Facultad de Ingeniería
UNER

Trabajo Práctico Nº 2

1. Sala de emergencias

Para el problema planteado en la primera consigna decidimos trabajar con Montículos Binarios(Montículo mínimo (Min Heap)), ya que es de gran utilidad para implementar una cola de prioridad. Presenta gran eficiencia en la inserción y extracción de elementos, rápida selección del paciente más crítico, porque el montículo ordena automáticamente considerando los niveles de riesgo, dejando al paciente más crítico como la raíz del mismo. También permite considerar el tiempo de llegada de cada paciente por lo que sí hay dos o más pacientes con el mismo nivel de atención será atendido el primero en llegar. Por último, esta estructura se puede implementar con cualquier tipo de datos.

El orden de complejidad O de la operación de inserción es de O(log n), ya que primero se inserta el elemento en el montículo binario en el final de la lista, después de una serie de comparaciones e intercambios con sus padres, si es necesario, se coloca en la posición correspondiente. En cada comparación e intercambio se reduce la distancia a la mitad entre el nuevo elemento y la posición final del mismo. Debido a que se requieren estos pasos para que el elemento se mueva desde la hoja hasta su posición final, la posición final será logarítmica (en base 2) en relación al número total de elementos dentro del montículo.

Por otro lado, el orden de complejidad O de la operación de eliminación (del elemento mínimo) implementada también es de O(log n), donde se reemplaza la raíz del montículo con el último elemento, se realizan comparaciones e intercambios con sus hijos hasta que se coloca en la posición correspondiente. Se reduce la distancia a la mitad entre el elemento y su posición final, por lo mismo que ocurre en la operación de inserción.

En ambos casos, “n” dependerá del número de elementos en el montículo.

Implementamos la clase Paciente, donde sus atributos son:

- self.__nombre
- self.__apellido
- self.__riesgo
- self.__descripcion

Y sus métodos:

- get_nombre
- get_apellido
- get_riesgo
- get_descripcion_riesgo
- __str__

La clase MonticuloBinario, sus atributos son:

- self.listaMonticulo
- self.tamanoActual

Y sus métodos:

- infiltArriba

- insertar
- infiltAbajo
- hijoMin
- eliminarMin
- construirMonticulo

2. Temperaturas DB

En este problema utilizamos Árbol Binario de Búsqueda. Los métodos y clases utilizados fueron:

NodoABB: es la clase que define un nodo en el árbol. Cada nodo tiene una clave, valor y referencias a sus hijos izquierdo y derecho. Los métodos son:

- __init__: O(1). Método que solo inicializa valores.
- esHoja
- tieneHijoIzquierdo
- tieneHijoDerecho

Para los métodos esHoja, tieneHijoIzquierdo, y tieneHijoDerecho el orden de complejidad es O(1), ya que solo verifican si ciertos punteros son None. Complejidad de tiempo constante.

La clase ABB es la que define el árbol binario de búsqueda. Los métodos son:

- __len__: devuelve el tamaño del árbol, es decir que su complejidad de tiempo es constante, O(1).
- agregar y _agregar: estos métodos insertan un nodo en el árbol. En el peor caso pueden tener que recorrer todo el camino desde la raíz hasta una hoja, complejidad O(n). En un ABB equilibrado, la cantidad de nodos en este camino es logarítmica en el número de nodos en el árbol, por lo que el orden de complejidad es O(log n).
- eliminar, _encontrar_sucesor, _eliminar: estos métodos eliminan un nodo del árbol. Como la inserción, el peor caso puede tener que recorrer todo el camino desde la raíz hasta una hoja, por lo que el orden de complejidad será O(log n).
- obtener y _obtener: estos métodos buscan un nodo en el árbol por su clave, su orden de complejidad será de O(log n). El peor caso ocurre como en los métodos anteriores, recorrer todo el árbol.
- __contains__: verifica si una clave está en el árbol. Utiliza el método “_obtener”, por lo que su orden de complejidad es la misma, O(log n).
- __getitem__ y __setitem__: estos métodos permiten acceder y modificar los valores en el árbol. Utilizan los métodos “obtener” y “agregar”, por lo que su orden de complejidad será O(log n).
- __delitem__: permite eliminar nodos del árbol usando el operador del. Utiliza el método “eliminar”, por lo que su orden de complejidad será O(log n).
- __iter__ y _inorden: estos métodos permiten iterar sobre las claves y los valores en el árbol en orden ascendente. Tienen que visitar todos los nodos en el árbol una vez, por ello su orden de complejidad será O(n).
- guardar_temperatura, devolver_temperatura, max_temp_rango, min_temp_rango, temp_extremos_rango, borrar_temperatura, devolver_temperaturas: estos métodos son específicos para un uso del árbol

como un registro de temperaturas. Utiliza los métodos “agregar”, “obtener” y “eliminar”, por lo que su orden de complejidad será O(log n).

- _in_rango y _in_rango_rec: su orden de complejidad, es O(n) en el peor de los casos, ya que estos métodos auxiliares permiten iterar sobre las claves y los valores en el árbol, por que deben visitar todos los nodos en el rango una vez.

3. Servicio de Transporte

En este problema implementamos el algoritmo de Dijkstra. Elegimos este algoritmo, a pesar de que por lo general se utiliza para obtener el camino más corto, también se puede modificar para obtener el máximo cuello de botella, tomando como punto de partida un nodo inicial que en nuestro caso sería el llamado “CiudadBs.As.”, y como nodo final cualquiera de las otras ciudades que se encuentran dentro del grafo creado a partir del archivo de texto brindado por la cátedra, obteniendo el peso máximo en cada caso. De la misma forma, usamos este algoritmo para obtener el costo mínimo de cada recorrido.

La clase ColaPrioridad tiene como atributos:

- self.cola
- self.indice: mantiene un orden entre los elementos.

Sus métodos son:

- insertar
- obtener
- esta_vacia
- decrementar_clave

En clase Vertice, para cada vértice tiene una clave, un valor o carga útil, referencias a sus vecinos y atributos para almacenar su distancia desde un vértice inicial y su vértice predecesor en un camino desde el vértice inicial. Los métodos de esta clase son:

- agregarVecino
- asignarDistancia
- asignarPredecesor
- obtenerConexiones
- obtenerPonderaciones
- obtenerPrecio

La clase Grafo tiene como atributos:

- self.listaVertices
- self.numVertices

Sus métodos son:

- agregarVertice
- obtenerVertice
- __contains__
- agregarArista
- obtenerVertices
- __iter__

Fuera de las clases anteriores definimos los métodos dijkstra_max_weight y dijkstra_min_cost.