Nueva Guatemala de la Asunción

Reporte de Fase 1

Proyecto EDD

Elaborado por: Estuardo Gabriel Son Mux
Carné: 202003894
Fecha: 19/12/2021

Alcances y Objetivos del Programa

El programa realizado tiene como objetivo la implementación de los conocimientos adquiridos durante las primeras semanas del curso de Estructuras de Datos, realizando una página web en la que se incluyeron las estructuras: Lista doblemente enlazada, árbol de búsqueda binaria, árbol AVL y matriz dinámica, todas en el lenguaje de programación JavaScript. Dichas estructuras fueron utilizadas para el manejo de los datos de la aplicación los cuales se ingresarán por medio de una página web que se ejecuta en Github pages.

Especificaciones Técnicas

Requisitos de Hardware

- Computadora con todos sus componentes para su correcto funcionamiento
- Laptop

Requisitos de Software

Sistema Operativo con el que se Realizo

El programa fue desarrollado en una laptop con sistema operativo Windows 10, 8GB de ram, Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz 1.70 GHz.

Nombre del dispositivo	DESKTOP-LJSJP7U
Procesador	Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz 1.70 GHz
RAM instalada	8.00 GB
Id. del dispositivo	515F0098-1032-4937-BDB1-C191D818D118

Lenguaje de Programación

JavaScript

Librerías

- Vis-Network
- CirculaJSON

IDE o Editor de Código

Visual Estudio

Lógica del Programa

Archivos Utilizados

La página web consta de varios archivos de los cuales se dividen entre el lenguaje de programación JavaScript y el lenguaje de etiquetado HTML. Sin embargo solo se hará mención de las estructuras utilizadas para la elaboración del proyecto que se encuentran en los archivos JS.

Archivos JavaScricpt (.js)

Arbol.js

En este archivo se encuentran las clases utilizadas para la creación del árbol de búsqueda binaria para los proveedores.

Clase nodoArbol

```
class nodoArbol{
    constructor(id, usuario, correo, direccion, telefono){
        this.objeto = new Proveedor(id, usuario, correo, direccion, telefono);
        this.izquierda = null;
        this.derecha = null;
    }
```

La primera clase que se presenta en dicho archivo es la clase nodoArbol con ella se crean los nodos que se ingresaran al árbol de búsqueda binaria para la formación de las ramas. Cuenta con los atributos:

- **Objeto:** Aquí se almacena un objeto del tipo proveedor del cual se hablará más adelante.
- **Izquierda:** Sirve para guardar un objeto del tipo nodoArbol.
- **Derecha:** Sirve para guardar un objeto del tipo nodoArbol.

Clase Arbol

Está clase cuenta con dos atributos:

- **raiz:** En este atributo se guarda el primer nodo del árbol.
- **Dot:** En este atributo se guarda el string generado por la función graficar.

Así mismo, cuenta con nueve métodos o funciones:

- **Insertar(id, usuario, correo, dirección, telefono):**

```
//Método para insertar datos al Arbol
insertar(id, usuario, correo, direccion, telefono){
    let nuevo = new nodoArbol(id, usuario, correo, direccion, telefono);
    if (this.raiz == null){
        this.raiz = nuevo;
        console.log("Se registro el Proveedor");
    }
    else{
       this.raiz = this.insertarAux(nuevo,this.raiz);
    }
}
```

  Este método recibe los parámetros id, usuario, correo, dirección y teléfono. Es el encargado de realizar la inserción de nodos en caso que el parámetro raíz este vacío, en caso contrario realiza el llamado a la función recursiva insertarAux.

- **InsertarAux(nuevo, padre):**

```
//Método auxiliar para insertar en caso que el arbol ya tenga un nodo
insertarAux(nuevo, padre){
    if (padre==null){
        padre = nuevo;
        console.log("Se registro el Proveedor");
        return padre;
    }
    else{
        if(nuevo.objeto.id > padre.objeto.id){
            padre.derecha = this.insertarAux(nuevo, padre.derecha);
        }
        else if(nuevo.objeto.id < padre.objeto.id){
            padre.izquierda = this.insertarAux(nuevo, padre.izquierda);
        }
        else{
            console.log("El dato ya existe");
        }
        return padre;
    }
}
```

Esta función recibe los parámetros nuevo y padre, los cuales serán nodos. El parámetro padre será el nodo actual en el cual se encuentra realizando comparaciones, el parámetro nuevo es el nodo que se desea ingresar al árbol.

- **Grafica(padre):**

```
//Metodo para la creacion del DOT
graficar(padre){
    if (padre != null){
        this.dot += padre.objeto.id+' [label="'+padre.objeto.id+' '+padre.objeto.usuario+' '+padre.objeto.correo+'"];';
        if (padre.izquierda != null){
            this.dot += padre.objeto.id+'->'+padre.izquierda.objeto.id+';';
        }
        if (padre.derecha != null){
            this.dot += padre.objeto.id+'->'+padre.derecha.objeto.id+';';
        }
        this.graficar(padre.izquierda);
        this.graficar(padre.derecha);
    }
    return this.dot;
}
```

Esta funcione recursiva recibe el parámetro padre el cual será la raíz del árbol binario de búsqueda, al finalizar retornara un String el cual tendrá la sintaxis correcta para la elaboración del árbol binario con la herramienta vis-network.

- **buscarE(dato):**

```
//Metodo de eliminacion en caso que el nodo a eliminar se encuentre en la raiz
buscarE(dato){
    if (dato==this.raiz.objeto.id){
        let masDerecha = this.masDerecha(this.raiz.izquierda);
        console.log(masDerecha);
        let masIzquierda = this.masIzquierda(this.raiz.derecha);
        console.log(masIzquierda)

        if(masDerecha == null && masIzquierda ==null){
            this.raiz = masDerecha;
        }
        else if(masIzquierda !=null){
            try{
                this.raiz.derecha = this.masIzquierdaRomperConexion(this.raiz.derecha,masIzquierda);
                if(this.raiz.derecha != null){
                    masIzquierda.derecha = this.raiz.derecha;
                }
                masIzquierda.izquierda = this.raiz.izquierda;
            }catch(e){
                console.log(e);
            }
            this.raiz = masIzquierda;
        }
        else{
            try{
                this.raiz.izquierda = this.masDerechaRomperConexion(this.raiz.izquierda,masDerecha);
                if(this.raiz.izquierda != null){
                    masDerecha.izquierda = this.raiz.izquierda;
                }
                masDerecha.derecha = this.raiz.derecha;
            }catch(e){
                console.log(e);
            }
            this.raiz = masDerecha;
        }
    }else{
        this.raiz = this.buscarEAux(dato, this.raiz);
    }
}
```

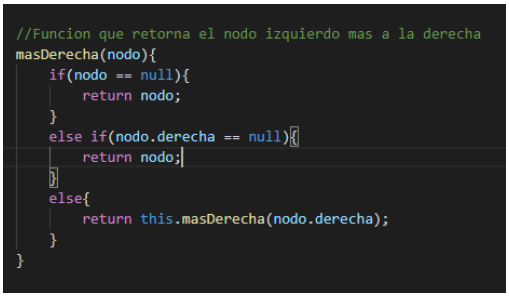
Este método es el encargado de eliminar el nodo cuyo id concuerde con el parámetro dato, enfocándose en el caso en el que la raíz sea el nodo a eliminar. Hace uso de las funciones masDerecha, masIzquierda, masIzquierdaRomperConexion y masDerechaRomperConexion.

- **buscaEAux(dato, padre):**

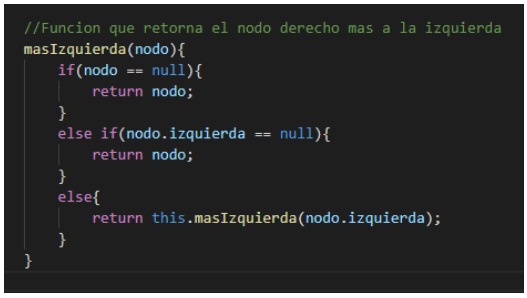
```
//Metodo de eliminacion en caso que el nodo a eliminar no se encuentre en la raiz
buscarEAux(dato, padre){
    if (padre==null){
        console.log("No se encontro ninguna conincidencia");
        return padre;
    }
    else if (dato > padre.objeto.id){
        padre.derecha = this.buscarEAux(dato, padre.derecha);
    }
    else if (dato < padre.objeto.id){
        padre.izquierda = this.buscarEAux(dato, padre.izquierda);
    }
    else if (dato == padre.objeto.id){
        let masDerecha = this.masDerecha(padre.izquierda);
        console.log(masDerecha);
        let masIzquierda = this.masIzquierda(padre.derecha);
        console.log(masIzquierda)


        if(masDerecha == null && masIzquierda ==null){
            return masDerecha;
        }
        else if(masIzquierda !=null){
            try{
                padre.derecha = this.masIzquierdaRomperConexion(padre.derecha,masIzquierda);
                if (padre.derecha != null){
                    masIzquierda.derecha = padre.derecha;
                }
                masIzquierda.izquierda = padre.izquierda;
            }catch(e){
                console.log(e);
            }
            return masIzquierda;
        }
        else{
            try{
                padre.izquierda = this.masDerechaRomperConexion(padre.izquierda,masDerecha);
                masDerecha.derecha = padre.derecha;
                if (padre.izquierda != null){
                 masDerecha.izquierda = padre.izquierda;
                }
            }catch(e){
                console.log(e);
            }
            return masDerecha;
        }
    }
    return padre;
```

Función complementaria a buscarE la cual contiene las condiciones necesarias para eliminar un nodo del árbol que no sea el nodo raíz, haciendo uso del parámetro dato que recibe el id del nodo a eliminar; y el parámetro padre que recibe el nodo padre de la iteración que se está realizando. Hace uso de las funciones masDerecha, masIzquierda, masIzquierdaRomperConexion y masDerechaRomperConexion.

- **masDerecha(nodo) y masIzquierda(nodo):**



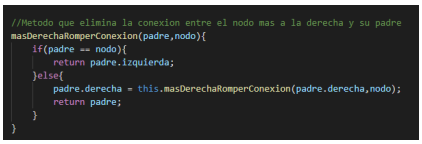
```
//Funcion que retorna el nodo izquierdo mas a la derecha
masDerecha(nodo){
    if(nodo == null){
        return nodo;
    }
    else if(nodo.derecha == null){
        return nodo;
    }
    else{
        return this.masDerecha(nodo.derecha);
    }
}
```

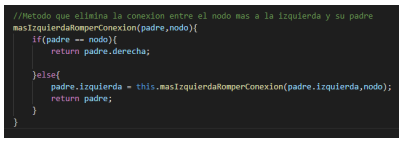


```
//Funcion que retorna el nodo derecho mas a la izquierda
masIzquierda(nodo){
    if(nodo == null){
        return nodo;
    }
    else if(nodo.izquierda == null){
        return nodo;
    }
    else{
        return this.masIzquierda(nodo.izquierda);
    }
}
```

Son funciones recursivas muy similares que reciben el parámetro nodo, el cual es el nodo izquierdo o derecho respectivamente del nodo que se desea encontrar el izquierdo más a la derecha o el nodo derecho más a la izquierda respectivamente.

- **masDerechaRomperConexion(padre, nodo) y masIzquierdaRomperConexion(padre, nodo):**



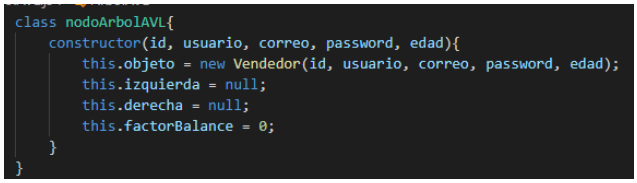


Ambas funciones son recursivas y retornan el parámetro padre con las modificaciones respectivas, la cual es romper la conexión entre el nodo izquierdo más a la derecha y el derecho más a la izquierda respectivamente.

ArbolAVL.js

En este archivo se almacenan las clases para la elaboración de la estructura de árbol AVL el cual se utiliza para el almacenamiento de vendedores.

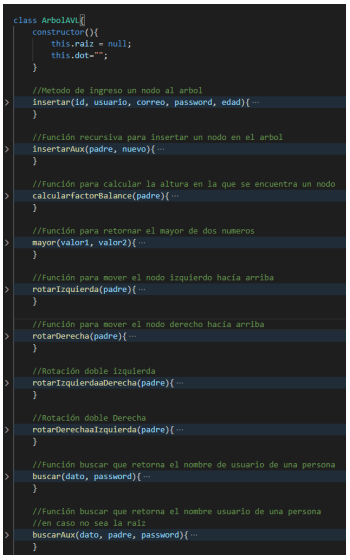
Clase nodoArbolAVL

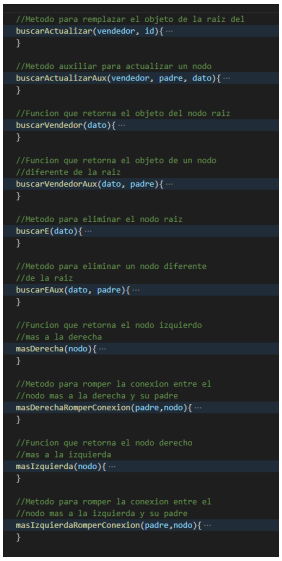


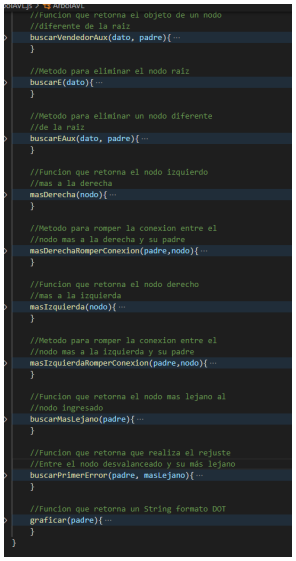
Con esta clase se crean los nodos que formarán parte de las ramas que se generarán al ingresar datos al árbol AVL. Posee los atributos:

- **objeto:** Almacena el objeto de tipo vendedor el cual contendrá la información del vendedor añadido.
- **Izquierda:** Conexión con el nodo hijo izquierdo.
- **Derecha:** Conexión con el nodo hijo derecho.

Clase ArbolAVL





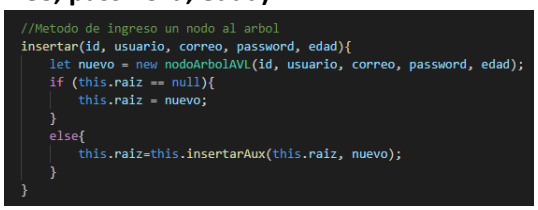


Esta clase se utiliza para la creación de la estructura de árbol AVL el cual es muy semejante al árbol de búsqueda binaría, sin embargo, tiene la característica de balancear sus ramas evitando que cada uno de sus nodos posea un factor de balanceo diferente de -1, 0 o 1. Está clase cuenta con dos atributos:

- **raiz:** En este atributo se guarda el primer nodo del árbol.
- **Dot:** En este atributo se guarda el string generado por la función graficar.

Así mismo, cuenta con veintitrés métodos o funciones:

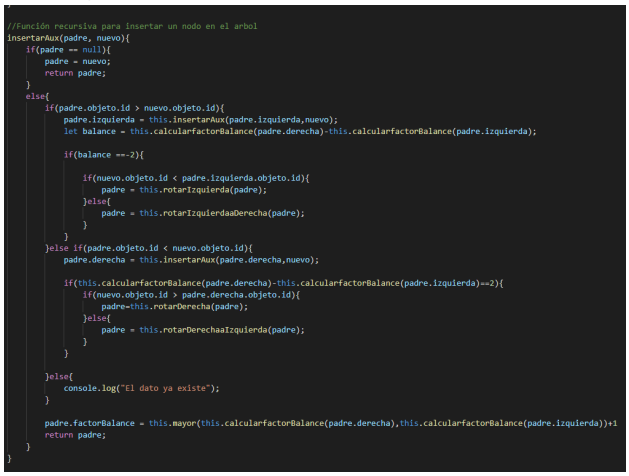
- **Insertar(id, usuario, correo, password, edad):**



```
//Metodo de ingreso un nodo al arbol
insertar(id, usuario, correo, password, edad){
    let nuevo = new nodoArbolAVL(id, usuario, correo, password, edad);
    if (this.raiz == null){
        this.raiz = nuevo;
    }
    else{
        this.raiz=this.insertarAux(this.raiz, nuevo);
    }
}
```

Este método se utiliza para la inserción de un nodo dentro del árbol AVL enfocándose en el caso en que se deba insertar en la raíz. En caso contrario utilizara la función insertarAux.

- **insertarAux(padre, nuevo):**



```
//Función recursiva para insertar un nodo en el arbol
insertarAux(padre, nuevo){
    if(padre == null){
        padre = nuevo;
        return padre;
    }
    else{
        if(padre.objeto.id > nuevo.objeto.id){
            padre.izquierda = this.insertarAux(padre.izquierda,nuevo);
            let balance = this.calcularfactorBalance(padre.derecha)-this.calcularfactorBalance(padre.izquierda);

            if(balance ==-2){

                if(nuevo.objeto.id < padre.izquierda.objeto.id){
                    padre = this.rotarIzquierda(padre);
                }else{
                    padre = this.rotarIzquierdaaDerecha(padre);
                }
            }
        }else if(padre.objeto.id < nuevo.objeto.id){
            padre.derecha = this.insertarAux(padre.derecha,nuevo);

            if(this.calcularfactorBalance(padre.derecha)-this.calcularfactorBalance(padre.izquierda)==2){
                if(nuevo.objeto.id > padre.derecha.objeto.id){
                    padre=this.rotarDerecha(padre);
                }else{
                    padre = this.rotarDerechaaIzquierda(padre);
                }
            }

        }else{
            console.log("El dato ya existe");
        }

        padre.factorBalance = this.mayor(this.calcularfactorBalance(padre.derecha),this.calcularfactorBalance(padre.izquierda))+1
        return padre;
    }
}
```

Esta función recibe los parámetros nuevo y padre, los cuales serán nodos. El parámetro padre será el nodo actual en el cual se encuentra realizando comparaciones y el parámetro nuevo es el nodo que se desea ingresar al árbol. Pero a diferencia de la función ingresar del ABB el árbol AVL utiliza las funciones: mayor, calcularFactorBalance, rotarIzquierda, rotarDerecha, rotarIzquierdaaDerecha y rotarDerechaaIzquierda; las cuales son funciones que permiten hacer el cambio de posición de los nodos ingresados para balancear la estructura del árbol. En caso que el nodo a ingresar sea mayor que el nodo al que se esta comparando se ejecutaran las funciones rotarDerecha o rotarDerechaaIzquierda, en caso contrario, se ejecutaran las funciones rotarIzquierda o rotarIzquierdaaDerecha, según sea necesario.

- **calcularfactorBalance(padre):**

```
//Función para calcular la altura en la que se encuentra un nodo
calcularfactorBalance(padre){
    if(padre == null){
        return -1;
    }else{
        return padre.factorBalance;
    }
}
```

Función que recibe el parámetro padre que será el nodo al cual se desea calcular su factor de Balance en caso no el nodo sea nulo retornará un -1, en caso contrario retornará el valor de factor de balance del nodo ingresado.

- **Mayor(valor1, valor2):**

```
//Función para retornar el mayor de dos numeros
mayor(valor1, valor2){
    if(valor1 > valor2){
        return valor1;
    }else{
        return valor2;
    }

}
```

Está función recibe los parámetros valor1 y valor2 los cuales serán comparados y retornara el valor que sea mayor.

- **rotarIzquierda(padre) y rotarDerecha(padre):**

```
//Función para mover el nodo izquierdo hacia arriba
rotarIzquierda(padre){
    let aux = padre.izquierda;
    padre.izquierda= aux.derecha;
    aux.derecha = padre;
    padre.factorBalance = this.mayor(this.calcularfactorBalance(padre.derecha),this.calcularfactorBalance(padre.izquierda)) + 1;
    aux.factorBalance = this.mayor(this.calcularfactorBalance(aux.derecha),this.calcularfactorBalance(aux.izquierda)) + 1;
    return aux;
}

//Función para mover el nodo derecho hacia arriba
rotarDerecha(padre){
    let aux = padre.derecha;
    padre.derecha= aux.izquierda;
    aux.izquierda = padre;
    padre.factorBalance = this.mayor(this.calcularfactorBalance(padre.izquierda),this.calcularfactorBalance(padre.derecha)) + 1;
    aux.factorBalance = this.mayor(this.calcularfactorBalance(aux.izquierda),this.calcularfactorBalance(aux.derecha)) + 1;
    return aux;
}
```

Estas funciones sirven para hacer las rotaciones elevando el nodo a la izquierda del padre y colocando al padre como su nodo derecho, y elevando el nodo a la derecha del padre y colocando al padre como su nodo izquierdo.

- **rotarIzquierdaaDerecha(padre) y rotarDerechaaIzquierda(padre):**

```
//Rotación doble izquierda
rotarIzquierdaaDerecha(padre){
    padre.izquierda = this.rotarDerecha(padre.izquierda);
    let aux = this.rotarIzquierda(padre);
    return aux;
}

//Rotación doble Derecha
rotarDerechaaIzquierda(padre){
    padre.derecha = this.rotarIzquierda(padre.derecha);
    let aux = this.rotarDerecha(padre);
    return aux;
}
```

Estas funciones son combinaciones de las funciones rotarIzquierda y rotarDerecha que dependen de la condición que el hijo izquierdo del padre sea menor al nodo nuevo o que el hijo derecho del padre sea mayor que el nodo ingresado.

- **Buscar(dato, password):**

```
//Función buscar que retorna el nombre de usuario de una persona
buscar(dato, password){
    if (dato==this.raiz.objeto.id && password == this.raiz.objeto.password){
        localStorage.setItem("actualVendedor", CircularJSON.stringify(this.raiz.objeto));
        return this.raiz.objeto.usuario;
    }else{
        return this.buscarAux(dato, this.raiz, password);
    }
}
```

Función que recibe los parámetros: dato, que almacena el id del objeto que se desea buscar; y password que almacena la contraseña del objeto que se desea buscar; dichos parámetros serán utilizados para realizar comparaciones en caso se haga una coincidencia en la raíz retornará el objeto del nodo raíz, en caso contrario, devolverá la función recursiva buscarAux.

- **buscarAux(dato, padre, password):**

```
//Función buscar que retorna el nombre usuario de una persona
//en caso no sea la raiz
buscarAux(dato, padre, password){

    console.log(dato)
    if (padre==null){
        return false;
    }
    else if (dato > padre.objeto.id){
        return this.buscarAux(dato, padre.derecha, password);
    }
    else if (dato < padre.objeto.id){
        return this.buscarAux(dato, padre.izquierda, password);
    }
    else if (dato == padre.objeto.id && password == padre.objeto.password){
        localStorage.setItem("actualVendedor", CircularJSON.stringify(padre.objeto));
        return true
    }
}
```

Función que retorna un True o False si los datos que se envían como parámetros coinciden con los datos de alguno de los nodos del árbol.

- **buscarActualizar(vendedor, id):**

```
//Metodo para remplazar el objeto de la raiz del
buscarActualizar(vendedor, id){
    if (id==this.raiz.objeto.id){
        this.raiz.objeto = vendedor
    }else{
        this.buscarActualizarAux(vendedor, this.raiz, id);
    }
}
```

Método para buscar el nodo con el objeto vendedor cuyo id sea, igual al parámetro id, posterior mente reemplazar dicho objeto por el objeto enviado por parámetro. Enfocándose principalmente en la raíz del árbol.

- **buscarActualizarAux(vendedor, padre, dato):**

```
//Metodo auxiliar para actualizar un nodo
buscarActualizarAux(vendedor, padre, dato){
    if (padre==null){
        console.log("No se actualizaron los datos");
    }
    else if (dato > padre.objeto.id){
        this.buscarActualizarAux(vendedor, padre.derecha, dato);
    }
    else if (dato < padre.objeto.id){
        this.buscarActualizarAux(vendedor, padre.izquierda, dato);
    }
    else if (dato == padre.objeto.id){
        padre.objeto = vendedor
    }
}
```

Método para buscar el nodo con el objeto vendedor cuyo id sea, igual al parámetro id, posterior mente reemplazar dicho objeto por el objeto enviado por parámetro. Enfocándose en los nodos diferentes la raíz del árbol.

- **buscarVendedor(dato) y buscarVendedorAux(dato, padre):**

```
//Funcion que retorna el objeto del nodo raiz
buscarVendedor(dato){
    if (dato==this.raiz.objeto.id){
        return this.raiz.objeto;
    }else{
        return this.buscarVendedorAux(dato, this.raiz);
    }
}
```

```
//Funcion que retorna el objeto de un nodo
//diferente de la raiz
buscarVendedorAux(dato, padre){
    if (padre==null){
        return null;
    }
    else if (dato > padre.objeto.id){
        return this.buscarVendedorAux(dato, padre.derecha);
    }
    else if (dato < padre.objeto.id){
        return this.buscarVendedorAux(dato, padre.izquierda);
    }
    else if (dato == padre.objeto.id){
        return padre.objeto;
    }
}
```

Funciones de búsqueda que retornan el objeto vendedor, del nodo cuyo objeto vendedor posea un atributo id de igual valor al parámetro dato.

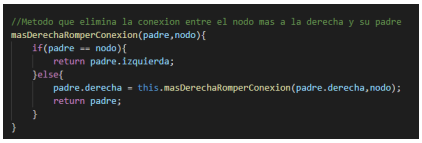
- **masDerecha(nodo) y masIzquierda(nodo):**

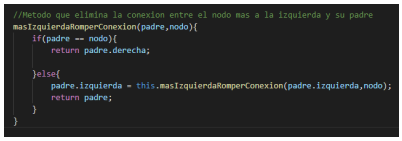
```
//Funcion que retorna el nodo izquierdo mas a la derecha
masDerecha(nodo){
    if(nodo == null){
        return nodo;
    }
    else if(nodo.derecha == null){
        return nodo;
    }
    else{
        return this.masDerecha(nodo.derecha);
    }
}
```

```
//Funcion que retorna el nodo derecho mas a la izquierda
masIzquierda(nodo){
    if(nodo == null){
        return nodo;
    }
    else if(nodo.izquierda == null){
        return nodo;
    }
    else{
        return this.masIzquierda(nodo.izquierda);
    }
}
```

Son funciones recursivas muy similares que reciben el parámetro nodo, el cual es el nodo izquierdo o derecho respectivamente del nodo que se desea encontrar el izquierdo más a la derecha o el nodo derecho más a la izquierda respectivamente.

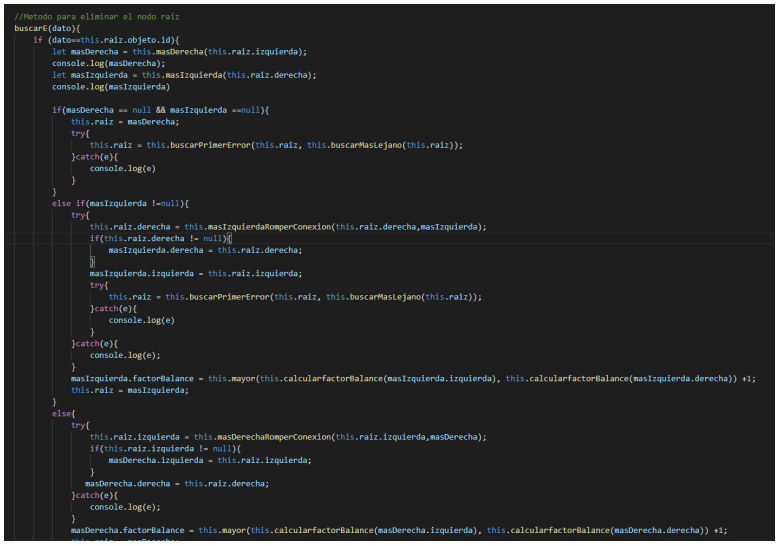
- **masDerechaRomperConexion(padre, nodo) y masIzquierdaRomperConexion(padre, nodo):**





Ambas funciones son recursivas y retornan el parámetro padre con las modificaciones respectivas, la cual es romper la conexión entre el nodo izquierdo más a la derecha y el derecho más a la izquierda respectivamente.

- **buscarE(dato) y buscarEAux(dato, padre):**



```
//Metodo para eliminar un nodo diferente
//de la raiz
buscarEAux(dato, padre){
    if (padre==null){
        console.log("No se encontro ninguna conincidencia");
        return padre;
    }
    else if (dato > padre.objeto.id){
        padre.derecha = this.buscarEAux(dato, padre.derecha);
    }
    else if (dato < padre.objeto.id){
        padre.izquierda = this.buscarEAux(dato, padre.izquierda);
    }
    else if (dato == padre.objeto.id){
        let masDerecha = this.masDerecha(padre.izquierda);
        let masIzquierda = this.masIzquierda(padre.derecha);

        if(masDerecha == null && masIzquierda ==null){
            return masDerecha;
        }
        else if(masIzquierda !=null){
            try{
                padre.derecha = this.masIzquierdaRomperConexion(padre.derecha,masIzquierda);
                if (padre.derecha != null){
                    masIzquierda.derecha = padre.derecha;
                }
                masIzquierda.izquierda = padre.izquierda;
            }catch(e){
                console.log(e);
            }
            masIzquierda.factorBalance = this.mayor(this.calcularfactorBalance(masIzquierda.izquierda), this.calcularfactorBalance(masIzquierda.derecha)) +1;
            try{
                masIzquierda = this.buscarPrimerError(masIzquierda, this.buscarMasLejano(masIzquierda));
            }catch(e){
                console.log(e)
            }
            return masIzquierda;
        }
        else{
            try{
                padre.izquierda = this.masDerechaRomperConexion(padre.izquierda,masDerecha);
                if (padre.izquierda != null){
                masDerecha.izquierda = padre.izquierda;
                }
                masDerecha.derecha = padre.derecha;
            }catch(e){
                console.log(e);
            }
```

Funciones se utilizan para la eliminación de nodos dentro del árbol AVL y necesitan de las diferentes funciones que se han descrito anteriormente como calcularFactorBalance o mayor, sin embargo, también necesita de funciones que se describirán más adelante como: buscarMasLejano y buscarPrimerError. Estás funciones realizan en primera instancia la búsqueda del nodo que concuerde con los datos introducidos en sus parámetros, una vez localizado realiza las mismas acciones que realiza el árbol de búsqueda binaría, pero añade dos pasos más al finalizar los mismo, los cuales se realizaran únicamente si el árbol se ha desbalanceado, en primer lugar se deberá crear buscar el nodo más lejano al nodo en que se encuentre ubicado y su factorBalance sea igual a 2 una vez encontrado dicho nodo se procederá a realizar las rotaciones respectivas para realizar el balanceo.

- **buscarMasLejano(padre):**

```
//Funcion que retorna el nodo mas lejano al
//nodo ingresado
buscarMasLejano(padre){
    if(padre.izquierda == null && padre.derecha == null){
        return padre;
    }
    else if (padre.izquierda==null){
        return this.buscarMasLejano(padre.derecha);
    }
    else if (padre.derecha==null){
        return this.buscarMasLejano(padre.izquierda);
    }
    else if (padre.izquierda.factorBalance >= padre.derecha.factorBalance){
        return this.buscarMasLejano(padre.izquierda);
    }
    else{
        return this.buscarMasLejano(padre.derecha);
    }
}
```

Esta función recibe el parámetro padre y realiza comparaciones en busca del nodo más lejano del nodo padre, es decir cuando los parámetros izquierda y derecha sean nulos, en caso contrario continuara con la recursividad con el nodo que tenga el mayor factorBalance.

- **buscarPrimerError(padre, masLejano):**

```
//Funcion que retorna que realiza el rejuste
//Entre el nodo desvalanceado y su más lejano
buscarPrimerError(padre, masLejano){
    if (masLejano.objeto.id > padre.objeto.id){
        if ((this.calcularfactorBalance(padre.derecha) - (this.calcularfactorBalance(padre.izquierda)))==2){
            if(masLejano.objeto.id > padre.derecha.objeto.id){
                padre = this.rotarDerecha(padre);
            }else{
                padre = this.rotarDerechaaIzquierda(padre);
            }
        }
    }
    if (masLejano.objeto.id < padre.objeto.id){
        if((this.calcularfactorBalance(padre.derecha) - this.calcularfactorBalance(padre.izquierda))==-2){
            if(masLejano.objeto.id < padre.izquierda.objeto.id){
                console.log("giro")
                padre = this.rotarIzquierda(padre);
                console.log(padre)
            }else{
                console.log("doble giro")
                padre = this.rotarIzquierdaaDerecha(padre);
            }
        }
    }
    else{
        if ((this.calcularfactorBalance(padre.derecha) - this.calcularfactorBalance(padre.izquierda))<-2){
            padre = this.buscarPrimerError(padre.izquierda, masLejano);
        }
        else if ((this.calcularfactorBalance(padre.derecha) - this.calcularfactorBalance(padre.izquierda))>2){
            padre = this.buscarPrimerError(padre.derecha, masLejano);
        }
    }
    return padre;
}
```

Función que realiza las rotaciones necesarias para el balance del árbol AVL posterior a una eliminación recibiendo el parámetro padre el cual si posee un factorBalance igual a 2 o -2 entonces realizara las rotaciones pertinentes dependiendo si el nodo a la izquierda del padre es mayor al nodo más lejano o si el nodo a la derecha del padre es menor al nodo más lejano.

- **Graficar(padre):**

```
//Funcion que retorna un String formato DOT
graficar(padre){
    if (padre != null){
        this.dot += padre.objeto.id+' [label="'+padre.objeto.id+' '+padre.objeto.usuario+' '+padre.objeto.correo+' '+padre.objeto.password+' '+padre.factorBalance+'"];';

        if (padre.izquierda != null){
            this.dot += padre.objeto.id+'->'+padre.izquierda.objeto.id+';';
        }
        if (padre.derecha != null){
            this.dot += padre.objeto.id+'->'+padre.derecha.objeto.id+';';
        }
        this.graficar(padre.izquierda);
        this.graficar(padre.derecha);
    }
    return this.dot;
}
```

Función que recibe el parámetro padre el cual debe ser la raíz del árbol, mediante la recursividad la función recorre cada uno de los nodos y concatena una serie de Strings para retronarlos con un formato DOT.

Matriz.js

Este archivo contiene las clases necesarias para la elaboración de la matriz dinámica.

Clase nodoCabecera

```
class nodoCabecera{
    constructor(valor){
        this.valor = valor;
        this.longitud = 0;
        this.derecha = null;
        this.izquierda = null;
        this.arriba = null;
        this.abajo = null;
    }
}
```

Esta clase cuenta con los atributos:

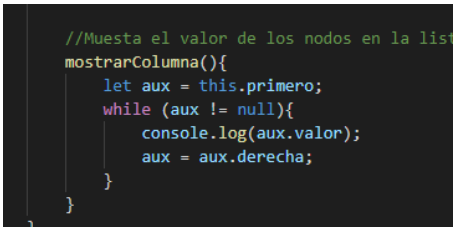
- **Valor:** Almacena el valor que servirá como identificador del nodo.
- **Longitud:** Almacena la cantidad de nodos que se encuentran enlazados al este nodo formando una lista.
- **Derecha, izquierda, arriba y abajo:** Conexión con los nodos sucesores o antecesores.

Clase listaCabeera

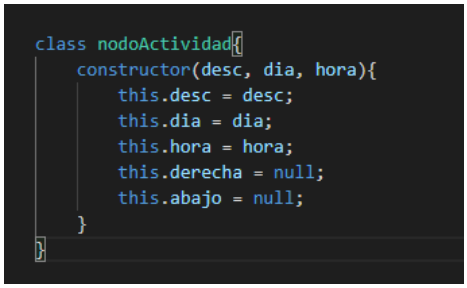
Esta clase cuenta con los atributos:

- **Primero:** Apuntador que contiene el primer nodo de la lista de cabeceras.
- **Ultimo:** Apuntador que contiene el ultimo nodo de la lista de cabeceras.

Esta clase cuenta con un único método que imprime en consola el valor identificador del nodo en el que se encuentra:



Clase nodoActividad

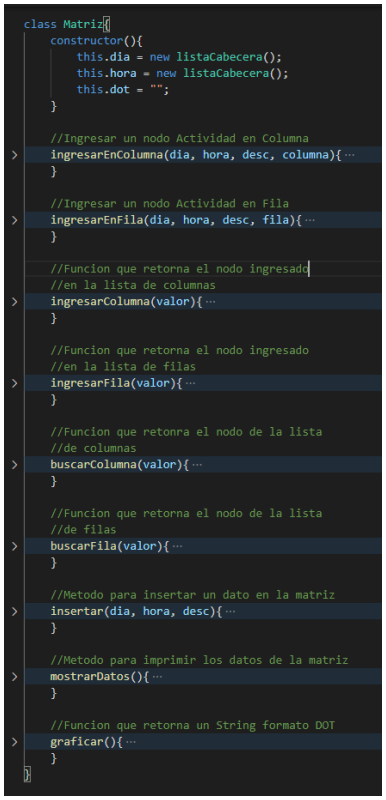


```
class nodoActividad{
    constructor(desc, dia, hora){
        this.desc = desc;
        this.dia = dia;
        this.hora = hora;
        this.derecha = null;
        this.abajo = null;
    }
}
```

Esta clase se utiliza para la creación de los nodos internos de la matriz dinámica, posee cinco atributos:

- **Desc:** Almacena la descripción de la actividad.
- **Dia:** Almacena el número de día de la actividad (columna).
- **Hora:** Almacena el número de hora de la actividad (fila).
- **Derecha, abajo:** Apuntadores a los nodos posteriores.

Clase Matriz



```
class Matriz{
    constructor(){
        this.dia = new listaCabecera();
        this.hora = new listaCabecera();
        this.dot = "";
    }

    //Ingresar un nodo Actividad en Columna
    ingresarEnColumna(dia, hora, desc, columna){ ···
    }

    //Ingresar un nodo Actividad en Fila
    ingresarEnFila(dia, hora, desc, fila){ ···
    }

    //Funcion que retorna el nodo ingresado
    //en la lista de columnas
    ingresarColumna(valor){ ···
    }

    //Funcion que retorna el nodo ingresado
    //en la lista de filas
    ingresarFila(valor){ ···
    }

    //Funcion que retonra el nodo de la lista
    //de columnas
    buscarColumna(valor){ ···
    }

    //Funcion que retorna el nodo de la lista
    //de filas
    buscarFila(valor){ ···
    }

    //Metodo para insertar un dato en la matriz
    insertar(dia, hora, desc){ ···
    }

    //Metodo para imprimir los datos de la matriz
    mostrarDatos(){ ···
    }

    //Funcion que retorna un String formato DOT
    graficar(){ ···
    }
}
```

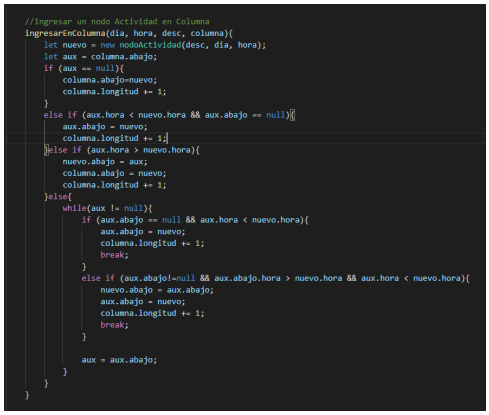
Esta es la clase en donde se juntan los nodos, posee 3 atributos y 9 funciones o métodos.

Sus atributos son:

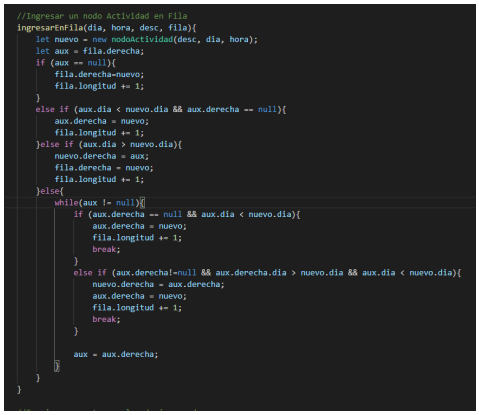
- **Dia:** Lista que almacena los nodos de cabecera que formaran las columnas.
- **Hora:** Lista que almacena los nodos de cabecera que forman las filas.
- **Dot:** Variable string donde se concatenarán los datos para la generación del DOT.

Listado de métodos y funciones:

- **ingresarEnColumna(dia, hora, desc, columna) e ingresarEnFila(dia, hora, desc, fila):**



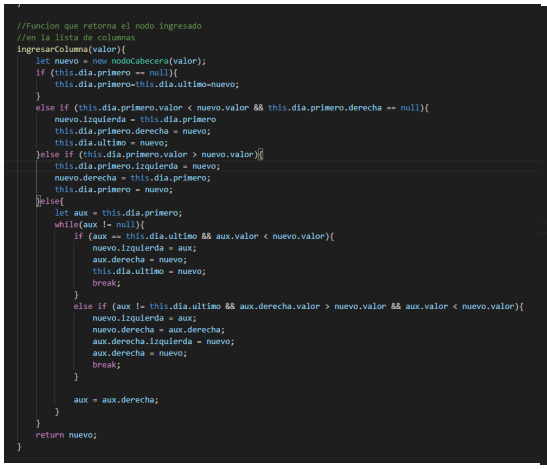
```
//Ingresar un nodo Actividad en Columna
ingresarEnColumna(dia, hora, desc, columna){
    let nuevo = new nodoActividad(desc, dia, hora);
    let aux = columna.abajo;
    if (aux == null){
        columna.abajo=nuevo;
        columna.longitud += 1;
    }
    else if (aux.hora < nuevo.hora && aux.abajo == null){
        aux.abajo = nuevo;
        columna.longitud += 1;
    }else if (aux.hora > nuevo.hora){
        nuevo.abajo = aux;
        columna.abajo = nuevo;
        columna.longitud += 1;
    }else{
        while(aux != null){
            if (aux.abajo == null && aux.hora < nuevo.hora){
                aux.abajo = nuevo;
                columna.longitud += 1;
                break;
            }
            else if (aux.abajo!=null && aux.abajo.hora > nuevo.hora && aux.hora < nuevo.hora){
                nuevo.abajo = aux.abajo;
                aux.abajo = nuevo;
                columna.longitud += 1;
                break;
            }

            aux = aux.abajo;
        }
    }
}
```



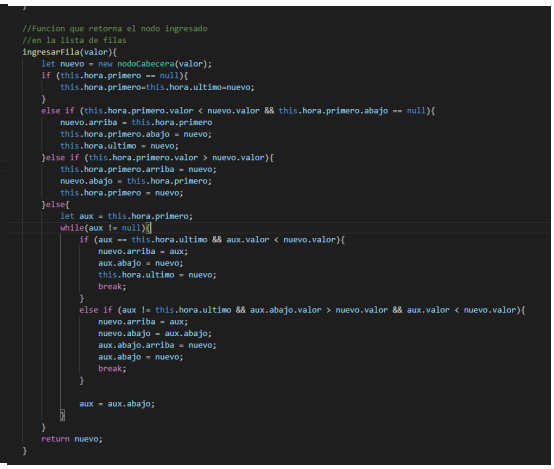
```
//Ingresar un nodo Actividad en Fila
ingresarEnFila(dia, hora, desc, fila){
    let nuevo = new nodoActividad(desc, dia, hora);
    let aux = fila.derecha;
    if (aux == null){
        fila.derecha=nuevo;
        fila.longitud += 1;
    }
    else if (aux.dia < nuevo.dia && aux.derecha == null){
        aux.derecha = nuevo;
        fila.longitud += 1;
    }else if (aux.dia > nuevo.dia){
        nuevo.derecha = aux;
        fila.derecha = nuevo;
        fila.longitud += 1;
    }else{
        while(aux != null){
            if (aux.derecha == null && aux.dia < nuevo.dia){
                aux.derecha = nuevo;
                fila.longitud += 1;
                break;
            }
            else if (aux.derecha!=null && aux.derecha.dia > nuevo.dia && aux.dia < nuevo.dia){
                nuevo.derecha = aux.derecha;
                aux.derecha = nuevo;
                fila.longitud += 1;
                break;
            }

            aux = aux.derecha;
        }
    }
}
```

Son métodos que ingresar nodosActividad dentro de la matriz, los parámetros dia y hora servirán para ubicarlos en sus respectivas filas de día y hora, el parámetro desc es la descripción para la actividad ingresada, por su parte el parámetro columna o fila es el encabezado en dónde se ingresara el nodo.

- **IngresarFila(valor) e IngresarColumna(valor):**



```
//Funcion que retorna el nodo ingresado
//en la lista de columnas
ingresarColumna(valor){
    let nuevo = new nodoCabecera(valor);
    if (this.dia.primero == null){
        this.dia.primero=this.dia.ultimo=nuevo;
    }
    else if (this.dia.primero.valor < nuevo.valor && this.dia.primero.derecha == null){
        nuevo.izquierda = this.dia.primero
        this.dia.primero.derecha = nuevo;
        this.dia.ultimo = nuevo;
    }else if (this.dia.primero.valor > nuevo.valor){
        this.dia.primero.izquierda = nuevo;
        nuevo.derecha = this.dia.primero;
        this.dia.primero = nuevo;
    }else{
        let aux = this.dia.primero;
        while(aux != null){
            if (aux == this.dia.ultimo && aux.valor < nuevo.valor){
                nuevo.izquierda = aux;
                aux.derecha = nuevo;
                this.dia.ultimo = nuevo;
                break;
            }
            else if (aux != this.dia.ultimo && aux.derecha.valor > nuevo.valor && aux.valor < nuevo.valor){
                nuevo.izquierda = aux;
                nuevo.derecha = aux.derecha;
                aux.derecha.izquierda = nuevo;
                aux.derecha = nuevo;
                break;
            }

            aux = aux.derecha;
        }
    }
    return nuevo;
}
```



```
//Funcion que retorna el nodo ingresado
//en la lista de filas
ingresarFila(valor){
    let nuevo = new nodoCabecera(valor);
    if (this.hora.primero == null){
        this.hora.primero=this.hora.ultimo=nuevo;
    }
    else if (this.hora.primero.valor < nuevo.valor && this.hora.primero.abajo == null){
        nuevo.arriba = this.hora.primero
        this.hora.primero.abajo = nuevo;
        this.hora.ultimo = nuevo;
    }else if (this.hora.primero.valor > nuevo.valor){
        this.hora.primero.arriba = nuevo;
        nuevo.abajo = this.hora.primero;
        this.hora.primero = nuevo;
    }else{
        let aux = this.hora.primero;
        while(aux != null){
            if (aux == this.hora.ultimo && aux.valor < nuevo.valor){
                nuevo.arriba = aux;
                aux.abajo = nuevo;
                this.hora.ultimo = nuevo;
                break;
            }
            else if (aux != this.hora.ultimo && aux.abajo.valor > nuevo.valor && aux.valor < nuevo.valor){
                nuevo.arriba = aux;
                nuevo.abajo = aux.abajo;
                aux.abajo.arriba = nuevo;
                aux.abajo = nuevo;
                break;
            }

            aux = aux.abajo;
        }
    }
    return nuevo;
}
```

Son métodos que se utilizan para ingresar nuevos nodos a las listas de dia y hora (columna y fila), las cuales se ingresan en un orden ascendente y en caso de haber un dato repetido el mismo se descarta.

- **buscarFila(valor) y buscarColumna(valor):**

```
//Funcion que retonra el nodo de la lista
//de columnas
buscarColumna(valor){
    let aux = this.dia.primero;
    try {
        while (aux != null){
            if (aux.valor == valor){
                return aux;
            }

            aux = aux.derecha;
        }
        return aux;
    } catch (error) {
        return aux;
    }
}

//Funcion que retorna el nodo de la lista
//de filas
buscarFila(valor){
    let aux = this.hora.primero;
    while (aux != null){
        if (aux.valor == valor){
            return aux;
        }

        aux = aux.abajo;
    }
    return aux;
}
```

Son funciones que retornan el nodo de las listas dia y hora, los cuales concuerden con el parámetro valor. Si al terminar la búsqueda no se encontró ninguna coincidencia entonces retornara el valor nulo.

- **Insertar(dia, hora, desc):**

```
//Metodo para insertar un dato en la matriz
insertar(dia, hora, desc){
    let comprobarC = this.buscarColumna(dia);
    let comprobarF = this.buscarFila(hora);
    if (comprobarC != null){
        this.ingresarEnColumna(dia, hora, desc, comprobarC);
    }else{
        this.ingresarEnColumna(dia, hora, desc, this.ingresarColumna(dia));
    }
    if (comprobarF != null){
        this.ingresarEnFila(dia, hora, desc, comprobarF);
    }else{
        this.ingresarEnFila(dia, hora, desc, this.ingresarFila(hora));
    }
}
```

Método encargado de insertar nodos, tanto en la lista de día y hora como dentro de la matriz haciendo llamado a los métodos ingresarColumna, ingresarFila, ingresarEnColumna e ingresarEnFila, y realizando las respectivas comprobaciones de la existencia de los nodos con las funciones buscarColumna y buscarFila.

- **mostrarDatos():**

```
//Metodo para imprimir los datos de la matriz
mostrarDatos(){
    let aux = this.hora.primero;
    while(aux != null){
        let tmp = aux.derecha;
        while (tmp != null){
            console.log(tmp.dia+" "+tmp.hora+" "+tmp.desc+" ");
            tmp = tmp.derecha;
        }
        aux = aux.abajo;
    }
}

//Funcion que retorna un String formato DOT
```

Método que recorre la matriz fila por fila imprimiendo en consola cada uno de los valores enconetrados.

- **Graficar():**

```
//Funcion que retorna un String formato DOT
graficar(){
    let aux = this.dia.primero;
    if(aux!=null){
        this.dot += 'calendario [pos="0,0!"];'
        this.dot += 'calendario->d'+aux.valor+';';
    }
    while(aux != null){
        this.dot += 'd'+aux.valor+' [label="Dia: '+aux.valor+' " pos="'+aux.valor+',0!"];'
        let tmp = aux.abajo;
        if (aux.derecha != null){
            this.dot += 'd'+aux.valor+'->d'+aux.derecha.valor+';';
            this.dot += 'd'+aux.derecha.valor+'->d'+aux.valor+';';
        }
        if (tmp!=null){
            this.dot += 'd'+aux.valor+'->d'+tmp.dia+'h'+tmp.hora+';';
            this.dot += 'd'+tmp.dia+'h'+tmp.hora+'->d'+aux.valor+';';
            while (tmp != null){
                this.dot += 'd'+tmp.dia+'h'+tmp.hora+' [label="'+tmp.desc+'"  pos="'+tmp.dia+','+tmp.hora+'!"];'
                if (tmp.abajo!=null){
                    this.dot += 'd'+tmp.dia+'h'+tmp.hora+'->d'+tmp.abajo.dia+'h'+tmp.abajo.hora+';';
                    this.dot += 'd'+tmp.abajo.dia+'h'+tmp.abajo.hora+'->d'+tmp.dia+'h'+tmp.hora+';';
                }
                tmp = tmp.abajo
            }
        }
            aux = aux.derecha;
    }

    let aux1 = this.hora.primero;
    if(aux1!=null){
        this.dot += 'calendario->h'+aux1.valor+';';
    }
    while(aux1 != null){
        this.dot += 'h'+aux1.valor+' [label="hora: '+aux1.valor+'" pos="0,'+aux1.valor+'!"];'
        let tmp = aux1.derecha;
        if (aux1.abajo != null){
            this.dot += 'h'+aux1.valor+'->h'+aux1.abajo.valor+';';
            this.dot += 'h'+aux1.abajo.valor+'->h'+aux1.valor+';';
        }
        if (tmp!=null){
            this.dot += 'h'+aux1.valor+'->d'+tmp.dia+'h'+tmp.hora+';';
            this.dot += 'd'+tmp.dia+'h'+tmp.hora+'->h'+aux1.valor+';';
            while (tmp != null){
                if (tmp.derecha!=null){
                    this.dot += 'd'+tmp.dia+'h'+tmp.hora+'->d'+tmp.derecha.dia+'h'+tmp.derecha.hora+';';
                    this.dot += 'd'+tmp.derecha.dia+'h'+tmp.derecha.hora+'->d'+tmp.dia+'h'+tmp.hora+';';
                }
```

Es una función que retorna un String que contiene un formato DOT para realizar la gráfica de la matriz dinámica.

ListaDoble.js

En este archivo se encuentran las clases para la formación de la estructura Lista Doblemente Enlazada.

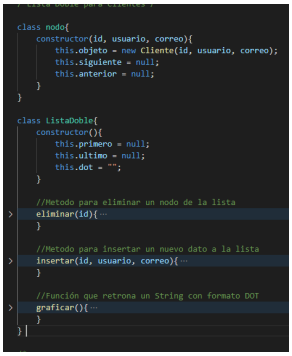
Clase nodo

```
class nodo{
    constructor(id, usuario, correo){
        this.objeto = new Cliente(id, usuario, correo);
        this.siguiente = null;
        this.anterior = null;
    }
}
```

Esta clase se utiliza para la creación de los nodos que pasaran a formar parte de la lista.

- **Objeto:** Almacena el objeto que contendrá el nodo.
- **Siguiente y anterior:** Son apuntadores a los nodos siguientes o anteriores al nodo creado.

Clase ListaDoble

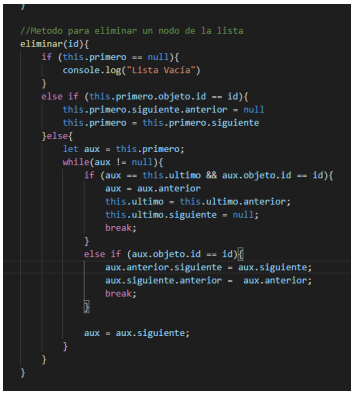


Esta clase se encarga del procesamiento de los nodos para la formación de la lista doblemente enlazada. Posee los atributos:

- **Primero:** Apuntador al primer nodo de la lista.
- **Ultimo:** Apuntador al ultimo nodo de la lista.
- **Dot:** Variable String que almacenara formato dot.

Métodos o funciones:

- **Eliminar(id):**



  Método encargado de eliminar un nodo perteneciente a la lista, cuyo objeto posea un id de igual valor al parámetro id.

- **Insertar(id, usuario, correo):**

```
//Metodo para insertar un nuevo dato a la lista
insertar(id, usuario, correo){
    let nuevo = new nodo(id, usuario, correo);
    if (this.primero == null){
        this.primero=this.ultimo=nuevo;
    }
    else if (this.primero.objeto.id < nuevo.objeto.id && this.primero.siguiente == null){
        nuevo.anterior = this.primero;
        this.primero.siguiente = nuevo;
        this.ultimo = nuevo;
    }else if (this.primero.objeto.id > nuevo.objeto.id){
        this.primero.anterior = nuevo;
        nuevo.siguiente = this.primero;
        this.primero = nuevo;
    }else{
        let aux = this.primero;
        while(aux != null){
            if (aux == this.ultimo && aux.objeto.id < nuevo.objeto.id){
                nuevo.anterior = aux;
                aux.siguiente = nuevo;
                this.ultimo = nuevo;
                break;
            }
            else if (aux != this.ultimo && aux.siguiente.objeto.id > nuevo.objeto.id && aux.objeto.id < nuevo.objeto.id){
                nuevo.siguiente = aux.siguiente;
                nuevo.anterior = aux;
                nuevo.siguiente.anterior = nuevo;
                aux.siguiente = nuevo;
                break;
            }

            aux = aux.siguiente;
        }
    }
}
```

Método que se encarga de insertar un nodo con los valores de parámetro dentro de una la lista de forma ascendente, en caso que encuentre una coincidencia de id el dato se descartara.

- **Graficar():**

```
//Función que retrona un String con formato DOT
graficar(){
    let aux = this.primero;
    while (aux != null){
        this.dot += aux.objeto.id+'[label="'+aux.objeto.id+' '+aux.objeto.usuario+' '+aux.objeto.correo+'"];';
        if (aux.siguiente != null){
            this.dot += aux.objeto.id+"->"+aux.siguiente.objeto.id+";";
            this.dot += aux.siguiente.objeto.id+"->"+aux.objeto.id+";";
        }
        aux = aux.siguiente;
    }
    return this.dot
}
}
```

Función que retorna un String con formato dot, recorriendo cada uno de los nodos y concatenándolos, para ser graficado.