Nombre de la práctica	Algoritmos de Balanceado AVL			**No.**	1
Asignatura:	Lenguajes y Autómatas ll	**Carrera:**	Ingeniería en Sistemas Computacionales	**Duración de la práctica (Hrs)**	4 hrs

I. Competencia(s) específica(s):

Define, diseña y programa las fases del analizador léxico y sintáctico de un traductor o compilador para preámbulo de la construcción de un compilador.

II. Lugar de realización de la práctica (laboratorio, taller, aula u otro):

Aula

III. Material empleado:

Visual Studio Code, Lenguaje de Programación Python, HTMLT5, CSS, JavaScript

IV. Desarrollo de la práctica:

BALANCEADOR DE ARBOLES AVL

Un árbol AVL es una estructura de datos que se utiliza en ciencias de la computación para almacenar y organizar datos de forma eficiente. Se trata de un tipo especial de árbol binario de búsqueda equilibrado en el que la diferencia de altura entre los subárboles izquierdo y derecho de cada nodo está limitada.

Funcionamiento de un árbol AVL:

- Cada nodo de un árbol AVL contiene un valor y dos referencias a sus subárboles izquierdo y derecho.
- El árbol cumple con la propiedad de que la clave de cualquier nodo en el subárbol izquierdo es menor que la clave del nodo y la clave de cualquier nodo en el subárbol derecho es mayor que la clave del nodo.
- La altura de un nodo se define como la longitud del camino más largo desde ese nodo hasta una hoja.
- El factor de equilibrio de un nodo se calcula como la diferencia entre la altura del subárbol derecho y la altura del subárbol izquierdo. Un árbol AVL debe cumplir que el factor de equilibrio de cada uno de sus nodos es -1, 0 o 1.

Pasos para balancear un árbol AVL:

1. Inserción: Cuando se inserta un nuevo nodo en un árbol AVL, se realiza la inserción de manera similar a un árbol binario de búsqueda. Después de insertar el nodo, se verifica el factor de equilibrio de cada nodo en el camino desde el nodo insertado hasta la raíz. Si algún nodo tiene un factor de equilibrio fuera de los límites permitidos (-1, 0, 1), se requieren rotaciones para restaurar el equilibrio.
2. Eliminación: Cuando se elimina un nodo de un árbol AVL, se realiza la eliminación de manera similar a un árbol binario de búsqueda. Después de eliminar el nodo, se verifica el factor de equilibrio de cada nodo en el camino desde el nodo eliminado hasta la raíz. Si algún nodo tiene un factor de equilibrio fuera de los límites permitidos, se requieren rotaciones para restaurar el equilibrio.
3. Rotaciones: Las rotaciones son operaciones que se aplican en los nodos del árbol para mantener o restaurar el equilibrio. Hay cuatro tipos de rotaciones posibles: rotación simple a la izquierda (LL), rotación simple a la derecha (RR), rotación doble a la izquierda (LR) y rotación doble a la derecha (RL). Las rotaciones se aplican según el tipo de desequilibrio y la estructura del árbol.

4. Actualización de factores de equilibrio: Después de realizar una rotación, es necesario actualizar los factores de equilibrio de los nodos afectados para reflejar los cambios en la estructura del árbol.
5. Recursividad: El proceso de balanceo en un árbol AVL es recursivo. Después de realizar una rotación, se verifica el equilibrio en el camino desde el nodo afectado hasta la raíz. Si es necesario, se aplican más rotaciones y se actualizan los factores de equilibrio hasta que todo el árbol esté balanceado.

El objetivo del balanceo en un árbol AVL es mantener la altura del árbol lo más pequeña posible para lograr una búsqueda eficiente. Al limitar la diferencia de altura entre los subárboles izquierdo y derecho de cada nodo, se garantiza que el tiempo de búsqueda, inserción y eliminación en el árbol AVL sea óptimo, con una complejidad de O(log n), donde "n" es el número de nodos en el árbol.

En resumen, el proceso de balanceo de un árbol AVL implica seguir estos pasos:

1. Insertar o eliminar un nodo en el árbol AVL.
2. Verificar el factor de equilibrio de los nodos en el camino desde el nodo afectado hasta la raíz.
3. Si se encuentra un nodo con un factor de equilibrio fuera de los límites permitidos (-1, 0, 1), se requiere una rotación.
4. Aplicar la rotación adecuada según el tipo de desequilibrio: LL, RR, LR o RL.
5. Actualizar los factores de equilibrio de los nodos afectados después de la rotación.
6. Repetir los pasos 2-5 hasta que todo el árbol esté balanceado.

El balanceo de un árbol AVL garantiza que la altura del árbol se mantenga óptima, lo que resulta en un rendimiento eficiente en las operaciones de búsqueda, inserción y eliminación. Al asegurar que el árbol esté equilibrado en todo momento, se evita que la estructura se degrade y se vuelva similar a una lista enlazada, lo cual afectaría negativamente la eficiencia.

En el presente algoritmo de balanceo de árbol AVL:

```
import sys
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QLineEdit, QPushButton, QVBoxLayout,
QWidget #Crea la ventana principal y sus componentes (etiqueta, input, y boton)


#Definimos la clase nodo
class Nodo:
    def __init__(self, data): #Creamos el constructor que recibe los siguientes atributos
        self.data = data  #Almacena el valor contenido en el nodo cuando se inserta
        self.izquierda = None #Apunta al nodo hijo izquierda actual, sino hay un nodo el valor sera
ninguno
        self.derecha = None #Apunta al nodo hijo derecho actual, sino hay un nodo el valor sera

class ArbolAVL: #Definimos la clase que inicializará lo que es el nodo raíz como vacio
    def __init__(self):
        self.raiz = None

    def insertar(self, data): #Inserta un nuevo nodo en el árbol  AVL
        self.raiz = self._insertar(self.raiz, data) #Inserción y asigna el valor del nodo raiz

    def _insertar(self, raiz, data): #Hace la insercion de un nuevo nodo
        if not raiz: #Si la raiz es ninguna
            return Nodo(data) #Se crea un nuevo nodo y se inserta como raiz
        elif data < raiz.data: #Si el dato ingresado es menor al dato de raíz lo acomodará a la
izquierda
            raiz.izquierda = self._insertar(raiz.izquierda, data) #Insertar el dato a la izquierda
        else:
            raiz.derecha = self._insertar(raiz.derecha, data) # Si es mayor insertar a la derecha

        factor_balance = self._obtener_factor_balance(raiz) #Obtiene el factor de balance

        if factor_balance > 1: #Si el factor de balance es mayor a 1 el nodo raiz
            if data < raiz.izquierda.data: #Si el dato es menor a la raíz la raíz el dato de la raíz
para a la izquierda
                return self._rotar_derecha(raiz) #Asigna el valor a la función de rotación a la derecha
            else: #Si no ocurre lo contrario
                raiz.izquierda = self._rotar_izquierda(raiz.izquierda)
                return self._rotar_derecha(raiz)
        if factor_balance < -1: #Aquí sucede lo mismo, si el valor del factor de balance es menor a -1
el nodo raíz
            if data > raiz.derecha.data:  #El dato de la raiz pasa a el dato de ese nodo a la derecha
                return self._rotar_izquierda(raiz) #Retorna el valor de la raíz a la izquierda
            else: #Sino a la derecha
                raiz.derecha = self._rotar_derecha(raiz.derecha)
                return self._rotar_izquierda(raiz)

        return raiz #Sale del condicional y retorna la raíz de acuerdo al nodo que se tenga en la raíz

    def _obtener_altura(self, raiz): #Obtiene la altura del arbol el maximo valor de la altura del
subnijo izquiero y derecho usando recursión
        if not raiz:
            return 0
        return max(self._obtener_altura(raiz.izquierda), self._obtener_altura(raiz.derecha)) + 1

    def _obtener_factor_balance(self, raiz): #Obtiene el factor de balance de cada nodo usando la
recursión
        if not raiz:
            return 0
        return self._obtener_altura(raiz.izquierda) - self._obtener_altura(raiz.derecha)

    def _rotar_izquierda(self, z):
        y = z.derecha
        if not y:
            return z

        T3 = y.izquierda
        #Hace la rotación izquierda intercambiando los nodos Y, Z
        y.izquierda = z
        z.derecha = T3

        return y

    def _rotar_derecha(self, z):
        y = z.izquierda
        if not y:
            return z

        T2 = y.derecha
        #Realiza la rotación derecha intercambiando los nodos Y, Z
        y.derecha = z
        z.izquierda = T2

        return y

    def dibujar(self): # Se encarga de realizar el dibujo pasandole los valore de x, y y el espaciado
entre los nodos
        self._dibujar_auxiliar(self.raiz, 0, 0, 20)

    def _dibujar_auxiliar(self, raiz, x, y, espaciado): #Este se encarga de  realizar el dibujo
recursivamente de cada subnijo
        if not raiz:
            return
        #Calculan las coordenadas para cada nodo
        x_izquierda = x - espaciado #Posiciona al nodo hijo izquierdo a una distancia horizontal hacia
la izquierda respecto a la vez que se encuentre ya ahí
        x_derecha = x + espaciado #Calcula el espaciado a la coordenada x y posicionarlo horizotalmente
hacia la derecha
        #Se encarga de dibujar el texto de cada nodo de acuerdo a su posicion que es en x y y y darle
los estilos
        plt.text(x, y, str(raiz.data), style='italic', bbox={'facecolor': '#37B449', 'edgecolor':
'darkgreen', 'pad': 1, 'boxstyle': 'circle'})

        if raiz.izquierda: #Verifica que el nodo actual tenga un hijo izquierdo, si la tiene la dibuja
desde (x, y) hasta (x_izquierda, y -1)
            plt.plot([x, x_izquierda], [y, y - 1], color='black')
            self._dibujar_auxiliar(raiz.izquierda, x_izquierda, y - 1, espaciado / 2)

        if raiz.derecha: #Verifica que el nodo actual tenga un hijo derecho, si la tiene la dibuja
desde (x, y) hasta (x_derecha, y -1);
            plt.plot([x, x_derecha], [y, y - 1], color='black')
            self._dibujar_auxiliar(raiz.derecha, x_derecha, y - 1, espaciado / 2)


class VentanaPrincipal(QMainWindow):
    def __init__(self):
        super().__init__()

        self.arbol_avl = ArbolAVL()

        self.setWindowTitle("Balanceador de Árboles AVL")

        self.etiqueta = QLabel("Ingresa un número:")
        self.entrada = QLineEdit()
        self.boton = QPushButton("Agregar")
        self.boton.clicked.connect(self.manejar_insercion)

        self.figura = plt.figure()
        self.canvas = FigureCanvas(self.figura)

        layout = QVBoxLayout()
        layout.addWidget(self.etiqueta)
        layout.addWidget(self.entrada)
        layout.addWidget(self.boton)
        layout.addWidget(self.canvas)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

    def manejar_insercion(self):
        #Maneja el evento de clic en el boton "Agregar"
        valor = self.entrada.text()
        try:
            numero = int(valor)
            self.arbol_avl.insertar(numero)
            self.entrada.clear()
            self.dibujar_arbol_avl()
        except ValueError:
            self.entrada.clear()
            self.entrada.setText("Entrada inválida. Ingresa un número.")

    def dibujar_arbol_avl(self):
        #Borra la figura actual y dibuja el arbol actualizado
        self.figura.clear()
        self.arbol_avl.dibujar()
        self.canvas.draw()


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ventana_principal = VentanaPrincipal()
    ventana_principal.show()
    sys.exit(app.exec_()) #salir
```

EXPLICACIÓN DEL ALGORITMO BALANCEADOR DE ARBOLES AVL

1. Importaciones de bibliotecas:
    - import sys: Se utiliza para acceder a los argumentos de la línea de comandos y finalizar la aplicación.
    - import matplotlib.pyplot as plt: Se utiliza para generar los nodos y las conexiones del árbol AVL y dibujarlos gráficamente.
    - from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas: Se utiliza para mostrar las figuras generadas por Matplotlib en la interfaz de PyQt5.
    - from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QLineEdit, QPushButton, QVBoxLayout, QWidget: Se importan las clases necesarias de la biblioteca PyQt5 para crear la ventana principal y sus componentes (etiqueta, entrada de texto y botón).
2. Definición de la clase Nodo:
    - Representa un nodo en el árbol AVL y almacena un valor (data) y las referencias a los nodos hijo izquierdo (izquierda) y derecho (derecha).
3. Definición de la clase ArbolAVL:
    - Representa el árbol AVL y contiene métodos para insertar nodos en el árbol, calcular la altura y el factor de balance de un nodo, y realizar rotaciones izquierda y derecha.
    - El método insertar permite insertar un nuevo nodo en el árbol AVL, manteniendo su propiedad de balance.
    - Los métodos privados (_insertar, _obtener_altura, _obtener_factor_balance, _rotar_izquierda, _rotar_derecha) son utilizados por el método insertar para realizar las operaciones necesarias en el árbol.
4. Método dibujar de la clase ArbolAVL:
    - Este método se encarga de dibujar el árbol AVL utilizando la biblioteca Matplotlib.
    - Utiliza el método privado _dibujar_auxiliar de forma recursiva para dibujar cada nodo y sus conexiones en la figura.
5. Definición de la clase VentanaPrincipal:
    - Representa la ventana principal de la aplicación.
    - Contiene un objeto de la clase ArbolAVL para almacenar y manipular el árbol AVL.
    - Define los componentes de la interfaz gráfica, como una etiqueta, una entrada de texto y un botón para agregar números al árbol.
    - El método manejar_insercion se ejecuta cuando se hace clic en el botón "Agregar" y maneja la inserción de números en el árbol AVL.
    - El método dibujar_arbol_avl se encarga de borrar la figura actual y dibujar el árbol AVL actualizado utilizando la biblioteca Matplotlib.
6. Bloque principal:
    - Crea una aplicación de PyQt5 y una instancia de la clase VentanaPrincipal.
    - Muestra la ventana principal y ejecuta la aplicación hasta que se cierre.

V. Conclusiones:

La implementación de árboles AVL en el código permite mantener un árbol balanceado automáticamente, lo que mejora la eficiencia de las operaciones de búsqueda, inserción y eliminación. Además, la interfaz gráfica proporciona una visualización clara y fácil de entender del árbol AVL en tiempo real.