LLEGA RAPIDITO

MANUAL TECNICO

Presentación

El siguiente manual ha sido desarrollado con la finalidad de dar a conocer la información necesaria para realizar el mantenimiento, instalación y exploración del software de **Llega Rapidito**, el cual consta de diferentes funcionalidades. El manual ofrece la información necesaria de ¿Cómo está desarrollado el software? Para que la persona (Desarrollador en Python) que quiera editar el software lo haga de una manera apropiada, dando a conocer la estructura del desarrollo del aplicativo.

Resumen

El manual detalla los aspectos técnicos e informativos del software de **Llega Rapidito** con la finalidad de explicar la estructura del aplicativo al personal que quiera administrarlo, editarlo o configurarlo. La siguiente guía se encuentra dividida en las herramientas que se usaron para el desarrollo del software con una breve explicación paso a paso. El aplicativo maneja diferentes funcionalidades los cuales se explicara que funcionamiento realiza cada uno de ellos, dando sugerencias para el debido uso del sistema informático.

Introducción

El manual se realiza con el fin de detallar el software para **Llega Rapidito** en términos técnicos para aquella persona que vaya a administrar, editar o configurar el aplicativo lo haga de una manera apropiada. El documento se encuentra dividido en las siguientes secciones:

- ENTORNO DE DESARROLLO: Se darán a conocer las herramientas de desarrollo que se utilizaron para el software, así como se darán breves explicaciones de instalaciones y configuraciones necesarias de las herramientas anteriormente expuestas
- DESCRIPCION DE CODIGO FUENTE: Se describirán las clases y métodos que fueron implementados dentro del código fuente de la aplicación
- DESCRIPCION DE TDAs: Se explicaran las estructuras de datos que se utilizaron para el desarrollo de la aplicación

ENTORNO DE DESARROLLO

El aplicativo de **Llega Rapidito** tiene la finalidad de administrar el servicio de Transporte con cobertura nacional, además de llevar el control de los Vehículos, Clientes y Viajes que maneja la empresa. Se recomienda que el siguiente manual sea manipulado únicamente por la persona que quiera administrar, editar o configurar el software de **Llega Rapidito** para velar por la seguridad de los datos que se almacenan. A continuación se presenta la instalación, configuración y descripción de las herramientas utilizadas para el desarrollo de la aplicación:

Visual Studio Code

- Descripción

Es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. También es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos del teclado y las preferencias. Es software gratuito y de código abierto, aunque la descarga oficial está bajo software privativo e incluye características personalizadas por Microsoft.

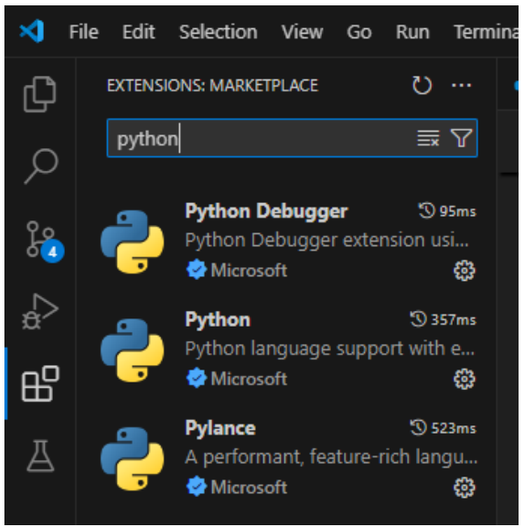
Visual Studio Code se basa en Electron, un framewoek que utiliza para implementar Chromium y Node.js como aplicaciones para escritorio, que se ejecuta en el motor de diseño Blink. Aunque utiliza el framework Electron, el software no usa Atom y en su luga emplea el mismo componente editor (Monaco) utilizado en Visual Studio Team Services (Anteriormente llamado Visual Studio Online)

- Instalación

Para la instalación de Visual Studio Code desde la página oficial de Visual Studio Code dirigiste al siguiente enlace: https://code.visualstudio.com/download en donde podrás descargar Visual Studio Code para su posterior instalación.

- Configuración

Lo único que necesitas es instalar el paquete de Python para poder desarrollar de una forma óptima y adecuada sobre este lenguaje, para ello dirigiste a extensiones, posteriormente busca "python", e instala las 3 opciones que a continuación se te presentan:



Graphviz

- Descripción

Graphviz es un software de visualización de gráficos de código abierto. La visualización de gráficos es una forma de representar información estructural como diagramas de redes y gráficos abstractos. Tiene importantes aplicaciones en redes, bioinformática, ingeniería de software, diseño de base de datos y sitios web, aprendizaje automático y en interfaces visuales para otros dominios técnicos.

Los programas de diseño de Graphviz toman descripciones de gráficos en un lenguaje de texto simple y crean diagramas en formato útiles, como imágenes y SVG para páginas web; PDF Postscript para incluir en otros documentos; o visualizar en un navegador concreto, como opción de colores, fuentes, diseños de nodos tabulares, estilos de líneas, hipervínculos y formas personalizadas.

- Instalación

Para los que instalaron WSL, es recomendable que instalen Graphviz en los dos sistemas operativos por si acaso uno de los dos fallara (no debería) tener el otro de reserva, de igual forma este software es ligero en descarga e instalación.

Linux: ejecutar los siguientes comandos en la consola/terminal de WSL o Linux directamente.

sudo apt update: Con este comando se buscaran actualizaciones para los paquetes de la distribución de Linux instalada

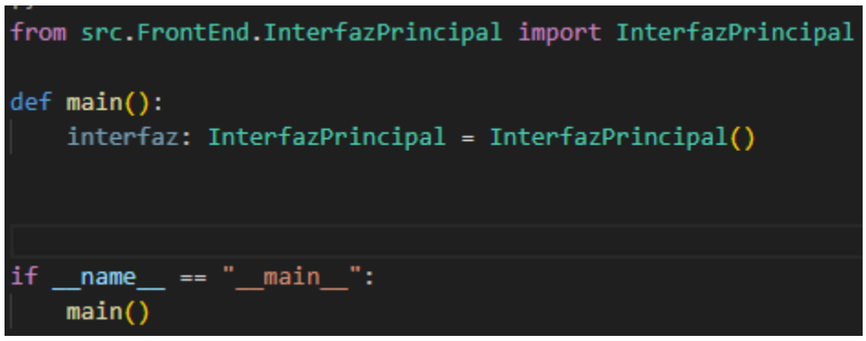
sudo apt upgrade: Con este comando se aplicaran las actualizaciones (si los hay) de los paquetes que anteriormente se encontraron. Si en el comando anterior no se encontraron paquetes por actualizar no es necesario ejecutar este comando.

sudo apt install graphviz: Este comando instalara todo lo necesario para tener Graphviz

Windows: Para la instalación de Graphviz desde su página oficial dirígete al siguiente enlace: https://graphviz.org/download/ en donde podrás descargar el ejecutable para Windows y así hacer su instalación.

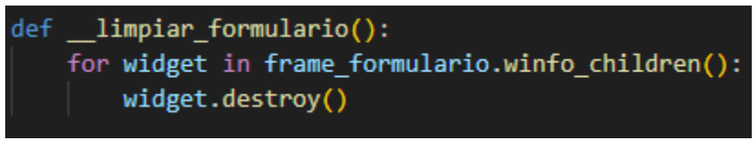
DESCRIPCION DE CODIGO FUENTE

- **main.py:** Clase main estándar de Python que es en donde se empieza a ejecutar el programa



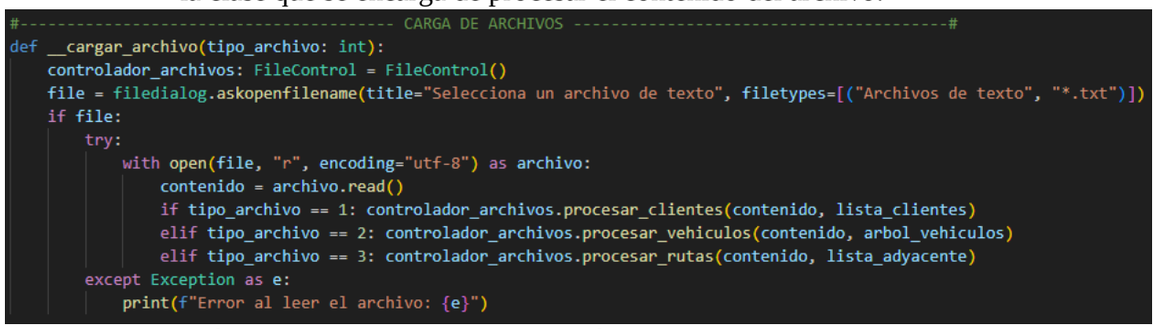
```python
from src.FrontEnd.InterfazPrincipal import InterfazPrincipal

def main():
    interfaz: InterfazPrincipal = InterfazPrincipal()



if __name__ == "__main__":
    main()
```

- **InterfazPrincipal.py:** Esta clase es la encargada de manejar toda la Interfaz de Usuario que tendrá la aplicación, además de contener la instancia de las clases más importantes que manejaran toda la lógica interna de la aplicación
    - __limpiar_formulario(): Método que se encarga de limpiar y eliminar todos los componentes que se pudieran tener actualmente en la vista para que se pueda cargar contenido nuevo sin ningún tipo de errores.



```python
def __limpiar_formulario():
    for widget in frame_formulario.winfo_children():
        widget.destroy()
```

    - __carga_archivo(int): Método que se encarga de mostrar la ventana para que el usuario pueda elegir el archivo de texto que servirá para cargar datos a la aplicación, esta ventana solo le permitirá al usuario elegir archivos de texto .txt, el parámetro de tipo *int* que recibe es para saber qué tipo de contenido de archivo se mandara a la clase que se encarga de procesar el contenido del archivo.



```python
#------------------------------------ CARGA DE ARCHIVOS ------------------------------------#
def __cargar_archivo(tipo_archivo: int):
    controlador_archivos: FileControl = FileControl()
    file = filedialog.askopenfilename(title="Selecciona un archivo de texto", filetypes=[("Archivos de texto", "*.txt")])
    if file:
        try:
            with open(file, "r", encoding="utf-8") as archivo:
                contenido = archivo.read()
                if tipo_archivo == 1: controlador_archivos.procesar_clientes(contenido, lista_clientes)
                elif tipo_archivo == 2: controlador_archivos.procesar_vehiculos(contenido, arbol_vehiculos)
                elif tipo_archivo == 3: controlador_archivos.procesar_rutas(contenido, lista_adyacente)
        except Exception as e:
            print(f"Error al leer el archivo: {e}")
```

    - __formulario_agregar_cliente(): Método que inicia limpiando el contenido anterior, para luego empezar a rende rizar los *Label*, *Entry* y *Button*, que representaran el formulario adecuado para que el usuario pueda hacer un nuevo registro de cliente, que luego será procesado por el siguiente método.
    - __procesar_agregar_cliente(int, str, str, str, int, str, str): Método que instancia un objeto del tipo *Cliente* con los datos de parámetro que recibe, para luego mandar al objeto a la clase que se encargara de agregarlo al sistema, y si todo sale satisfactoriamente se mostrar un mensaje emergente informando que se ha realizado el registro de forma exitosa, y por ultimo limpia el formulario, para no tener problemas con contenido basura.

```python
def __formulario_agregar_cliente():
    __limpiar_formulario()
    Label(frame_formulario, text="FORMULARIO PARA CREACION DE CLIENTE", foreground="red").grid(row=0, columnspan=2, padx=5, pady=5)
    Label(frame_formulario, text="DPI: ").grid(row=1, column=0, padx=5, pady=5)
    dpi = IntVar()
    dpi_entry = Entry(frame_formulario, textvariable=dpi)
    dpi_entry.grid(row=1, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Nombre:").grid(row=2, column=0, padx=5, pady=5)
    nombre = StringVar()
    nombre_entry: Entry = Entry(frame_formulario, textvariable=nombre)
    nombre_entry.grid(row=2, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Apellidos: ").grid(row=3, column=0, padx=5, pady=5)
    apellidos = StringVar()
    apellidos_entry = Entry(frame_formulario, textvariable=apellidos)
    apellidos_entry.grid(row=3, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Genero:").grid(row=4, column=0, padx=5, pady=5)
    genero = StringVar()
    genero_entry = Entry(frame_formulario, textvariable=genero)
    genero_entry.grid(row=4, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Telefono: ").grid(row=5, column=0, padx=5, pady=5)
    telefono = IntVar()
    telefono_entry = Entry(frame_formulario, textvariable=telefono)
    telefono_entry.grid(row=5, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Direccion:").grid(row=6, column=0, padx=5, pady=5)
    direccion = StringVar()
    direccion_entry = Entry(frame_formulario, textvariable=direccion)
    direccion_entry.grid(row=6, column=1, padx=5, pady=5)
    Button(frame_formulario, text="Registrar", command=lambda: __procesar_agregar_cliente(dpi.get(), nombre.get(), apellidos.get(), genero.get(), telefono.get(), direccion.get())).grid(row=7, columnspan=3, pady=10)

def __procesar_agregar_cliente(dpi: int, nombre: str, apellidos: str, genero: str, telefono: int, direccion: str):
    nuevo_cliente: Cliente = Cliente(dpi, nombre, apellidos, genero, telefono, direccion)
    lista_clientes.insertar_cliente(nuevo_cliente)
    messagebox.showinfo("EXITO!!!", "Nuevo Cliente Registrado en el Sistema")
    __limpiar_formulario()
```

- __formulario_modificar_cliente():Método que inicia limpiando el contenido anterior, para luego empezar a rende rizar los *Label*, *Entry* y *Button*, que representaran el formulario adecuado para que el usuario pueda hacer la modificación de datos de un cliente, que luego será procesado por el siguiente método.
- __procesar_modificar_cliente(int, str, str, str, int, str, str): Método que instancia un objeto del tipo *Cliente* con los datos de parámetro que recibe, para luego mandar al objeto a la clase que se encargara de hacer la modificación en el sistema, y si todo sale satisfactoriamente se mostrar un mensaje emergente informando que se ha realizado el cambio de forma exitosa, y por ultimo limpia el formulario, para no tener problemas con contenido basura.

```python
def __formulario_modificar_cliente():
    __limpiar_formulario()
    if lista_clientes.mostrar_lista(frame_formulario):
        Label(frame_formulario, text="FORMULARIO PARA MODIFICACION DE CLIENTE", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
        Label(frame_formulario, text="DPI: ").grid(row=2, column=0, padx=5, pady=5)
        dpi = IntVar()
        dpi_entry = Entry(frame_formulario, textvariable=dpi)
        dpi_entry.grid(row=2, column=1, padx=5, pady=5)
        Label(frame_formulario, text="Nombre:").grid(row=3, column=0, padx=5, pady=5)
        nombre = StringVar()
        nombre_entry: Entry = Entry(frame_formulario, textvariable=nombre)
        nombre_entry.grid(row=3, column=1, padx=5, pady=5)
        Label(frame_formulario, text="Apellidos: ").grid(row=4, column=0, padx=5, pady=5)
        apellidos = StringVar()
        apellidos_entry = Entry(frame_formulario, textvariable=apellidos)
        apellidos_entry.grid(row=4, column=1, padx=5, pady=5)
        Label(frame_formulario, text="Genero:").grid(row=5, column=0, padx=5, pady=5)
        genero = StringVar()
        genero_entry = Entry(frame_formulario, textvariable=genero)
        genero_entry.grid(row=5, column=1, padx=5, pady=5)
        Label(frame_formulario, text="Telefono: ").grid(row=6, column=0, padx=5, pady=5)
        telefono = IntVar()
        telefono_entry = Entry(frame_formulario, textvariable=telefono)
        telefono_entry.grid(row=6, column=1, padx=5, pady=5)
        Label(frame_formulario, text="Direccion:").grid(row=7, column=0, padx=5, pady=5)
        direccion = StringVar()
        direccion_entry = Entry(frame_formulario, textvariable=direccion)
        direccion_entry.grid(row=7, column=1, padx=5, pady=5)
        Button(frame_formulario, text="Modificar", command=lambda: __procesar_modificar_cliente(dpi.get(), nombre.get(), apellidos.get(), genero.get(), telefono.get(), direccion.get())).grid(row=8, columnspan=3, pady=

def __procesar_modificar_cliente(dpi: int, nombre: str, apellidos: str, genero: str, telefono: int, direccion: str):
    cliente_modificado: Cliente = Cliente(dpi, nombre, apellidos, genero, telefono, direccion)
    lista_clientes.modificar_cliente(cliente_modificado)
    __limpiar_formulario()
```

- __formulario_eliminar_cliente(): Método que inicia limpiando el contenido anterior, para luego rende rizar (siempre que haya contenido que mostrar) los *Label*, *Entry* y *Button*, que representan el formulario adecuado para que el usuario pueda hacer la eliminación de un cliente en el sistema, que luego será procesado por el siguiente método.

- __procesar_eliminar_cliente(int): Método que envía el parámetro de tipo *int* que recibe al método de la clase que se encargara de eliminar al cliente del sistema, para luego limpiar el formulario y así no tener problemas con contenido basura.

```
def __formulario_eliminar_cliente():
    __limpiar_formulario()
    if lista_clientes.mostrar_lista(frame_formulario):
        Label(frame_formulario, text="FORMULARIO PARA ELIMINACION DE CLIENTE", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
        Label(frame_formulario, text="DPI: ").grid(row=2, column=0, padx=5, pady=5)
        dpi = IntVar()
        dpi_entry = Entry(frame_formulario, textvariable=dpi)
        dpi_entry.grid(row=2, column=1, padx=5, pady=5)
        Button(frame_formulario, text="Eliminar", command=lambda: __procesar_eliminar_cliente(dpi.get())).grid(row=3, columnspan=3, pady=10)

def __procesar_eliminar_cliente(dpi: int):
    lista_clientes.eliminar_cliente(dpi)
    __limpiar_formulario()
```

- __formulario_mostrar_informacion_cliente():Método que inicia limpiando el contenido anterior, para luego rende rizar (siempre que haya contenido que mostrar) los *Label*, y *Button*, que representan el formulario adecuado para que el usuario pueda consultar la información de un cliente en el sistema en base a su dpi, que luego será procesado por el siguiente método.
- __procesar_mostrar_informacion_clietne(): Método que invoca al meto de la clase que se encarga de mostrar la información de cliente solicitado

```
def __formulario_mostrar_informacion_cliente():
    __limpiar_formulario()
    if lista_clientes.mostrar_dpis(frame_formulario):
        Label(frame_formulario, text="FORMULARIO PARA MOSTRAR INFORMACION DE CLIENTE", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
        Button(frame_formulario, text="Mostrar Informacion", command=lambda: __procesar_mostrar_informacion_cliente()).grid(row=2, columnspan=3, pady=10)

def __procesar_mostrar_informacion_cliente():
    lista_clientes.mostrar_informacion()
```

- __formulario_agregar_vehiculo(): Método que inicia limpiando el contenido anterior, para luego empezar a rende rizar los *Label*, *Entry* y *Button*, que representaran el formulario adecuado para que el usuario pueda hacer un nuevo registro de vehiculo, que luego será procesado por el siguiente método.
- __procesar_agregar_vehiculo(str, str, int, float): Método que instancia un objeto del tipo *Vehiculo* con los datos de parámetro que recibe, para luego mandar al objeto a la clase que se encargara de agregarlo al sistema, y si todo sale satisfactoriamente se mostrar un mensaje emergente informando que se ha realizado el registro de forma exitosa, y por ultimo limpia el formulario, para no tener problemas con contenido basura.

```
def __formulario_agregar_vehiculo():
    __limpiar_formulario()
    Label(frame_formulario, text="FORMULARIO PARA CREACION DE VEHICULO", foreground="red").grid(row=0, columnspan=2, padx=5, pady=5)
    Label(frame_formulario, text="Placa: ").grid(row=1, column=0, padx=5, pady=5)
    placa = StringVar()
    placa_entry = Entry(frame_formulario, textvariable=placa)
    placa_entry.grid(row=1, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Marca:").grid(row=2, column=0, padx=5, pady=5)
    marca = StringVar()
    marca_entry = Entry(frame_formulario, textvariable=marca)
    marca_entry.grid(row=2, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Modelo: ").grid(row=3, column=0, padx=5, pady=5)
    modelo = IntVar()
    modelo_entry = Entry(frame_formulario, textvariable=modelo)
    modelo_entry.grid(row=3, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Precio:").grid(row=4, column=0, padx=5, pady=5)
    precio = DoubleVar()
    precio_entry = Entry(frame_formulario, textvariable=precio)
    precio_entry.grid(row=4, column=1, padx=5, pady=5)
    Button(frame_formulario, text="Registrar", command=lambda: __procesar_agregar_vehiculo(placa.get(), marca.get(), modelo.get(), precio.get())).grid(row=5, columnspan=3, pady=10)

def __procesar_agregar_vehiculo(placa: str, marca: str, modelo: int, precio: float):
    nuevo_vehiculo: Vehiculo = Vehiculo(placa, marca, modelo, precio)
    arbol_vehiculos.insertar_vehiculo(nuevo_vehiculo)
    messagebox.showinfo("EXITO!!!", "Nuevo Vehiculo Registrado en el Sistema")
    __limpiar_formulario()
```

- __formulario_modificar_vehiculo():Método que inicia limpiando el contenido anterior, para luego empezar a rende rizar los *Label*, *Entry* y *Button*, que representaran el formulario adecuado para que el usuario pueda hacer la modificación de datos de un vehículo, que luego será procesado por el siguiente método.

- __procesar_modificar_vehiculo(str, str, int, float): Método que instancia un objeto del tipo *Vehiculo* con los datos de parámetro que recibe, para luego mandar al objeto a la clase que se encargara de hacer la modificación en el sistema, y si todo sale satisfactoriamente se mostrar un mensaje emergente informando que se ha realizado el cambio de forma exitosa, y por ultimo limpia el formulario, para no tener problemas con contenido basura.

```
def __formulario_modificar_vehiculo():
    __limpiar_formulario()
    arbol_vehiculos.recorrer_arbol(frame_formulario)
    Label(frame_formulario, text="FORMULARIO PARA MODIFICACION DE VEHICULO", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
    Label(frame_formulario, text="Placa: ").grid(row=2, column=0, padx=5, pady=5)
    placa = StringVar()
    placa_entry = Entry(frame_formulario, textvariable=placa)
    placa_entry.grid(row=2, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Marca:").grid(row=3, column=0, padx=5, pady=5)
    marca = StringVar()
    marca_entry = Entry(frame_formulario, textvariable=marca)
    marca_entry.grid(row=3, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Modelo: ").grid(row=4, column=0, padx=5, pady=5)
    modelo = IntVar()
    modelo_entry = Entry(frame_formulario, textvariable=modelo)
    modelo_entry.grid(row=4, column=1, padx=5, pady=5)
    Label(frame_formulario, text="Precio:").grid(row=5, column=0, padx=5, pady=5)
    precio = DoubleVar()
    precio_entry = Entry(frame_formulario, textvariable=precio)
    precio_entry.grid(row=5, column=1, padx=5, pady=5)
    Button(frame_formulario, text="Modificar", command=lambda: __procesar_modificar_vehiculo(placa.get(), marca.get(), modelo.get(), precio.get())).grid(row=6, columnspan=3, pady=10)

def __procesar_modificar_vehiculo(placa: str, marca: str, modelo: int, precio: float):
    vehiculo_modificado: Vehiculo = Vehiculo(placa, marca, modelo, precio)
    arbol_vehiculos.modificar_vehiculo(vehiculo_modificado)
    __limpiar_formulario()
```

- __formulario_eliminar_vehiculo(): Método que inicia limpiando el contenido anterior, para luego rende rizar (siempre que haya contenido que mostrar) los *Label*, *Entry* y *Button*, que representan el formulario adecuado para que el usuario pueda hacer la eliminación de un vehículo en el sistema, que luego será procesado por el siguiente método.
- __procesar_eliminar_vehiculo(str): Método que envía el parámetro de tipo *str* que recibe al método de la clase que se encargara de eliminar al vehículo del sistema, para luego limpiar el formulario y así no tener problemas con contenido basura.

```
def __formulario_eliminar_vehiculo():
    __limpiar_formulario()
    arbol_vehiculos.recorrer_arbol(frame_formulario)
    Label(frame_formulario, text="FORMULARIO PARA ELIMINACION DE VEHICULO", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
    Label(frame_formulario, text="Placa: ").grid(row=2, column=0, padx=5, pady=5)
    placa = StringVar()
    placa_entry = Entry(frame_formulario, textvariable=placa)
    placa_entry.grid(row=2, column=1, padx=5, pady=5)
    Button(frame_formulario, text="Eliminar", command=lambda: __procesar_eliminar_vehiculo(placa)).grid(row=3, columnspan=3, pady=10)

def __procesar_eliminar_vehiculo(placa: str):
    vehiculo_eliminar: Vehiculo = Vehiculo(placa, "", 0, 0)
    arbol_vehiculos.eliminar_vehiculo(arbol_vehiculos.get_raiz(), vehiculo_eliminar)
    __limpiar_formulario()
```

- __formulario_mostrar_informacion_vehiculo():Método que inicia limpiando el contenido anterior, para luego rende rizar (siempre que haya contenido que mostrar) los *Label*, y *Button*, que representan el formulario adecuado para que el usuario pueda consultar la información de un vehículo en el sistema en base a su placa, que luego será procesado por el siguiente método.
- __procesar_mostrar_informacion_vehiculo(): Método que invoca al meto de la clase que se encarga de mostrar la información del vehículo solicitado

```
def __formulario_mostrar_informacion_vehiculo():
    __limpiar_formulario()
    arbol_vehiculos.mostrar_placas(frame_formulario)
    Label(frame_formulario, text="FORMULARIO PARA MOSTRAR INFORMACION DE VEHICULO", foreground="red").grid(row=1, columnspan=2, padx=5, pady=5)
    Button(frame_formulario, text="Mostrar Informacion", command=lambda: __procesar_mostrar_informacion_vehiculo()).grid(row=2, columnspan=3, pady=10)

def __procesar_mostrar_informacion_vehiculo():
    arbol_vehiculos.mostrar_informacion()
```

- **FileControl.py:** Clase que se encarga de procesar el contenido de archivo de texto que corresponde a las cargas de datos de las diferentes entidades que maneja la aplicación, la clase analiza los datos recibidos para luego instanciar clases dependiendo el conjunto de datos que reciba.
    - procesar_clientes(str, ListaClietnes): Método que procesa el contenido *str* que recibe como parámetro, el proceso se realiza dividiendo el texto en secciones divididas por **(;)**, luego estas secciones de texto se les elimina los saltos de línea si llegaran a tener, para finalmente dividir esta sección de texto en datos divididos por **(,)** que servirán como argumentos para instanciar clases del tipo *Cliente* que posteriormente se agregaran a la estructura de datos que le corresponde. Al finalizar todo el proceso de análisis del texto, se enviara un mensaje informando que la carga de datos fue realizada con éxito.

```
def procesar_clientes(self, contenido: str, lista_clientes: ListaClientes):
    print("PROCESANDO CLIENTES...")
    lineas: list[str] = contenido.split(";")
    for linea in lineas:
        if linea.__contains__("\n"):
            linea = linea.replace("\n", "")
        if linea != "":
            contenido: list[str] = linea.split(",")
            cliente: Cliente = Cliente(int(contenido[0]), contenido[1], contenido[2], contenido[3], int(contenido[4]), contenido[5])
            lista_clientes.insertar_cliente(cliente)
    messagebox.showinfo("EXITO!!!", "Carga de Clientes Realizado Exitosamente")
```

    - procesar_vehiculos(str, ArolB): Método que procesa el contenido *str* que recibe como parámetro, el proceso se realiza dividiendo el texto en secciones divididas por **(;)**, luego estas secciones de texto se les elimina los saltos de línea si llegaran a tener, para finalmente dividir esta sección de texto en datos divididos por **(:)** que servirán como argumentos para instanciar clases del tipo *Vehiculo* que posteriormente se agregaran a la estructura de datos que le corresponde. Al finalizar todo el proceso de análisis del texto, se enviara un mensaje informando que la carga de datos fue realizada con éxito.

```
def procesar_vehiculos(self, contenido: str, arbol_vehiculos: ArbolB):
    print("PROCESANDO VEHICLUOS...")
    lineas: list[str] = contenido.split(";")
    for linea in lineas:
        if linea.__contains__("\n"):
            linea = linea.replace("\n", "")
        if linea != "":
            contenido: list[str] = linea.split(":")
            vehiculo: Vehiculo = Vehiculo(contenido[0], contenido[1], int(contenido[2]), float(contenido[3]))
            arbol_vehiculos.insertar_vehiculo(vehiculo)
    messagebox.showinfo("EXITO!!!", "Carga de Vehiculos Realizado Exitosamente")
```

    - procesar_rutas(str, ListaAdyacencia): Método que procesa el contenido *str* que recibe como parámetro, el proceso se realiza dividiendo el texto en secciones divididas por (**%**), luego estas secciones de texto se les elimina los saltos de línea y los espacios en blanco si llegaran a tener, para finalmente dividir esta sección de texto en datos divididos por **(/)** que servirán como argumentos para instanciar clases del tipo *Cliente* que posteriormente se agregaran a la estructura de datos que le corresponde. Al finalizar todo el proceso de análisis del texto, se enviara un mensaje informando que la carga de datos fue realizada con éxito.

```
def procesar_rutas(self, contenido: str, lista_adyacente: ListaAdyacencia):
    print("PROCESANDO RUTAS...")
    lineas: list[str] = contenido.split("%")
    for linea in lineas:
        if linea.__contains__("\n"): linea = linea.replace("\n", "")
        if linea.__contains__(" "): linea = linea.replace(" ", "")
        if linea != "":
            contenido: list[str] = linea.split("/")
            ruta: Ruta = Ruta(contenido[0], contenido[1], int(contenido[2]))
            lista_adyacente.insertar(ruta)
    messagebox.showinfo("EXITO!!!", "Carga de Rutas Realizado Exitosamente")
```

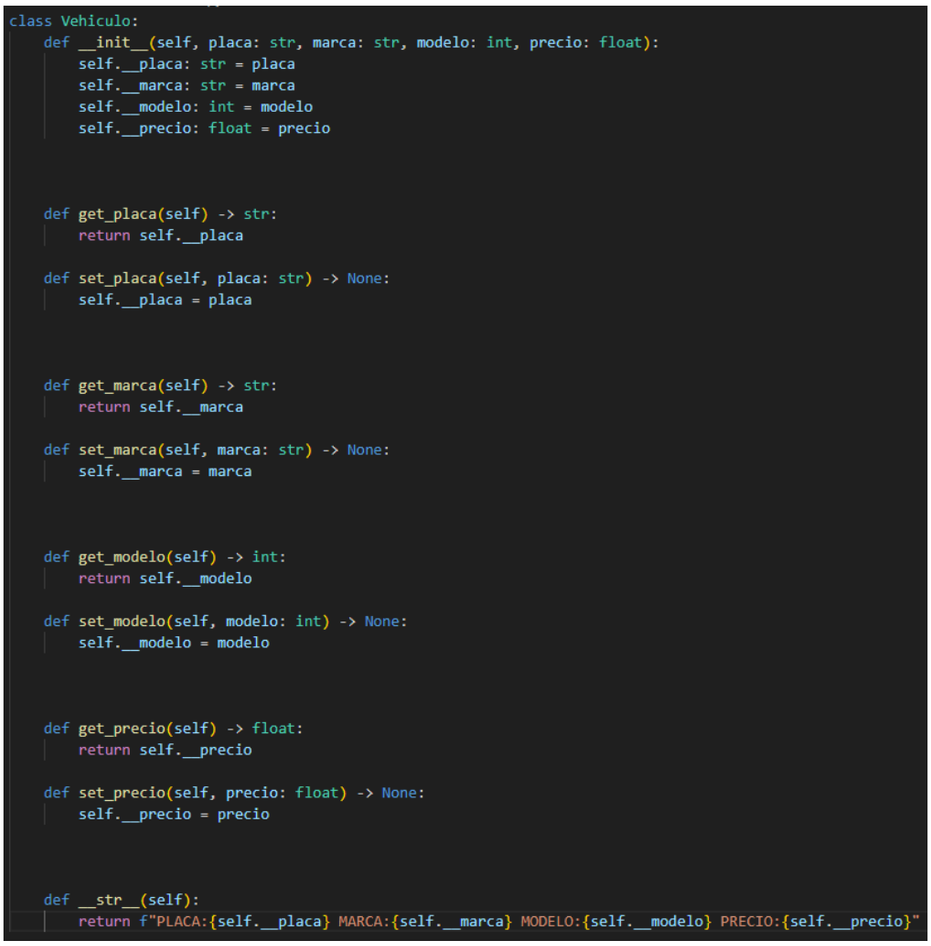
- **Cliente.py:** Es la definición de la Clase Cliente en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un usuario dentro de la aplicación

```python
class Cliente:
    def __init__(self, dpi: int, nombre: str, apellido: str, genero: str, telefono: int, direccion: str):
        self.__dpi: int = dpi
        self.__nombre: str = nombre
        self.__apellido: str = apellido
        self.__genero: str = genero
        self.__telefono: int = telefono
        self.__direccion: str = direccion



    def get_dpi(self) -> int:
        return self.__dpi

    def set_dpi(self, dpi: int) -> None:
        self.__dpi = dpi



    def get_nombre(self) -> str:
        return self.__nombre

    def set_nombre(self, nombre: str) -> None:
        self.__nombre = nombre



    def get_apellido(self) -> str:
        return self.__apellido

    def set_apellido(self, apellido: str) -> None:
        self.__apellido = apellido



    def get_genero(self) -> str:
        return self.__genero

    def set_genero(self, genero: str) -> None:
        self.__genero = genero



    def get_telefono(self) -> int:
        return self.__telefono

    def set_telefono(self, telefono: int) -> None:
        self.__telefono = telefono
```

- **Ruta.py:** Es la definición de la Clase Ruta en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un usuario dentro de la aplicación

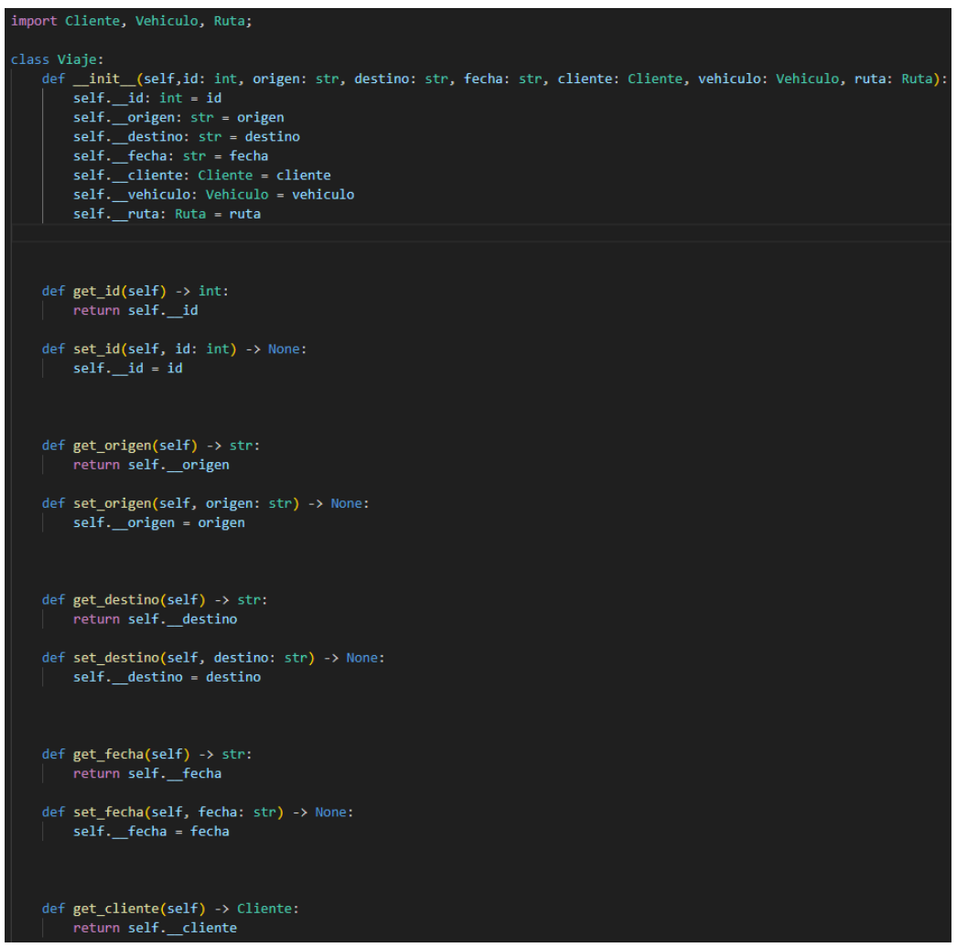
```python
class Ruta:
    def __init__(self, origen: str, destino: str, tiempo: int):
        self.__origen: str = origen
        self.__destino: str = destino
        self.__tiempo: int = tiempo



    def get_origen(self) -> str:
        return self.__origen

    def set_origen(self, origen: str) -> None:
        self.__origen = origen



    def get_destino(self) -> str:
        return self.__destino

    def set_destino(self, destino: str) -> None:
        self.__destino = destino



    def get_tiempo(self) -> int:
        return self.__tiempo

    def set_tiempo(self, tiempo: int) -> None:
        self.__tiempo = tiempo
```

- **Vehiculo.py:** Es la definición de la Clase Vehículo en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un usuario dentro de la aplicación



```python
class Vehiculo:
    def __init__(self, placa: str, marca: str, modelo: int, precio: float):
        self.__placa: str = placa
        self.__marca: str = marca
        self.__modelo: int = modelo
        self.__precio: float = precio



    def get_placa(self) -> str:
        return self.__placa

    def set_placa(self, placa: str) -> None:
        self.__placa = placa



    def get_marca(self) -> str:
        return self.__marca

    def set_marca(self, marca: str) -> None:
        self.__marca = marca



    def get_modelo(self) -> int:
        return self.__modelo

    def set_modelo(self, modelo: int) -> None:
        self.__modelo = modelo



    def get_precio(self) -> float:
        return self.__precio

    def set_precio(self, precio: float) -> None:
        self.__precio = precio



    def __str__(self):
        return f"PLACA:{self.__placa} MARCA:{self.__marca} MODELO:{self.__modelo} PRECIO:{self.__precio}"
```

- **Viaje.py:** Es la definición de la Clase Viaje en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un usuario dentro de la aplicación



```python
import Cliente, Vehiculo, Ruta;

class Viaje:
    def __init__(self,id: int, origen: str, destino: str, fecha: str, cliente: Cliente, vehiculo: Vehiculo, ruta: Ruta):
        self.__id: int = id
        self.__origen: str = origen
        self.__destino: str = destino
        self.__fecha: str = fecha
        self.__cliente: Cliente = cliente
        self.__vehiculo: Vehiculo = vehiculo
        self.__ruta: Ruta = ruta



    def get_id(self) -> int:
        return self.__id

    def set_id(self, id: int) -> None:
        self.__id = id



    def get_origen(self) -> str:
        return self.__origen

    def set_origen(self, origen: str) -> None:
        self.__origen = origen



    def get_destino(self) -> str:
        return self.__destino

    def set_destino(self, destino: str) -> None:
        self.__destino = destino



    def get_fecha(self) -> str:
        return self.__fecha

    def set_fecha(self, fecha: str) -> None:
        self.__fecha = fecha



    def get_cliente(self) -> Cliente:
        return self.__cliente
```

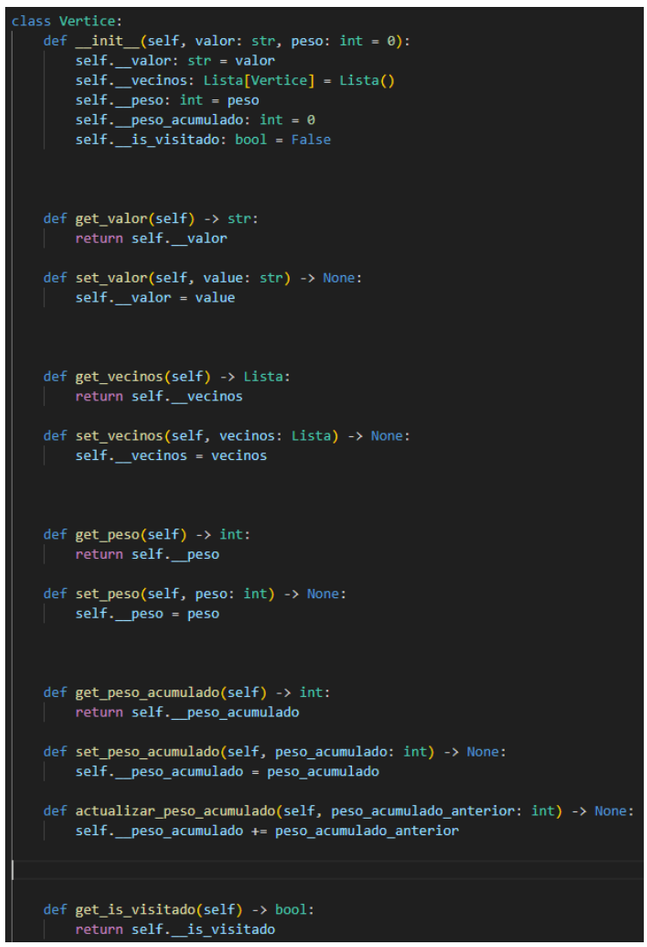
- **NodoArbolB.py:** Es la definición de la Clase NodoArbolB en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un nodo dentro de la estructura de Árbol B.

```python
class NodoArbolB:
    def __init__(self, is_hoja: bool = False):
        self.__is_hoja: bool = is_hoja
        self.__claves: list[Vehiculo] = []  # (m - 1)      m=orden_arbol
        self.__hijos: list[NodoArbolB] = []   # m     m=orden_arbol



    def get_is_hoja(self) -> bool:
        return self.__is_hoja

    def set_is_hoja(self, is_hoja: bool) -> None:
        self.__is_hoja = is_hoja



    def get_claves(self) -> list[Vehiculo]:
        return self.__claves

    def set_calves(self, claves: list[Vehiculo]) -> None:
        self.__claves = claves



    def get_hijos(self):
        return self.__hijos

    def set_hijos(self, hijos) -> None:
        self.__hijos = hijos
```

- **NodoLista.py:** Es la definición de la Clase NodoLista en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un nodo dentro de la estructura de Lista.

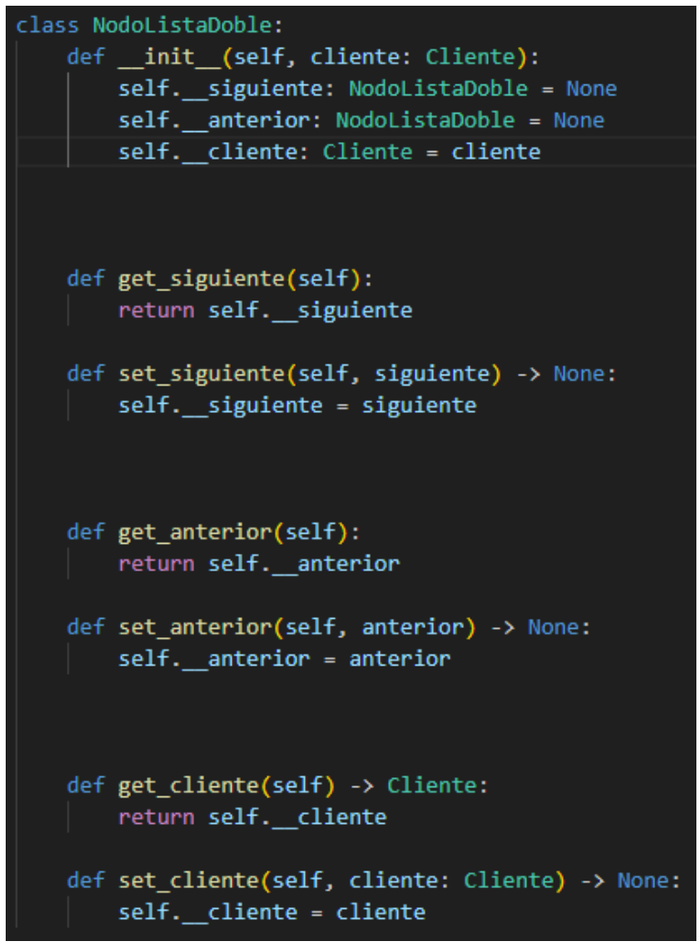
```python
class NodoLista:
    def __init__(self, valor_vertice):
        from ..Vertice import Vertice
        self.__valor_vertice: Vertice = valor_vertice
        self.__siguiente: NodoLista = None



    def get_valor_vertice(self):
        return self.__valor_vertice

    def set_valor_vertice(self, valor_vertice) -> None:
        self.__valor_vertice = valor_vertice



    def get_siguiente(self):
        return self.__siguiente

    def set_siguiente(self, siguiente) -> None:
        self.__siguiente = siguiente
```

- **Vertice.py:** Es la definición de la Clase Vértice en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un nodo dentro de la estructura de Grafo.



```python
class Vertice:
    def __init__(self, valor: str, peso: int = 0):
        self.__valor: str = valor
        self.__vecinos: Lista[Vertice] = Lista()
        self.__peso: int = peso
        self.__peso_acumulado: int = 0
        self.__is_visitado: bool = False



    def get_valor(self) -> str:
        return self.__valor

    def set_valor(self, value: str) -> None:
        self.__valor = value



    def get_vecinos(self) -> Lista:
        return self.__vecinos

    def set_vecinos(self, vecinos: Lista) -> None:
        self.__vecinos = vecinos



    def get_peso(self) -> int:
        return self.__peso

    def set_peso(self, peso: int) -> None:
        self.__peso = peso



    def get_peso_acumulado(self) -> int:
        return self.__peso_acumulado

    def set_peso_acumulado(self, peso_acumulado: int) -> None:
        self.__peso_acumulado = peso_acumulado

    def actualizar_peso_acumulado(self, peso_acumulado_anterior: int) -> None:
        self.__peso_acumulado += peso_acumulado_anterior


    def get_is_visitado(self) -> bool:
        return self.__is_visitado
```

- **NodoListaDoble.py:** Es la definición de la Clase NodoListaDoble en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un nodo dentro de la estructura de Lista Doble.



```python
class NodoListaDoble:
    def __init__(self, cliente: Cliente):
        self.__siguiente: NodoListaDoble = None
        self.__anterior: NodoListaDoble = None
        self.__cliente: Cliente = cliente



    def get_siguiente(self):
        return self.__siguiente

    def set_siguiente(self, siguiente) -> None:
        self.__siguiente = siguiente



    def get_anterior(self):
        return self.__anterior

    def set_anterior(self, anterior) -> None:
        self.__anterior = anterior



    def get_cliente(self) -> Cliente:
        return self.__cliente

    def set_cliente(self, cliente: Cliente) -> None:
        self.__cliente = cliente
```

- **NodoListaSimple.py:** Es la definición de la Clase NodoListaSimple en donde no tiene nada de especial con respecto a los métodos, pero que es la representación de un nodo dentro de la estructura de Lista Simple.

```python
class NodoListaSimple:
    def __init__(self, viaje: Viaje):
        self.__siguiente: NodoListaSimple = None
        self.__viaje: Viaje = viaje




    def get_siguiente(self) -> NodoListaSimple:
        return self.__siguiente

    def set_siguiente(self, siguiente: NodoListaSimple) -> None:
        self.__siguiente = siguiente



    def get_viaje(self) -> Viaje:
        return self.__viaje

    def set_viaje(self, viaje: Viaje) -> None:
        self.__viaje = viaje
```

DESCRIPCION DE TDAs

Lista Circular Doblemente Enlazada

- Descripción

Es una estructura donde el último elemento tiene como referencia siguiente al primer elemento, y la referencia al anterior del primer elemento de la lista también es el último. Es una especia de lista doblemente enlazada pero que posee una característica adicional para el desplazamiento dentro de la lista “esta no tiene fin” y tiene 2 apuntadores a si misma.

A través del uso de la lista doble podemos acceder a los datos recorriéndolos hacia adelante hasta el final o hacia atrás hasta el inicio. En una lista enlazada doblemente circular, cada nodo tiene dos enlaces, similares a los de la lista doblemente enlazada, excepto que el enlace anterior del primer nodo apunta al último y el enlace siguiente del último nodo, apunta al primero. Como en una lista doblemente enlazada, las inserciones y eliminaciones pueden ser hechas desde cualquier punto con acceso a algún nodo cercano.

Aunque estructuralmente una lista circular doblemente enlazada no tiene ni principio ni fin, un puntero de acceso externo puede establecer el nodo apuntador que está en la cabeza y así mantener el orden tan bien como en una lista doblemente enlazada. En las listas circulares dobles, nunca se llega a una posición en la que ya no sea posible desplazarse. Cuando se llega al último elemento, el desplazamiento volverá a comenzar desde el primer elemento.

- Operaciones Básicas
    - Insertar: El primer paso es crear un nodo para el dato que vamos a insertar. Si la lista está vacía, o el valor del primer elemento de la lista es mayor que el del nuevo, insertamos el nuevo nodo en la primera posición de la lista. En caso contrario, buscamos el lugar adecuado para la inserción, tenemos un puntero “anterior”. Lo inicializamos con el valor de la lista, y avanzaremos mientras anterior -> siguiente no sea nullptr y el dato que contiene anterior -> siguiente sea menor o igual que el dato que queremos insertar. Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

- Buscar: A la hora de buscar elementos en una lista circular solo hay que tener una precaución, es necesario almacenar el puntero del nodo en que se empezó a la búsqueda, para poder detectar el caso en que no exista el valor que se busca. Por lo demás, a búsqueda es igual que en el caso de las listas abiertas, salvo que podemos empezar en cualquier punto de la lista.

Árbol B

- Descripción

El árbol B es un tipo especial de árbol de búsqueda auto equilibrado en el que cada nodo puede contener más de una clave y puede tener más de dos hijos. Es una forma generalizada del árbol de búsqueda binario. También se le conoce como árbol m-way de altura equilibrada. La necesidad de B-tree surgió cuando la necesidad de un menor tiempo para acceder a medios de almacenamiento físicos como un disco duro. Los dispositivos de almacenamiento secundario son más lentos y tienen mayor capacidad.

Había una necesidad de este tipo de estructuras de datos que minimizo el acceso al disco. Otras estructuras de datos, como arboles binarios de búsqueda, arboles AVL, arboles rojo-negro, etc., pueden almacenar solo una clave en un nodo. Si se debe almacenar una gran cantidad de claves, la altura de dichos árboles se vuelve muy grande y el tiempo de acceso aumenta. Sin embargo, el árbol B puede almacenar muchas claves en un sol nodo y puede tener varios nodos secundarios. Esto reduce la altura significativamente, lo que permite accesos más rápidos al disco.

- Operaciones Básicas
  - Insertar: La inserción de un elemento en un árbol B consta de dos eventos: buscar el nodo apropiado para insertar el elemento y dividir el nodo si es necesario. La operación de inserción siempre se lleva a cabo con el enfoque de abajo hacia arriba. Veamos a continuación estos acontecimientos:
    1. Si el árbol esta vacío, asigne un nodo raíz e inserte la clave
    2. Actualice el numero permitido de claves en el nodo
    3. Busque el nodo apropiado para la inserción
    4. Si el nodo está lleno, siga los pasos a continuación.
    5. Insertar los elementos en orden creciente
    6. Ahora, hay elementos mayores que su límite. Por lo tanto, se divide en la mediana.
    7. Empuje la clave mediana hacia arriba y haga que las claves de la izquierda sean un hijo izquierdo y las claves de la derecha sea un hijo derecho.
    8. Si el nodo no está lleno, siga los pasos a continuación
    9. Inserte el nodo en orden creciente.
  - Eliminar: La eliminación de un elemento en un árbol B consta de tres eventos principales: buscar el nodo donde existe la clave que se va a eliminar, eliminar la clave y equilibrar el árbol si es necesario. Al eliminar de un árbol, puede producirse una situación denominada desbordamiento. El desbordamiento ocurre cuando un nodo contiene menos de la cantidad mínima de claves que debería contener. Los términos que se deben de comprender antes de estudiar la operaciones de eliminación son:
    1. Predecesor en orden: La clave más grandes en el hijo izquierdo de un nodo se denomina su predecesor en orden
    2. Sucesor en orden: La clave más pequeña en el hijo derecho de un un nodo se denomina su sucesor en orden
  - Búsqueda: La búsqueda de un elemento en un árbol B es la forma generalizada de buscar un elemento en un árbol binario de búsqueda. Se siguen los siguientes pasos:
    1. A partir del nodo raíz, compara k (clave de nodo) con la primera clave del nodo. Si k = clave_nodo, se devuelve el nodo y el índice (opcional)
    2. Si k no se encontró, y esta sobre un nodo hoja, se devuelve None (es decir, no encontrado)
    3. Si k < clave_nodo, busca el hijo izquierdo de esta clave de forma recursiva

4. Si hay más de una clave en el nodo actual y k > clave_nodo, compara k con la siguiente clave del nodo. Si k < clave_nodo, busca el hijo izquierdo de esta clave, de esta forma hasta llegar a la última clave de nodo, y buscar en todo caso en su hijo derecho.
5. Repita los pasos 1 a 4 hasta llegar a la hoja.

Grafo

- Descripción

Un grafo es un conjunto de vértices/nodos unidos por enlaces llamados aristas/arcos, que permiten presentar relaciones binarias entre elemento de un conjunto, típicamente, un grafo se representa gráficamente como un conjunto de puntos unidos por líneas. Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras.

Para nuestro caso hemos implementado el tipo de grafo no dirigido etiquetado, los grafos no dirigidos son aquellos que constan un conjunto de vértices que están conectados a un conjunto de aristas de forma no direccional. Esto significa que una arista puede indistintamente recorrerse desde cualquiera de sus puntos y en cualquier dirección. Los grafos etiquetados o con peso, son aquellos que concentran aristas que pueden poseer información adicional donde podemos reflejar costos, valores u otros datos. Estos grafos también son denominados como redes de actividad y el número asociado al arco, se le denomina factor de peso. Este grafo es el que más comúnmente utilizamos para representar situación de la vida real.