Universidad de San Carlos de Guatemala

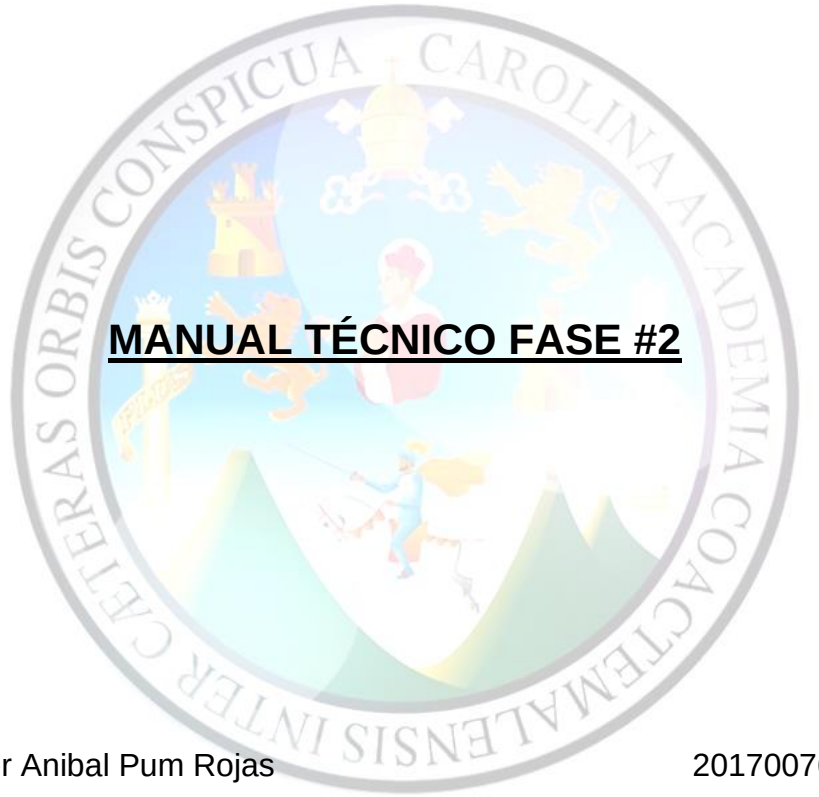
Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Estructuras de Datos

Ing. Luis Espino

Aux. Josué Pérez



MANUAL TÉCNICO FASE #2

Elder Anibal Pum Rojas 201700761

Guatemala, 4 de Enero del 2022

INTRODUCCIÓN

A la fase anterior se le agregaran funcionalidades, haciendo uso de estructuras de datos avanzadas con las cuales poder manejar la información necesaria para cada funcionalidad. El lenguaje utilizado seguirá siendo JavaScript por lo que todas las EDD's deben ser desarrolladas en este lenguaje y la aplicación debe estar publicada en GitHub Pages.

Seguridad

Para la seguridad se plantea la posibilidad de encriptar la información más sensible de los usuarios de la aplicación (vendedores), esto para evitar el robo de la información asociada a los servicios que se brindan en el sistema, para poder visualizarla el usuario que trate de hacerlo deberá contar con una llave que le permita el acceso a la misma.

Esto será implementado de la siguiente manera, para la encriptación de contraseñas se debe realizar utilizando "Sha256", esto posibilitará que se encripte las contraseñas, correo y cualquier otra información que considere sensible, de una manera segura.

Posible sugerencia puede ser con la librería sjcl para java, pero el uso de las librerías se deja libre para esta parte de la encriptación. La forma de proceder es que será solicitado en el usuario administrador la contraseña maestra para la encriptación y esta contraseña será la utilizada al momento de realizar dicha encriptación, esto puede ser recibido en una entrada de texto en el Frontend.

Inventario

Para esta fase, se agregará el módulo de manejo de inventario, este será manejado atravesó de un Árbol B de orden 5. En esta estructura se deben almacenar los productos que el administrador ingrese por medio de carga masiva o de manera manual.

Los campos utilizados serán los siguientes:

- ID
- Nombre
- Precio
- Cantidad

Registro de Ventas

Para el registro de ventas, se utilizará una tabla hash general, donde se almacenarán las ventas de todos los vendedores.

Para esta estructura se utilizará una tabla hash con los siguientes parámetros:

Para calcular las claves, la función hash a utilizar debe ser el método de división. El tipo de hash será cerrado y el método para resolver colisiones debe ser exploración cuadrática.

El tamaño inicial de la tabla hash será de 7 y cuando se llegue a un 50% de uso, se tendrá que hacer rehash de la tabla, el tamaño debe crecer al número primo próximo.

La información que debe almacenar cada nodo en la tabla hash es la siguiente:

- ID Venta
- Nombre Vendedor
- Nombre Cliente
- Total de la Venta (Calculado de la lista de productos)
- Lista enlazada de productos (productos registrados en el árbol B)

La tabla hash será manejada por hashsing cerrado, las listas mostradas en la imagen son el atributo de los productos de cada venta.

Blockchain

Esta estructura se utilizará para almacenar todas las transacciones de ventas realizadas. Este registro deberá realizarse cada cierto periodo de tiempo, este tiempo lo podrá cambiar el administrador desde la vista de configuración, pero inicialmente el tiempo será de 5 min.

Creación de bloque:

Una vez cumplido el tiempo configurado, la aplicación deberá crear un bloque de blockchain, donde se guarden las transacciones realizadas en el periodo de tiempo definido, si no hubiera transacciones de debe crear el bloque “vacío”.

Cada bloque deberá almacenar la siguiente información:

- Índice: el número correlativo del bloque, el bloque inicial tendrá el valor 0.
- Fecha: Almacena el momento exacto en el que se genera el bloque, debe de tener el siguiente formato (DD-MM-YY::HH:MM::SS).
- Data: Deben ser todas las somarizaciones de las transacciones realizadas.
- Nonce: Es un número entero que se debe iterar de uno en uno hasta encontrar un hash válido, por defecto el número de ceros al inicio del hash debe de ser de 4 ceros (0000) pero este valor debe poder cambiarse desde el panel de administrador.
- PreviousHash: Es el hash del bloque anterior, nos permite validar que la cadena del bloque no esté alterada. Para el primer bloque el hash anterior debe ser 0000.
- Hash: Es el hash del bloque actual.

Crear el hash:

Para crear el hash de este bloque se debe hacer uso del algoritmo SHA256, utilizando las propiedades listadas anteriormente. Todos estos bloques deben ir como cadenas concatenadas sin espacios en blanco:

SHA256(Indice + Fecha + PreviousHash + Data + Nonce)

Para encontrar un hash válido para nuestro bloque, debemos tomar en cuenta lo mencionado en la prueba de trabajo.

Prueba de trabajo:

Es un sistema con el fin de dificultar el proceso de generación de hashes válidos, para evitar comportamientos indeseados como ataques o spam. Para esto se debe iterar el campo Nonce hasta encontrar un hash válido para el bloque.

Para que el hash cumpla con la condición de que contenga una cantidad de ceros al inicio, por defecto el número de ceros al inicio del hash debe de ser de 4 ceros (0000) pero este valor debe poder cambiarse desde el panel de administrador.

Guardar bloque:

Luego de terminar el proceso de crear un nuevo bloque, se procede a almacenar el archivo en la carpeta de bloques.

Para generar el bloque el estudiante debe generar un archivo en el cual guardará la información necesaria para que, al ser leída, la aplicación siga teniendo los mismos datos que tenía al momento de cerrarse o ser interrumpida.

Estos bloques no son legibles para personas que tienen la intención de vulnerar el sistema o manipular los datos.

Grafo

Para esta fase, se implementa un módulo de rutas de las bodegas de los proveedores, esto para administrar de mejor manera el recurso de transportes de las empresas.

Este módulo nos permitirá ingresar todas las bodegas de los proveedores con las distancias entre ella y calcular la ruta optima de los camiones de la empresa para hacer un recorrido para recoger productos.

Para esto se utilizará la estructura de datos de Grafo ponderado.

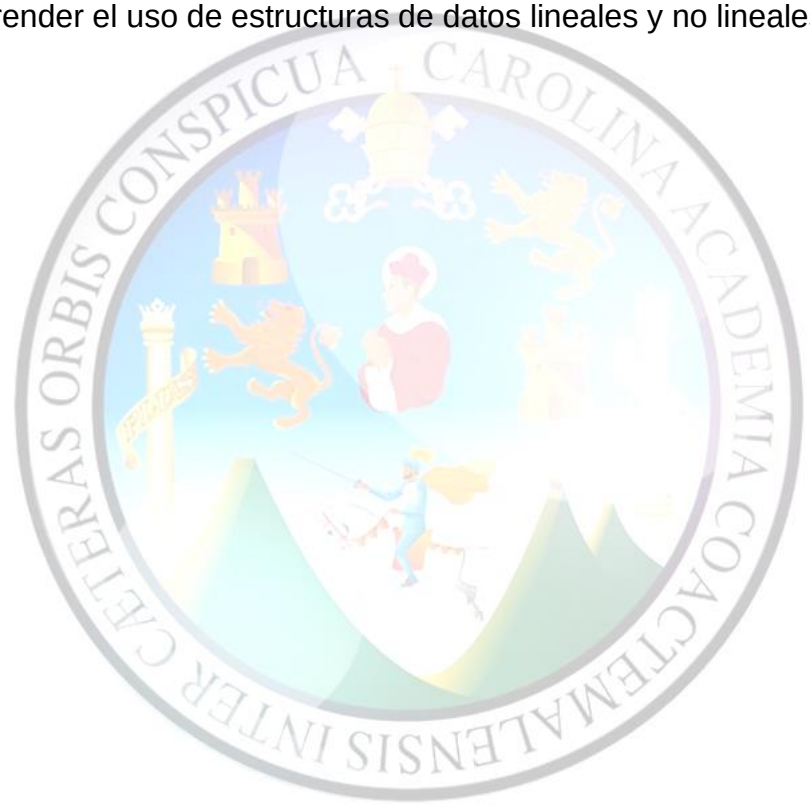
OBJETIVOS

Objetivos Generales

- Aplicar los conocimientos adquiridos en el curso de Lenguajes Formales y de Programación en el desarrollo de la aplicación

Objetivos Específicos

- Familiarizarse con el lenguaje de programación JavaScript
- Envolverse en el ámbito del manejo de memoria
- Familiarizarse con el uso de Git
- Familiarizarse con el manejo de lectura de archivos
- Comprender el uso de estructuras de datos lineales y no lineales



ESPECIFICACIÓN TÉCNICA

Requisitos de Hardware

- RAM: 128 MB Mínimo
- Procesador: Mínimo Pentium 2 a 266 Mhz

Requisitos de Software

- Sistema Operativo:
  - MS Windows XP o superior
  - Apple OSX 10.4.X o superior
  - GNU/LINUX 2.6.X o superior
- Exploradores Web:
  - Google Chrome (en todas las plataformas)
  - Mozilla Firefox (en todas las plataformas)
  - Internet Explorer (Windows)
  - Microsoft Edge (Windows, Android, iOS, Windows 10 móvil)
  - Safari (Mac, iOS)
  - Opera (Mac, Windows)
- Lenguaje de Programación o IDE:
  - Para el desarrollo de este software se utilizó JavaScript y como entorno se utilizó Visual Studio Code
- Tecnologías Aplicadas
  - JavaScript
  - HTML
  - CSS
  - Bootstrap
  - Graphviz

EXPLICACIÓN GENERAL

Inventario

Para la implementación del inventario o manejo de productos en general se utilizó un árbol B de grado 5, el cual contiene los datos del ID del producto, nombre del producto, precio y cantidad, por lo cual se implementó el nodo que se ve en la siguiente imagen:

```
class Nodo {
    constructor (id, nombre, precio, cantidad) {
        this.id = id
        this.nombre = nombre
        this.precio = precio
        this.cantidad = cantidad
        //Apuntadores de lista de tipo Nodo
        this.siguiente = null
        this.anterior = null
        //Apuntadores de arbol tipo pagina
        this.izquierda = null
        this.derecha = null
    }
}
```

Luego se procedió a construir el Árbol B utilizando este nodo, por lo cual fue necesario el uso de un método de Insertar y en el constructor de la clase tenemos atributos de “primero” y “último” para tener referencias a la lista que se crea en cada nodo del árbol B. Tal y como se muestra a continuación:

```
//Lista ordenada para almacenar valores
class ListaNodo {
    constructor() {
        this.primero = null
        this.ultimo = null
        this.size = 0
    }
```

```
insertar(nuevo) {
    if (this.primero == null) {
        this.primero = nuevo
        this.ultimo = nuevo
        this.size++
        return true
    } else {
        if (this.primero == this.ultimo) { //Solo hay un valor en la lista
            if (nuevo.id < this.primero.id) {
                nuevo.siguiente = this.primero
                this.primero.anterior = nuevo
                //Cambiar punteros de paginas
                this.primero.izquierda = nuevo.derecha

                this.primero = nuevo
                this.size++
                return true
            } else if (nuevo.id > this.ultimo.id) {
                this.ultimo.siguiente = nuevo
                nuevo.anterior = this.ultimo
                //Cambiar punteros de paginas
                this.ultimo.derecha = nuevo.izquierda

                this.ultimo = nuevo
                this.size++
                return true
            } else { //El id es igual al primero
                console.log("Ya existe el id con ese valor en la lista")
                return false
            }
```

```
    } else { //hay mas de un dato
        if (nuevo.id < this.primero.id) {
            nuevo.siguiente = this.primero
            this.primero.anterior = nuevo
            //Cambiar punteros de paginas
            this.primero.izquierda = nuevo.derecha

            this.primero = nuevo
            this.size++
            return true
        } else if (nuevo.id > this.ultimo.id) {
            this.ultimo.siguiente = nuevo
            nuevo.anterior = this.ultimo
            //Cambiar punteros de paginas
            this.ultimo.derecha = nuevo.izquierda

            this.ultimo = nuevo
            this.size++
            return true
```

```
            return true
        } else {
            let aux = this.primero
            while (aux != null) {
                if (nuevo.id < aux.id) {
                    nuevo.siguiente = aux
                    nuevo.anterior = aux.anterior
                    //cambiar punteros de paginas
                    aux.izquierda = nuevo.derecha
                    aux.anterior.derecha = nuevo.izquierda
                    ////////////////////////////////////
                    aux.anterior.siguiente = nuevo
                    aux.anterior = nuevo
                    this.size++
                    return true
                } else if (nuevo.id == aux.id) {
                    console.log("Ya existe un id con ese valor en la lista")
                    return false
                } else {
                    aux = aux.siguiente
                }
            }
        }
    }
```

Posteriormente tenemos los métodos para la paginación del árbol B, como se muestra a continuación:

```
//Pagina del arbol B

class pagina {
    constructor() {
        this.raiz = false
        this.clavesMax = 4
        this.clavesMin = 2
        this.size = 0
        this.claves = new ListaNodo()
    }

    insertarEnPagina(nodo) {
        if (this.claves.insertar(nodo)) {
            this.size = this.claves.size
            if (this.size < 5) {
                return this
            } else if (this.size == 5) { //Dividir pagina
                return this.dividirPagina(this)
            }
        }
        return null
    }
```

Se implementó una función para Insertar y otra para Dividir cada página, como se muestra a continuación:

```
dividirPagina(pag) {
    let temp = pag.claves.primero
    for (var i = 0; i < 2; i++) { //Ubicarnos en la posicion [2] de la lista (mitad)
        temp = temp.siguiente
    }

    //Pasar los valores de la pagina a nodos independientes
    let primero = pag.claves.primero
    let segundo = pag.claves.primero.siguiente
    let tercero = temp.siguiente
    let cuarto = pag.claves.ultimo

    primero.siguiente = null
    primero.anterior = null

    segundo.siguiente = null
    segundo.anterior = null

    tercero.siguiente = null
    tercero.anterior = null

    cuarto.siguiente = null
    cuarto.anterior = null

    temp.siguiente = null
    temp.anterior = null

    //Crear nuevas paginas
    let pagIzquierda = new pagina()
    pagIzquierda.insertarEnPagina(primero)
    pagIzquierda.insertarEnPagina(segundo)
```

```
        //Crear nuevas paginas
        let pagIzquierda = new pagina()
        pagIzquierda.insertarEnPagina(primero)
        pagIzquierda.insertarEnPagina(segundo)

        let pagDerecha = new pagina()
        pagDerecha.insertarEnPagina(tercero)
        pagDerecha.insertarEnPagina(cuarto)

        temp.izquierda = pagIzquierda
        temp.derecha = pagDerecha

        return temp
    }

    esHoja(pag) {
        if (pag.claves.primero.izquierda == null) {
            return true
        } else {
            return false
        }
    }
}
```

Posteriormente tenemos el método del árbol B completo, que nos permite hacer todas las funcionalidades que hemos hecho tanto en el Nodo como en la página del árbol B. En la estructura del árbol B tenemos un valor llamado “raíz” que nos sirve para saber cuál es el primer valor que se agrega al nodo, así como un orden, el cual por defecto es 5 y la altura que por defecto es 0, tal y como se muestra en la siguiente imagen:

```
//Arbol B

class ArbolB {
    constructor() {
        this.raiz = null
        this.orden = 5
        this.altura = 0
    }
```

Posteriormente tenemos el método de insertarNodo, el cual obtiene cada uno de los valores que ya establecimos para los productos y así se va desglosando el método utilizando otros métodos contenidos en el Nodo o en la página, ya sea para insertar o dividir dicha página, como se muestra a continuación:

```
insertarNodo(id, nombre, precio, cantidad) {
    let nuevo = new Nodo(id, nombre, precio, cantidad)

    if (this.raiz == null) {
        this.raiz = new pagina()
        this.raiz.raiz = true
        this.raiz = this.raiz.insertarEnPagina(nuevo)
    } else {
        if (this.altura == 0) {
            let respuesta = this.raiz.insertarEnPagina(nuevo)
            if (respuesta instanceof pagina) { //La raiz no se dividio
                this.raiz = respuesta
            } else {
                this.altura++
                this.raiz = new pagina()
                this.raiz = this.raiz.insertarEnPagina(respuesta)
            }
        } else { //Ya existe mas de una pagina, hay que recorrer el arbol para insertar el nuevo
            if (this.raiz == null) {
                console.log("La raiz es null")
                return
            }
            let respuesta = this.insertarRecorrer(nuevo, this.raiz)
            if (respuesta instanceof Nodo) { //La raiz no se dividio
                this.altura++
                this.raiz = new pagina()
                this.raiz = this.raiz.insertarEnPagina(respuesta)
            } else if (respuesta instanceof pagina) {
                this.raiz = respuesta
            }
        }
    }
}
```

Después tenemos un método para recorrer todo nuestro árbol e ir insertando en cada página, así como para dividir incluso y nos permite saber cómo se va moviendo nuestro árbol, tal y como se muestra en la siguiente imagen:

```
insertarRecorrer(nuevo, paginaActual) {
    if (paginaActual.esHoja(paginaActual)) {
        let respuesta = paginaActual.insertarEnPagina(nuevo)
        return respuesta
    } else {
        if (nuevo.id < paginaActual.claves.primero.id) { //Va a la izquierda
            let respuesta = this.insertarRecorrer(nuevo, paginaActual.claves.primero.izquierda)
            if (respuesta instanceof Nodo) { //La pagina se divdio y el nodo se tiene que insertar en la pagina actual
                return paginaActual.insertarEnPagina(respuesta)
            } else if (respuesta instanceof pagina) {
                paginaActual.claves.primero.izquierda = respuesta
                return paginaActual
            }
        } else if (nuevo.id > paginaActual.claves.ultimo.id) { //Va a la derecha porque es mayor al ultimo
            let respuesta = this.insertarRecorrer(nuevo, paginaActual.claves.ultimo.derecha)
            if (respuesta instanceof Nodo) {
                return paginaActual.insertarEnPagina(respuesta)
            } else if (respuesta instanceof pagina) {
                paginaActual.claves.ultimo.derecha = respuesta
                return paginaActual
            }
        } else { //En los apuntadores de los nodos de en medio
            let aux = paginaActual.claves.primero

            while (aux != null) {
                if (nuevo.id < aux.id) {
                    let respuesta = this.insertarRecorrer(nuevo, aux.izquierda)
                    if (respuesta instanceof Nodo) {
                        return paginaActual.insertarEnPagina(respuesta)
                    } else if (respuesta instanceof pagina) {
                        aux.izquierda = respuesta
                        aux.anterior.derecha = respuesta
                        return paginaActual
                    }
                } else if (nuevo.id == aux.id) {
                    return paginaActual
                } else {
                    aux = aux.siguiente
                }
            }
        }
    }
    return this
}
```

Después tenemos nuestros métodos para graficar:

```
graficar() {
    let cadena="digraph arbolB{\n";
    cadena+="rankr=TB;\n";
    cadena+="node[shape = box,fillcolor=\"azure2\" color=\"black\" style=\"filled\"];\n";
    //metodos para graficar el arbol
    cadena+= this.graficarNodos(this.raiz);
    cadena+=  this.graficarEnlaces(this.raiz);
    cadena+="}\n"

    const elemento = document.getElementById("contenido-dot")
    elemento.innerHTML = cadena
    console.log(cadena)

    return cadena;
}
```

```
    graficarNodos(raizActual) {
        let cadena = ""

        if (raizActual.esHoja(raizActual)) {
            cadena+="node[shape=record label= \"<p0>"
            let contador=0;
            let aux = raizActual.claves.primero;
            while(aux!=null){
                contador++;
                cadena+="|{"+aux.id+","+aux.nombre+","+aux.precio+","+aux.cantidad+"}|<p"+contador+"> ";
                aux= aux.siguiente;
            }
            cadena+="\"]"+raizActual.claves.primero.id+";\n";
            return cadena;
        }else{
            cadena+="node[shape=record label= \"<p0>"
            let contador=0;
            let aux = raizActual.claves.primero;
            while(aux!=null){
                contador++;
                cadena+="|{"+aux.id+","+aux.nombre+","+aux.precio+","+aux.cantidad+"}|<p"+contador+"> ";
                aux= aux.siguiente;
            }
            cadena+="\"]"+raizActual.claves.primero.id+";\n";

            //recorrer los hicos de cada clave
            aux = raizActual.claves.primero;
            while(aux != null){
                cadena+= this.graficarNodos(aux.izquierda);
                aux = aux.siguiente;
            }
            cadena+= this.graficarNodos(raizActual.claves.ultimo.derecha);
            return cadena;
        }
    }
```

```
    graficarEnlaces(raizActual){
        let cadena="";
        if(raizActual.esHoja(raizActual)){
            return ""+raizActual.claves.primero.id+";\n";
        }else{
            //cadena += ""+raiz_actual.claves.primero.dato+";\n";

            let aux = raizActual.claves.primero;
            let contador =0;
            let raiz_actual_txt = raizActual.claves.primero.id;
            while(aux != null){
                cadena+= "\n"+raiz_actual_txt+":p"+contador+"->"+this.graficarEnlaces(aux.izquierda);
                contador++;
                aux = aux.siguiente;
            }
            cadena+="\n"+raiz_actual_txt+":p"+contador+"->"+this.graficarEnlaces(raiz_actual.claves.ultimo.derecha);
            return cadena;
        }
    }
}
```

Grafo

Para la implementación del grafo ponderado se utilizó una estructura de grafo como tal, la cual contiene un Nodo y una lista de adyacentes, el cual el nodo contiene los datos del ID, nombre y los adyacentes contiene un id, nombre y una distancia, tal y como se muestra a continuación:

```
class Nodo {
    constructor(id, nombre) {
        this.id = id
        this.nombre = nombre
        this.anterior = null
        this.siguiente = null
        this.distancia = 0
        this.adyacentes = new listaAdyacentes()
    }
}
```

Después en nuestra clase de Adyacentes tenemos como valores el primero y el úttimo, esto para hacer referencia a una lista doblemente enlazada la cual nos permitira mantener el control del grafo como tal, para saber que valor es el primero y que valor es el útlimo de la lista, tal y como se muestra a continuación:

```
class listaAdyacentes {
    constructor() {
        this.primero = null
        this.ultimo = null
    }
```

Después viene nuestro método de insertar en la lista de adyacentes, el cual recibe un id y una distancia:

```
    insertar(id, p) {
        let nuevo = new Nodo(id)
        nuevo.distancia = p
        if (this.primero == null) {
            this.primero = nuevo
            this.ultimo = nuevo
        } else {
            if (this.primero == this.ultimo) {
                this.primero.siguiente = nuevo
                nuevo.anterior = this.primero
                this.ultimo = nuevo
            } else {
                nuevo.anterior = this.ultimo
                this.ultimo.siguiente = nuevo
                this.ultimo = nuevo
            }
        }
    }
}
```

Después viene nuestra clase grafo, la cual igualmente contiene un valor primero y un valor último:

```
class Grafo{
    constructor() {
        this.primero = null
        this.ultimo = null
    }
```

Igualmente posee un método de insertar, pero, además, también posee un método para buscar ID's y también para agregar adyacentes (muy útil a la hora de leer archivos en modalidad carga masiva), así como el método para mostrar en consola y los métodos de graficar, respectivamente:

```
insertar(id, nombre) {
    let nuevo = new Nodo(id, nombre)
    if (this.primero == null) {
        this.primero = nuevo
        this.ultimo = nuevo
    } else {
        if (this.primero == this.ultimo) {
            this.primero.siguiente = nuevo
            nuevo.anterior = this.primero
            this.ultimo = nuevo
        } else {
            nuevo.anterior = this.ultimo
            this.ultimo.siguiente = nuevo
            this.ultimo = nuevo
        }
    }
}
```

```
buscar(id) {
    let aux = this.primero
    while (aux != null) {
        if (aux.id == id) {
            return aux
        } else {
            aux = aux.siguiente
        }
    }
    return null
}
```

```
    agregarAdyacente(id, idAdyacente, distancia) {
        let principal = this.buscar(id)

        if (principal != null) {
            principal.adyacentes.insertar(idAdyacente, distancia)
        } else {
            console.log("No existe el nodo origen")
        }
    }

    mostrar() {
        let aux = this.primero
        while (aux != null) {
            console.log("-> " + aux.id)
            let aux2 = aux.adyacentes.primero
            while (aux2 != null) {
                console.log("    -" + aux2.id)
                aux2 = aux2.siguiente
            }
            aux = aux.siguiente
        }
    }
```

```
    graficar() {
        let cadena= "digraph grafo {\n rankdir=\"LR\" \n"
        let aux = this.primero;
        while(aux != null){
            cadena+="n"+aux.id+"[label= \""+aux.id+", "+aux.nombre+"\"];\n"
            aux = aux.siguiente;
        }
        // graficar relaciones
        aux = this.primero;
        while(aux != null){
            let aux2 = aux.adyacentes.primero;
            while(aux2 != null){
                cadena+= "n"+aux.id+" -> n"+aux2.id+" [label=\""+aux2.distancia+"km\"]; \n";
                aux2 = aux2.siguiente;
            }
            aux = aux.siguiente;
        }
        cadena += "}"
        console.log(cadena);

        const elemento = document.getElementById("contenido-dot")
        elemento.innerHTML = cadena
    }
}
```

Registro de Ventas:

Para la implementación del registro de ventas se utilizó una Tabla Hash de valor 7, utilizando el cálculo de primo inmediato, así como el método de división y también para resolver las colisiones se utilizó la exploración cuadrática. Los valores de la tabla hash son el ID de la venta, el nombre del vendedor, el nombre del cliente, el total de la venta y la lista enlazada de productos provenientes del árbol B.

```
class Nodo {
    constructor(idV, nombreV, nombreC, total, lista) {
        this.idV = idV
        this.nombreV = nombreV
        this.nombreC = nombreC
        this.total = total
        this.lista = lista
    }
}
```

En la clase Hash tenemos un valor de claves que nos sirve para guardar los valores en cada posición de la tabla Hash, así como las claves usadas, que estas se representan en los espacios llenos de la propia tabla hash e igualmente un tamaño de la tabla, que por defecto es 7 e irá incrementando hacia el siguiente número primo:

```
class Hash {
    constructor() {
        this.claves = this.iniciarArreglo(7)
        this.clavesUsadas = 0
        this.size = 7
    }
```

El método de iniciarArreglo únicamente es con la finalidad de iniciar la tabla hash con un valor de 7 por defecto, hasta que se calcule el siguiente número primo, así como el método de calcularHash utilizando el método de división (dividiendo el ID de la venta con el tamaño de la tabla hash):

```
    iniciarArreglo(tamanio) {
        let claves = []
        for (var i = 0; i < tamanio; i++) {
            claves[i] = null
        }
        return claves
    }

    calcularHash(idV) {
        //Metodo de division
        let resultado = 0
        resultado = idV % this.size
        return resultado
    }
```

El método de solucionar colisiones utiliza el método e la exploración cuadrática:

```
solucionColisiones(indice) { //Metodo de exploracion cuadratica
    let nuevoIndice = 0
    let i = 0
    let disponible = false

    while (disponible == false) {
        nuevoIndice = indice + Math.pow(i, 2)
        //Hay que validar que nuevoIndice sea menor al tamaño de la tabla
        if (nuevoIndice >= this.size) {
            nuevoIndice = nuevoIndice - this.size
        }
        //Validar que la posicion del nuevo indice este disponible
        if (this.claves[nuevoIndice] == null) {
            disponible = true
        }
        i++
    }
    return nuevoIndice
}
```

Seguido de eso sigue el método de insertar y el rehash, donde en rehash hacemos lo de encontrar el siguiente número primo y subir el tamaño de nuestra tabla hash:

```
insertar(nuevo) {
    let indice = this.calcularHash(nuevo.idV)

    //Validaciones
    if (this.claves[indice] == null) { //Cuando la posicion esta disponible
        this.claves[indice] = nuevo
        this.clavesUsadas++
    } else { //Cuando existe una colision, hay que resolverla para continuar el flujo del programa
        indice = this.solucionColisiones(indice)
        this.claves[indice] = nuevo
        this.clavesUsadas++
    }

    //Despues de la validacion, toca la validacion de tamaño
    let porcentajeUso = this.clavesUsadas / this.size
    if (porcentajeUso >= 0.5) {
        this.rehash()
    }
}
```

```
rehash() {
    //Encontrar el siguiente número primo
    let primo = false
    let newSize = this.size
    while (primo == false) {
        newSize++
        let contador = 0
        for (var i = newSize; i > 0; i--) {
            if (newSize % i == 0) {
                contador++
            }
        }

        //Validra cuatns veces se dividio exactamente
        if (contador == 2) {
            primo = true
        }
    }
    //Crear nuevo arreglo con el tamaño del siguiente número primo
    let clavesAux = this.claves
    this.size = newSize
    this.claves = this.iniciarArreglo(newSize)
    this.clavesUsadas = 0

    for (var i = 0; i < this.clavesAux.length; i++) {
        if (clavesAux[i] != null) {
            this.insertar(clavesAux[i])
        }
    }
}
```

Lo siguiente es el método de recorrer, en lo cual para evitarnos problemas se recurrió a hacer una tabla en el HTML de la vista para poder desplegar los cambios en la tabla hash, así como también para poder ingresar los productos se recurrió a una clase Producto y un array simple, así que el array guarda el valor del producto de forma fácil y se le hace un push para que guarde los valores:

```
recorrer() {
    let cadenaTabla = ""
    let contador = -1
    for (var i = 0; i < this.size; i++) {
        if (this.claves[i] != null) {
            contador += 1
            console.log("--> " + this.claves[i].idV)
            cadenaTabla += "\n<tr>\n\t<td>"+contador+" | "+" Id: " + this.claves[i].idV + " Vendedor: " + this.claves[i].nombreV + " Cliente: " + this.claves[i].nombreC + " Total: " + this.cl
        } else {
            contador += 1
            console.log("--------------------")
            cadenaTabla += "\n<tr>\n\t<td>"+contador+" |</td>\n</tr>\n"
        }
    }
    const elemento = document.getElementById("mynetwork")
    elemento.innerHTML = cadenaTabla
    console.log(cadenaTabla)
}
}
```

```
class Producto {
    constructor(id, nombre, precio, cantidad) {
        this.id = id
        this.nombre = nombre
        this.precio = precio
        this.cantidad = cantidad
    }
}
```

```
function leerArchivoJSON(e) {
    const archivo = e.target.files[0]
    if (!archivo) {
        return
    }
    const lector = new FileReader()
    lector.onload = function(e) {
        const contenido = e.target.result
        var json = JSON.parse(contenido)
        for (x of json.ventas) {
            let total = 0
            let lista = new Array()
            for (z of x.productos) {
                console.log(z.cantidad)
                total += z.precio * z.cantidad
                let producto = new Producto(z.id, z.nombreC, z.precio, z.cantidad)
                lista.push(producto)
            }
            let ventaNueva = new Nodo(x.id, x.vendedor, x.cliente, total, lista)
            hash.insertar(ventaNueva)
        }
    }
    lector.readAsText(archivo)
    alert("Archivo JSON compilado y datos agregados correctamente")
}

document.querySelector('#archivo1').addEventListener('change', leerArchivoJSON, false)
```

Encriptación

Para la parte de encriptación, se trata de proteger los datos más vulnerables de la aplicación, como los datos de los vendedores (árbol AVL) por lo cual se necesita 0que, durante la carga masiva de datos, se haga un reporte con los datos totalmente encriptados y otro reporte aparte con los datos desencriptados utilizando diferentes librerías y métodos, como por ejemplo crypto JS.

Para poder desencriptar la información se le pedirá al administrador una llave maestra que abrirá el método de desencriptar. Por lo regular se utilizará SHA256 para la encriptación y desencriptación de datos, como se muestra a continuación:

```
const CryptoJS = require('crypto-js');

/**
 *  @MD5 primarily is used for integrity checks on files
 *  Is not collision resistant, and is not advised to be used
 *  with SSL certifications.
 */
const MD5 = CryptoJS.MD5('message');

/**
 * @SHA is a widely used and established hash function developed by
 * the NSA. SHA1 is most recommended.
 */
const SHA = CryptoJS.SHA1("message");

/**
 * @RIPEMD is a lesser known hash function developed in 1992. RIPEMD, despite its
 * uncommon reputation, is widely used in cryptocurrencies such as Bitcoin.
 */
const RIPE = CryptoJS.RIPEMD160("Message");
```

El proceso para encriptar sería como el siguiente ejemplo:

```
/*
 * @AES is considered a victor of 15 compared ciphers in a 5-year evaluation process.
 * This ciphertext is currently considered a U.S Federal Information Processing Standard
 * (FIPS).
 */
const encrypted = CryptoJS.AES.encrypt("TestBird", "test");


console.log(encrypted.toString());

/*
Output: U2FsdGVkX1+ZO3sDSG6QrOheo+1lmi2UzAuk75/D1wg=
*/
```

Y el proceso para desencriptar como el siguiente ejemplo:

```
const decrypted = CryptoJS.AES.decrypt(encrypted, "test").toString(CryptoJS.enc.Utf8);

console.log(decrypted);
/*
Output: TestBird
*/
```

Blockchain

Esta estructura se utiliza para almacenar todas las transacciones de ventas realizadas y este registro debe de hacerse cada cierto período de tiempo, el cual el administrador tiene como predeterminado 5 minutos (tiempo modificable).

Cada bloque contiene un índice, una fecha, la data, el nonce, el previous Hash y el hash actual, como se muestra a continuación:

```
const crypto = require('crypto');

class bloque{
    constructor(indice,data,previusHash){
        this.indice = indice;
        this.data = data;
        this.fecha = Date.now();
        this.previusHash = previusHash;
        this.hash = this.crearHash();
        this.nonce =0;

        this.prueba_de_trabajo(3);
    }
```

Donde nuestro bloque contiene cada uno de los datos pedidos anteriormente.

Para lo siguiente, al crear el hash se utiliza el algoritmo SHA256 para poder encriptar la información, similar a la funcionalidad de encriptación del árbol AVL del punto anterior.

```
crearHash(){
  //usar libreria
  return crypto.createHash('sha256').update(this.indice+this.data+this.fecha+this.previusHash+this.nonce).digest('hex');
}
```

Después tenemos nuestro método de prueba de trabajo:

```
    prueba_de_trabajo(dificultad){
      while(this.hash.substring(0,dificultad) !== Array(dificultad+1).join("0")){
        this.nonce++;
        this.hash = this.crearHash();
        //console.log("->nonce " +this.nonce);
      }
      //console.log(this.hash);
      return this.hash;
    }
}
```

Después viene nuestra clase cadena, que contiene un índice, un array de cadena y también el bloque génesis:

```
class cadena{
  constructor(){
    this.indice=0;
    this.cadena =[];
    this.cadena.push(this.Bloque_genesis());
  }
```

El bloque génesis nos sirve para saber en donde estamos posicionados actualmente con respecto a otros registros existentes:

```
  Bloque_genesis(){
    let genesis = new bloque(this.indice,"bloque Genesis","");
    this.indice++;
    return genesis;
  }
```

Después tenemos nuestros métodos de agregar y recorrer:

```
  agregar(data){
    let nuevo = new bloque(this.indice,data,this.cadena[this.indice-1].hash);
    this.indice++;
    this.cadena.push(nuevo);
  }

  recorrer(){
    for(let item of this.cadena){
      console.log(item);
    }
  }

}
```

CONCLUSIÓN

Conocer el uso de las estructuras de datos es importante, ya que como futuros ingenieros deberemos desarrollar soluciones que sean las más eficientes para no utilizar tantos recursos en los dispositivos donde se ejecute nuestra aplicación, por lo cual, al implementar estructuras de datos en la memoria RAM logramos optimizar procesos de búsqueda y almacenamiento con lo cual podemos lograr el objetivo de tener una aplicación funcional y bien optimizada.