peor caso ocurren cuando el árbol binario de búsqueda está desbalanceado, lo que puede hacer que la complejidad sea **O(n)** en lugar de **O(log n)**.
* **En términos de eficiencia en espacio**, tanto el árbol binario de búsqueda como la cola de prioridad requieren **O(n)** espacio, lo cual es bastante razonable dada la necesidad de almacenar hasta **n** pacientes.

**Clase Grafo**

La clase Grafo tiene métodos para agregar vértices y aristas, así como para mostrar el grafo. Es importante entender que los vértices y las aristas son almacenados en un diccionario de listas (un **grafo representado por adyacencia**).

**Métodos de la clase Grafo:**

**Agregar vértice**:

* + **Complejidad en tiempo**: **O(1)**, ya que los diccionarios en Python tienen un acceso directo por clave.
  + **Complejidad en espacio**: **O(1)** para agregar un vértice, aunque en el caso general, el espacio utilizado será **O(n)**, donde **n** es el número de vértices en el grafo.

**Agregar arista**:

* + **Complejidad en tiempo**: **O(1)** para insertar una arista en las listas de adyacencia de cada vértice.
  + **Complejidad en espacio**: **O(1)** por cada arista, ya que se está agregando un par de elementos (vértice destino, peso) a la lista de adyacencias de un vértice.

**Mostrar grafo**:

* + **Complejidad en tiempo**: **O(n + m)**, donde **n** es el número de vértices y **m** es el número de aristas, ya que tenemos que recorrer todas las conexiones de cada vértice.
  + **Complejidad en espacio**: **O(n + m)** para almacenar el grafo en memoria.

**Recorridos y Algoritmos**

**DFS (Búsqueda en profundidad)**

El algoritmo **DFS (Depth First Search)** es utilizado para encontrar un camino entre dos vértices en el grafo.

* **Método utilizado**: dfs\_camino(grafo, inicio, objetivo, visitados=None, camino=None)
  + **Complejidad en tiempo**: **O(n + m)**, donde **n** es el número de vértices y **m** el número de aristas. Esto se debe a que en el peor caso, DFS explora todos los vértices y aristas del grafo (cada vértice y cada arista será visitado al menos una vez).
  + **Complejidad en espacio**: **O(n)**, porque la recursión puede llegar a tener una profundidad de **n** en el peor caso (en un grafo muy desbalanceado o en forma de lista). También se usa espacio adicional para almacenar los vértices visitados y el camino recorrido.

**BFS (Búsqueda en anchura)**

El algoritmo **BFS (Breadth First Search)** se utiliza para encontrar el camino más corto en un grafo no ponderado entre dos vértices.

**Método utilizado**: bfs\_camino\_mas\_corto(grafo, inicio, objetivo)

* + **Complejidad en tiempo**: **O(n + m)**, donde **n** es el número de vértices y **m** el número de aristas. Al igual que en DFS, BFS recorre todo el grafo, pero en este caso explorando los vértices por niveles.
  + **Complejidad en espacio**: **O(n)**, debido a la cola utilizada para almacenar los vértices a explorar y el conjunto de vértices visitados.

**Ordenamiento Topológico**

El ordenamiento topológico se utiliza en grafos dirigidos acíclicos (DAGs) para ordenar los vértices de manera que para cada arista dirigida **u -> v**, **u** aparece antes que **v** en el orden.

* **Método utilizado**: ordenamiento\_topologico(grafo)
  + **Complejidad en tiempo**: **O(n + m)**, ya que se realiza un DFS sobre cada vértice y sus aristas. El recorrido sobre el grafo se realiza una vez por vértice y por arista.
  + **Complejidad en espacio**: **O(n)** para almacenar el estado de los vértices (visitados y en la pila).

**Dijkstra (Camino mínimo ponderado)**

El algoritmo de **Dijkstra** se utiliza para encontrar el camino más corto desde un vértice de inicio a todos los demás vértices en un grafo ponderado.

* **Método utilizado**: dijkstra(grafo, inicio)
  + **Complejidad en tiempo**: **O((n + m) log n)**, donde **n** es el número de vértices y **m** el número de aristas. La complejidad de tiempo se debe al uso de una cola de prioridad (implementada con heapq), que permite extraer el vértice con la menor distancia en **O(log n)** y actualizar las distancias en **O(log n)** por cada vértice adyacente.
  + **Complejidad en espacio**: **O(n + m)** para almacenar el grafo y la cola de prioridad, además de los valores de distancia para cada vértice.

**Resumen de Complejidad de Tiempo y Espacio**

| **Operación** | **Complejidad en tiempo** | **Complejidad en espacio** |
| --- | --- | --- |
| **Agregar vértice** | **O(1)** | **O(1)** |
| **Agregar arista** | **O(1)** | **O(1)** |
| **Mostrar grafo** | **O(n + m)** | **O(n + m)** |
| **DFS (Búsqueda en profundidad)** | **O(n + m)** | **O(n)** |
| **BFS (Búsqueda en anchura)** | **O(n + m)** | **O(n)** |
| **Ordenamiento Topológico** | **O(n + m)** | **O(n)** |
| **Dijkstra (Camino mínimo ponderado)** | **O((n + m) log n)** | **O(n + m)** |

**Conclusión**

* **En términos de tiempo**, los algoritmos más costosos son **Dijkstra** (con complejidad **O((n + m) log n)**) debido al uso de una cola de prioridad, y los algoritmos de recorrido (**DFS** y **BFS**), que tienen complejidad **O(n + m)**, lo que es eficiente para grafos dispersos.
* **En términos de espacio**, los algoritmos y estructuras de datos requieren **O(n)** espacio para almacenar el estado de los vértices, y **O(m)** espacio adicional para almacenar las aristas del grafo.

Arbol general:

**Clase Nodo**

La clase Nodo es bastante simple, ya que cada nodo solo almacena un valor y una lista de hijos.

**Métodos de la clase Nodo:**

* **Inicialización del nodo**:
  + **Complejidad en tiempo**: **O(1)**. La operación de crear un nodo solo involucra la asignación de valores a las propiedades valor y hijos.
  + **Complejidad en espacio**: **O(1)**. Cada nodo tiene espacio para almacenar su valor y una lista de hijos.
* **Método agregar\_hijo**:
  + **Complejidad en tiempo**: **O(1)**. Solo se agrega un hijo a la lista de hijos del nodo.
  + **Complejidad en espacio**: **O(1)**. Solo se agrega un elemento a la lista de hijos del nodo.
* **Método \_\_repr\_\_**:
  + **Complejidad en tiempo**: **O(1)**. La representación en cadena del nodo es una operación de acceso a un atributo y concatenación de cadenas.
  + **Complejidad en espacio**: **O(1)**, ya que se genera una cadena de longitud constante.

**2. Clase Arbol**

La clase Arbol representa un árbol jerárquico de nodos, y tiene métodos para agregar nodos hijos y mostrar el árbol de manera jerárquica.

**Métodos de la clase Arbol:**

* **Método agregar\_hijo\_a\_nodo**:
  + **Complejidad en tiempo**: **O(1)**. Este método solo agrega un hijo a un nodo, lo que es una operación constante si la lista de hijos del nodo es accesible.
  + **Complejidad en espacio**: **O(1)**, ya que se agrega solo un elemento a la lista de hijos de un nodo.
* **Método mostrar\_arbol** (recursivo):
  + **Complejidad en tiempo**: **O(n)**, donde **n** es el número total de nodos en el árbol. Esto es porque el algoritmo recursivo visita todos los nodos del árbol una vez.
  + **Complejidad en espacio**: **O(n)** en el peor caso debido a la pila de recursión. Esto se debe a que cada llamada recursiva almacena información de los nodos hasta que el árbol se recorre completamente.

**3. Análisis de Complejidad en Tiempo y Espacio**

En este código, tenemos dos árboles representados jerárquicamente: uno para el paciente1 y otro para el paciente2. A continuación, desglosamos las complejidades de los métodos y operaciones principales:

**Operaciones con los Nodos:**

* **Agregar un nodo hijo**:
  + **Tiempo**: **O(1)**, ya que solo estamos agregando un hijo a una lista de hijos de un nodo específico.
  + **Espacio**: **O(1)**, ya que solo estamos agregando un nuevo nodo a la lista de hijos.
* **Mostrar el árbol**:
  + **Tiempo**: **O(n)**, ya que necesitamos visitar todos los nodos en el árbol y mostrarlos. Este recorrido es recursivo, por lo que la complejidad de tiempo es lineal respecto al número de nodos.
  + **Espacio**: **O(n)** debido a la pila de recursión utilizada para recorrer el árbol, donde **n** es el número de nodos. La cantidad de memoria ocupada en la pila de recursión es proporcional a la profundidad del árbol.

**Creación de arbol para Paciente 1:**

En este caso, se crea un árbol para el paciente1 con la siguiente estructura:

* Raíz: **Paciente1**
  + Visitas:
    - **Visita 1**: Contiene:
      * **Diagnóstico**: Gripe
      * **Tratamiento**: Tarifol cada 6 hs
    - **Visita 2**: Contiene:
      * **Diagnóstico**: Diarrea
      * **Tratamiento**: Ibuprofeno cada 6 hs

**Número de nodos**: En total, para este paciente, tenemos los siguientes nodos:

* 1 nodo raíz (Paciente1).
* 2 nodos de visita.
* 2 nodos de diagnóstico.
* 2 nodos de tratamiento.

Total de nodos: 7.

**Complejidad de mostrar el árbol para paciente1**:

* **Tiempo**: **O(7)**.
* **Espacio**: **O(7)**.

**Creación de arbol para Paciente 2:**

El árbol para el paciente2 sigue una estructura similar:

* Raíz: **Paciente2**
  + Visitas:
    - **Visita 1**: Contiene:
      * **Diagnóstico**: Dolor Lumbar
      * **Tratamiento**: Diclofenac
    - **Visita 2**: Contiene:
      * **Diagnóstico**: Dolor de muela
      * **Tratamiento**: Keterolac

**Número de nodos**: Similar al caso anterior:

* 1 nodo raíz (Paciente2).
* 2 nodos de visita.
* 2 nodos de diagnóstico.
* 2 nodos de tratamiento.

Total de nodos: 7.

**Complejidad de mostrar el árbol para paciente2**:

* **Tiempo**: **O(7)**.
* **Espacio**: **O(7)**.

**Resumen de Complejidad de Tiempo y Espacio**

| **Operación** | **Complejidad en tiempo** | **Complejidad en espacio** |
| --- | --- | --- |
| **Agregar un nodo hijo** | **O(1)** | **O(1)** |
| **Mostrar árbol (recursivo)** | **O(n)** | **O(n)** |

**Conclusión**

* **Tiempo**: Las operaciones de agregar un hijo a un nodo son constantes (**O(1)**), y mostrar el árbol tiene una complejidad lineal respecto al número de nodos (**O(n)**), donde **n** es el número total de nodos en el árbol.
* **Espacio**: El espacio requerido es también lineal en cuanto al número de nodos (**O(n)**), ya que en el peor caso, la pila de recursión para mostrar el árbol podría almacenar todos los nodos si el árbol es muy profundo.