Facultad de Ingeniería

Tecnicatura en Procesamiento y Explotación de Datos

Algoritmos y estructuras de datos

Trabajo Práctico n° 2

Colignon, Sabrina
Narváez, Micaela
Samaniego, Francisco

Desarrollo:

Ejercicio 1

Para la realización del primer ejercicio del TP elegimos como estructura un **montículo binario de mínimos,** este mismo se regirá teniendo en cuenta el valor que se le asigna a cada paciente según el riesgo de la urgencia. De esta manera, las personas que lleguen a la guardia y necesiten una atención urgente, se les asignará un nivel de riesgo igual a 1, a los que tengan una urgencia moderada un nivel de riesgo igual a 2 y a los de riesgo bajo un nivel igual a 3, de esta manera, nuestro montículo se ordenará infiltrado hacia arriba a los pacientes con un mayor nivel de peligro pero menor en el valor que se le da para asignar el riesgo (1, 2 y 3) logrando así un nivel de prioridad óptimo para atenderlos en la guardia. Utilizamos la implementación que nos brindaba el libro de teoría, cabe aclarar que esta estructura, internamente funciona con una lista de Python y cuenta con los métodos necesarios para ser operada:

- Inicializador: inicializa la clase montículo binario
- Insertar: método para agregar un elemento al final de la lista
- Infiltrar arriba: método para infiltrar un nuevo ítem hacia arriba en el caso que sea necesario
- Infiltrar abajo: método que infiltra un nodo hacia abajo en el caso de ser necesario.
- Hijo minimo: método para encontrar un hijo menor
- Eliminar mínimo: método para eliminar un nodo del montículo.

Además utilizamos la sobrecarga de dos operadores que nos serán útiles para poder trabajar:

- __iter__ : método para poder iterar o recorrer sobre el montículo.
- __len__: método que retorna el tamaño de la estructura.

Cada uno de estos métodos mencionados posee un orden de complejidad que será analizado en la siguiente tabla:

Análisis del orden de complejidad Big-O para cada métodos implementado en la clase “Montículo Binario”		
Método	Orden de complejidad	Análisis
Insertar(elemento)	O($log_2(n)$)	Inserta siempre los elementos en el último lugar de la lista. Se llama al método infiltrar arriba por ende se tiene en cuenta su orden de complejidad.
Infiltrar arriba(elemento)	O($log_2(n)$)	Infiltrar un nuevo ítem hacia arriba en el árbol hasta donde sea necesario
Infiltrar abajo(elemento)	O($log_2(n)$)	Infiltrar un nuevo ítem hacia abajo en el árbol hasta donde sea necesario
Hijo mínimo (elemento)	O(n)	Recorre el montículo y

		encuentra el menor hijo
Eliminar minimo (elemento)	O($log_2(n)$)	Elimina un nodo del montículo y restaurar la estructura del montículo. Se llama al método infiltrar abajo por ende se tiene en cuenta su orden de complejidad.
Iterador (unalista)	O(n)	Método para iterar y recorrer el montículo binario
Tamaño (unalista)	$O(1)$	Orden de complejidad O(1) ya que se encarga de invocar al método tamaño del montículo.

Para poder continuar resolviendo el problema propuesto, decidimos implementar una clase que guarde los datos de cada paciente, como su nombre, apellido y el valor de riesgo que se le asigna al entrar a la guardia según el nivel de urgencia.
Por último, es necesario comprobar si el montículo realmente cumple con su función por lo que creamos un nuevo código que simula una sala de guardia, cargando la llegada de los pacientes, dando mayor prioridad a los que posee un nivel de riesgo igual a 1 (riesgo alto), esto se puede ver en la siguiente imagen:

```
for i in range(n):
    # Fecha y hora de entrada de un paciente
    ahora = datetime.datetime.now()
    fecha_y_hora = ahora.strftime('%d/%m/%Y %H:%M:%S')
    print('-*-'*15)
    print('\n', fecha_y_hora, '\n')

    # Se crea un paciente un paciente por segundo
    # La criticidad del paciente es aleatoria
    paciente = pac.Paciente()
    cola_de_espera.insertar(paciente)
```

Imagen 1: Ciclo que gestiona la simulación de una guardia

Ejercicio 2

Para la realización del segundo ejercicio implementamos un árbol AVL, en él desarrollamos las clases nodo, AVL e iterador, cada una con los métodos correspondientes para su uso.
La codificación de estos métodos se copió y modificó, en lo casos necesarios, de la bibliografía. Se tuvieron que agregar la clase “Iterador” y métodos como “rotar derecha”.
Luego para aplicar el AVL desarrollamos una clase denominada “Temperaturas_DB” en la cuál se simula una base de datos la temperatura del planeta tierra dentro de un rango de fechas, en ella encontramos los métodos que están descritos en la tabla debajo.
Para esta clase se requería que en gran cantidad de métodos el usuario ingresará fechas, al ser estas ingresadas como string, para su manejar mejor las comparaciones se convirtieron a formato datetime, las temperaturas o “carga_util” se ingresa como valor flotante.
Para finalizar, implementamos un test para evaluar los resultados de los métodos de la clase “Temperaturas_DB”.

Análisis del orden de complejidad Big-O para cada métodos implementado en la clase “Temperaturas_DB”		
Método	Orden de complejidad	Análisis
guardar_temperatura (temperatura, fecha)	O($log(n)$)	Guardar temperatura funciona internamente como un insertar por lo que en el caso promedio tiene un orden de complejidad logarítmico por n.
devolver_temperatura (fecha)	O($log(n)$)	Utilizando el método obtener del árbol AVL, devolver temperatura tiene como orden promedio logaritmo de n por n..
max_temp_rango (fecha1, fecha2)	O($n\ log(n)$)	Utiliza el método obtener por lo que su orden de complejidad se rige por logaritmo de n por n.
min_temp_rango (fecha1, fecha2)	O($n\ log(n)$)	Utiliza el método obtener por lo que su orden de complejidad se rige por logaritmo de n por n.
temp_extremos_rango (fecha1, fecha2)	O($n\ log_2(n)$)	Utiliza los métodos min_temp_rango y max_temp_rango por lo que su orden de complejidad se rige por el logaritmo en base 2 de N

borrar_temperatura (fecha)	O($log_2(n)$)	Borrar una temperatura utiliza el método eliminar del AVL por lo que su orden de complejidad es el mismo que el de este: logaritmo de base 2 de n.
mostrar_temperaturas (fecha1, fecha2)	O(n)	Utiliza un objeto Iterador para ir recorriendo el árbol por nodos e ir guardando en una lista las temperaturas de menor a mayor
mostrar_cantidad_muestras ()	$O(1)$	Orden de complejidad O(1) ya que se encarga de invocar al método tamaño del AVL

Ejercicio 3: