Universidad Tecmilenio

Proyecto Final

Emmanuel Rojas

Desarrollo de Software

Campus Chihuahua

Estructura de datos

3er Semestre

Dafne Abigail Pineda Alarcón

Al07077540

FERRETERÍA

La actualización de documento esta en la página 11.

En este proyecto se usó el caso de una ferretería la cual trabaja con clientes, empleados y sus artículos, todo esto, guardado en tickets con la información de cliente y empleado. El código se hizo con el lenguaje de Java en función de clases y métodos.

La clase Main es el menú principal que ve el cliente, donde se declararon la inicialización de todos los objetos para que se trabaje con los mismo datos en cada uno.

```
class Main{
    Run | Debug
    public static void main(String[] args){
        Scanner read = new Scanner(System.in);
        //Crear objetos
        Cola inventario = new Cola();
        //Fila de los clientes
        Monticulo fila = new Monticulo();
        //Lista de empleados
        ListasEnlazadas empleados = new ListasEnlazadas();
        //tiquets
        Pilas tickets = new Pilas(capacity:50);
```

El inventario contiene los artículos de la ferretería. La fila contiene a los clientes que lleguen. En empleados se registran a los empleados que se contratan. Tickets guarda un historial de atención al cliente.

El menú principal contiene las opciones para interactuar con el inventario, los clientes, los empleados y los tickets, aparte de un ultimo que permite el reinicio del programa, que es SALIR. Las opciones se ingresan con el número enlistado de la opción.

```
        //menu
        boolean menuPrincipal = true;
        while(menuPrincipal){
            System.out.println("\n\n===Ferreteria Luca's Son=== \n"+
                               "1. Inventario.\n"+
                               "2. Cliente en fila\n"+
                               "3. Empleados\n"+
                               "4. Imprimir ticket\n"+
                               "5. Salir\n");
            int menu = read.nextInt(); read.nextLine();
```

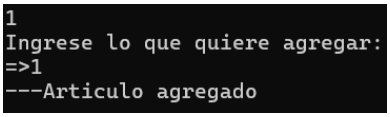
```
===Ferreteria Luca's Son===
1. Inventario.
2. Cliente en fila
3. Empleados
4. Imprimir ticket
5. Salir
```

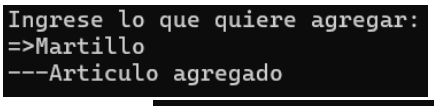
En inventario se tiene otro menú el cual permite escoger agregar, sacar y ver los articulos. La opción deseada se escoge con el número enlistado en la opción.

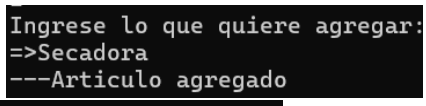
```
case 1:
    System.out.println("\n====INVENTARIO===\n"+
                       "1. Agregar articulo.\n"+
                       "2. Sacar articulo\n"+
                       "3. Ver articulos \n");
    int menuInventario = read.nextInt(); read.nextLine();
    switch (menuInventario) {
```

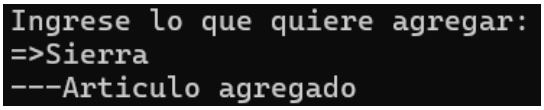
```
====INVENTARIO===
1. Agregar articulo.
2. Sacar articulo
3. Ver articulos
```

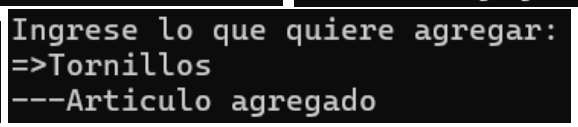
Los articulo se pueden agregar con cualquier simbolo, se les recomienda que sea solo alfabetico para la comprensión de los empleados. ADVERTENCIA: Cada vez que se agrega un articulo, te regresa al menú principal, solo selecciones de nuevo el número enlistado de INVENTARIO.



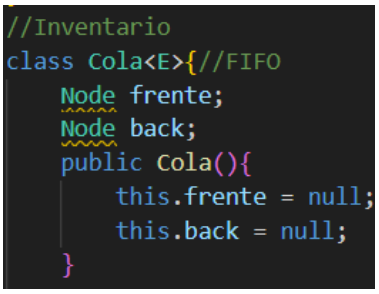




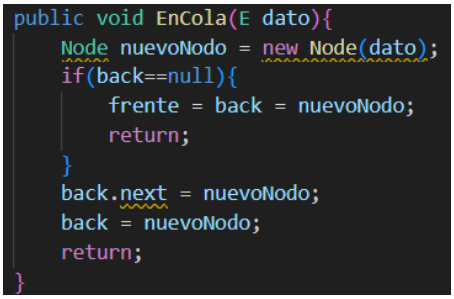




Los artículos se guardan en un objeto que se crean en la clase Cola. Necesita tres atributos que son el frente y trasero para respetar la estructura FIFO(First In, FIrst Out).



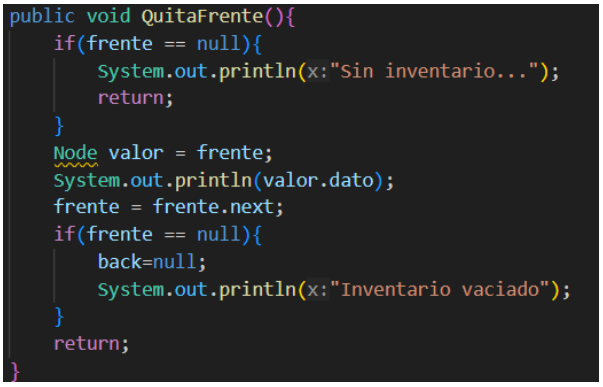
```java
//Inventario
class Cola<E>{//FIFO
    Node frente;
    Node back;
    public Cola(){
        this.frente = null;
        this.back = null;
    }
```



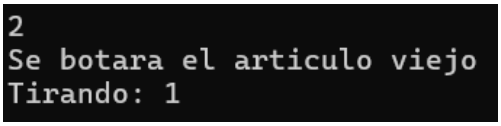
```java
public void EnCola(E dato){
    Node nuevoNodo = new Node(dato);
    if(back==null){
        frente = back = nuevoNodo;
        return;
    }
    back.next = nuevoNodo;
    back = nuevoNodo;
    return;
}
```

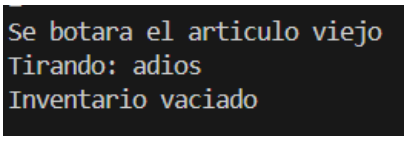
La primera vez que se agregue un articulo, el metodo EnCola pone la cabeza y la cola comoel mismo articulo, en cambio, las siguientes solo actualizaran la cola y el anterior objeto se enlazara al objeto actual.

La segunda opción de un articulo es sacar el primer articulo que se ingreso. Esto sucede gracias al metodo QuitaFrente, el cual tiene un filtro para evitar el error si no hay articulos, pero si los hay, la cabeza se recorre al siguiente articulo y se pierde el anterior. Si se eliminan todos los articulos, tanto cabeza como trasero se actualizan y se notifica del vaciado del inventario.



```java
public void QuitaFrente(){
    if(frente == null){
        System.out.println(x:"Sin inventario...");
        return;
    }
    Node valor = frente;
    System.out.println(valor.dato);
    frente = frente.next;
    if(frente == null){
        back=null;
        System.out.println(x:"Inventario vaciado");
    }
    return;
}
```





Al mostrar los artículos, solo se imprimen estos a base del método mostrar. Si no hay artículos, se le notificara al usuario. De haber artículos, se tomara una variable temporal que pasara/actualizara por todos los objetos para imprimir lo que contiene cada uno.

```java
public void mostrar(){
    System.out.println(x:"--------Inventario--------");
    if(back == null){
        System.out.println(x:"Inventario vacio");
        return;
    }
    Node lista = frente;
    System.out.println(x:"-------------------------");
    while(lista != null){
        System.out.println("|\t"+lista.dato+"\t\t|");
        lista = lista.next;
    }
    System.out.println(x:"-------------------------");
    return;
}
```

```
3
--------Inventario--------
-------------------------
|       Martillo                |
|       Secadora                |
|       Sierra          |
|       Tornillos               |
-------------------------
```

La opción de empleados permite el uso de otro menú para interactuar con los empleados: Contratar, despedir al primero o el ultimo, ver todos los empleados y ver el empleado que sigue luego de x empleado ingresado. La opción es con el número enlistado.

```java
System.out.print("\n===EMPLEADOS===\n"+
                 "1. Contratar empleado\n"+
                 "2. Despedir primer empleado\n"+
                 "3. Despedir ultimo empleado\n"+
                 "4. Ver todos los empleados\n"+
                 "5. Empleado siguiente\n"+
                 "=>");
 int menuEmpleado = read.nextInt(); read.nextLine();
```

```
===EMPLEADOS===
1. Contratar empleado
2. Despedir primer empleado
3. Despedir ultimo empleado
4. Ver todos los empleados
5. Empleado siguiente
=>
```

La opción de contratar un empleado solo se solicita el nombre del empleado y esta llama a un método push. Este método esta dentro de la clase ListasEnlazadas que tendrá una cabeza y una cola para crear un inicio y un fin de esta estructura circular.

```java
case 1:
    System.out.print(s:"Ingrese el nombre del empleado: \n=>>");
    String nombre = read.nextLine();
    empleados.push(nombre);
    break;
```

```java
class ListasEnlazadas<E>{//Lista enlazada circular
    Node cabeza;
    Node cola;
    public ListasEnlazadas(){
        this.cabeza=null;
        this.cola=null;
    }
```

El metodo push tiene un filtro para agregar al primer empleado el cual guarda tanto en cabeza como cola y aparte empieza la estructura circular apuntandose a si mismo. Si ya hay empleados previamente guardados, se creara el nuevo objeto del empleado y el anterior objeto a este se enlazara al mismo para actualizar la cola de nuevo y enlazar este a la cabeza. Se enviara una notificación de que se pudo contratar al usuario.

```java
public void push(String nombre){
    if(cola==null){
        Node empleadoNuevo=new Node(nombre);
        this.cabeza=this.cola=empleadoNuevo;
        empleadoNuevo.next = cabeza;
        System.out.println(x:"Empleado contratado");
        return;
    }
    Node empleadoNuevo = new Node(nombre);
    cola.next=empleadoNuevo;
    cola = empleadoNuevo;
    empleadoNuevo.next=cabeza;
    System.out.println(x:"Empleado contratado");
}
```

```
Ingrese el nombre del empleado:
=>>Dafne
Empleado contratado
Ingrese el nombre del empleado:
=>>Alfredo
Empleado contratado
Ingrese el nombre del empleado:
=>>Claudia
Empleado contratado
Ingrese el nombre del empleado:
=>>Sol
Empleado contratado
```

Al despedir al primer usuario contratado, nos apoyamos de un metodo llama EliminarPrimero el cual tiene un filtro por si esta vacio el registro. Si no esta vacio, solo se actualiza la cabeza al siguiente objeto y la cola debera apuntar a la nueva cabeza.

```java
case 2:
    empleados.EliminarPrimero();
    break;
```

```java
public boolean isCabezaEmpty(){
    return cabeza == null;
}
```

```java
public void EliminarPrimero(){
    if(isCabezaEmpty()){System.out.println(x:"Sin empleados");return;}
    cabeza = cabeza.next;
    cola.next = cabeza;
    System.out.println(x:"Despedido el primer empleado");
}
```

```
=>2
Despedido el primer empleado
```

El metodo EliminarUltimo es el metodo que tambien despide a un empleado. Tambien tiene un filtro por si esta vacio. Con un While se logra acceder al penultimo objeto que apunta a la cola y se mueve la cola al penultimo objeto y actualizamos la cola apuntando a la cabeza.

```java
case 3:
    empleados.EliminarUltimoo();
    break;
```

```
=>3
Despedido el ultimo empleado
```

```java
public void EliminarUltimoo(){
    if(isCabezaEmpty()){System.out.println(x:"Sin empleados");return;
    Node temporal = cabeza;
    while(temporal.next != cola){
        temporal = temporal.next;
    }
    cola = temporal;
    cola.next = cabeza;
    System.out.println(x:"Despedido el ultimo empleado");
}
```

Al mostrar, se avisa si esta vacía la lista, en cambio, de tener registros, se imprime en apoyo de una variable que recorre a todos los empleados hasta llegar al empleado declarado como la cola, para que no se cicle la rotación.

```
case 4:
    empleados.mostrar();
    break;
```

```
===Nombres y orden de empleados==

Alfredo -> Claudia -> ¬
```

```
public void mostrar(){
    System.out.println(x:"\n===Nombres y orden de empleados==\n");
    if(isCabezaEmpty()){System.out.println(x:"Sin empleados");return;}
    Node temporal = cabeza;
    do{
        System.out.print(temporal.dato+" -> ");
        temporal=temporal.next;
    }while(temporal!=cabeza);
    System.out.println(x:"¬");
}
```

La opción de buscar un empleado, es a través del nombre de este, si el nombre esta mal escrito, no mostrara nada, en cambio, se comparara con todos los nombres y se imprimirá que empleado sigue de este.

```
case 5:
    System.out.print(s:"Ingrese el nombre de un empleado: \n=>>");
    nombre=read.nextLine();
    empleados.buscar(nombre);
    break;
```

```
Ingrese el nombre de un empleado:
=>>Claudia
El siguiente empleado despues de Claudia es Alfredo
```

```
public void buscar(E nombre){
    if(isCabezaEmpty()){System.out.println(x:"Sin empleados");return;}
    Node temporal = cabeza;
    do{
        if(temporal.dato.equals(nombre)){
            temporal=temporal.next;
            System.out.println("El siguiente empleado despues de "+nombre+" es "+temporal.dato);
            return;
        }
        temporal = temporal.next;
        //System.out.println("entro y checo a "+temporal.dato);
    }while(temporal!=cabeza);
    System.out.println(x:"No se encontro a este empleado");
    return;
}
```

El método getName da el nombre según las veces que se salta a los empleados y regresa solo el nombre de este hasta que se acabe las veces que se saltan a los anteriores.

```java
public String getName(int rotacion){
    int iteracion = 0;
    Node temp = cabeza;
    do{
        temp = temp.next;
        iteracion++;
    }while(iteracion < rotacion);
    return temp.dato.toString();
}
public boolean isCabezaEmpty(){
    return cabeza == null;
}
```

Default permite indicarle al usuario que escogió una opción no valida.

```java
default:
    System.out.println(x:"Opcion invalida");
    break;
```

La opción de los clientes abre un menú con solo dos opciones, la cual permite formar a un cliente y ver la fila. Donde se le pide el nombre al cliente y la cantidad de artículos que comprara en total, estos se enviaran a dos métodos, uno, permite el registro del cliente y el otro el registro en tickets.

```java
case 2:
    System.out.println("\n===CLIENTES===\n"+
                       "1. Se formo un cliente\n"+
                       "2. Ver la fila");
    int menuCliente = read.nextInt(); read.nextLine();
```

```
===CLIENTES===
1. Se formo un cliente
2. Ver la fila
```

```java
switch(menuCliente){
    case 1:
        System.out.print(s:"Si se ha formado un cliente, ingrese su nombre:\n=>");
        String nombre = read.nextLine();
        System.out.print(s:"Cantidad de articulos que comprara:\n=>" );
        int articulos = read.nextInt(); read.nextLine();
        fila.nuevoCliente(nombre, articulos);
        tickets.push(empleados, nombre);
        break;
```

La clase de Monticulo guarda a los clientes, estos son a través de un arreglo previamente ya establecido para 50 personas y un size que controlara la cantidad de rendijas que de verdad estas llenas.

```java
class Monticulo{//Colas Priorizadas
    Node clientes[];
    int size;


    public Monticulo(){
        this.clientes = new Node[50];
        this.size = 0;
    }
```

En el método nuevoCliente, se guarda el nombre y los artículos, si esta lleno, que se verifica con otro método, se imprime el aviso, en cambio, si no, se crea un nuevo objeto del cliente y verificamos si es el primero, si es el primero, no pasara nada, en cambio, si es el segundo, se agrega a la lista y se hace una verificación.

```
//Movieminentos basicos
    public void nuevoCliente(String nombre, int articulos){
        if(lleno()){System.out.println(x:"Fila demasiado larga"); return;}
        //Creamos nuevo nodo
        Node nuevoCliente = new Node(nombre, articulos);
        //Agregamos al cliente en el index 1 si no hay clientes
        if(size == 0){
            clientes[++size]=nuevoCliente;
            System.out.println(x:"Primer cliente a la fila\n");
            return;
        }
        //Agregamos cliente
        clientes[++size] = nuevoCliente;
        flotar(size);
    }
```

```
//Si esta lleno o vario
    public boolean vacio(){return size==0;}
    public boolean lleno(){return size==clientes.length-1;}
```

EL método flotar, lo que hace es buscar el índex donde se encuentra el papa y este número se entrega a otro método que verifica si el primer cliente tiene más artículos que el siguiente. Luego, hay un ciclo el cual vuelve a verificar todos los clientes para ver si hay algún cliente al frente con más artículos que los que tiene atrás.

```
//Mover un numero hasta arriba
    public void flotar(int indexNuevoCliente){
        int indexPapa = setPapa(indexNuevoCliente);
        bajarPapa(indexPapa);
        if(size>2){
            for(int i = 2; i<size; i++){
                bajarPapa(setPapa(i));
            }
        }
    }
```

```
//decimos la posicion de los hijos
    public int setPapa(int posicion){return posicion / 2;}
    public int setHijoDer(int qnPapa){return (2*qnPapa+1);}
    public int setHijoIzq(int qnPapa){return (2*qnPapa);}
```

En el método de ver si hay otros clientes, se verifica primero si hay otros clientes formados atrás, si no, no tendría sentido cambiarlo. Si si hay un cliente atrás, se verifican las cantidad de productos que tiene cada uno, si el cliente de atrás tiene menos artículos que el de enfrente, se cambian de lugar, y luego se hace otra verificación si hay otro cliente que sea tenga menos artículos que el de en frente. Si todo lo anterior fue cambiado, los dos clientes de atrás se deben comparar entre ellos para saber cual va en frente, esto se hace con el método cambiarHermanos.

```java
//cambiar al papa por los hijos
    public void bajarPapa(int indexPapa){
        if(hayHijoIzq(indexPapa)){
            int hermanoIzq = setHijoIzq(indexPapa);
            if(clientes[hermanoIzq].prioridad < clientes[indexPapa].prioridad){
                Node nombrePapa = clientes[indexPapa];
                Node nombreIzq = clientes[setHijoIzq(indexPapa)];
//Cambiamos los lugares
                clientes[hermanoIzq] = nombrePapa;
                clientes[indexPapa] = nombreIzq;
                if(hayHijoDer(indexPapa)){cambiarHermanos(indexPapa);}
                return;
            }
            if(hayHijoDer(indexPapa)){
                int hermanoDer = setHijoDer(indexPapa);
                if(clientes[hermanoDer].prioridad < clientes[indexPapa].prioridad){
                    Node nombrePapa = clientes[indexPapa];
                    Node nombreDer = clientes[hermanoDer];
                    clientes[indexPapa] = nombreDer;
                    clientes[hermanoDer] = nombrePapa;
                    cambiarHermanos(indexPapa);
                    return;
                }
            }
        }
    }
//Juntamos a hijos y fathers
    public boolean hayHijoIzq(int Papa){return setHijoIzq(Papa) <= this.size;}
    public boolean hayHijoDer(int Papa){return setHijoDer(Papa) <= this.size;}
```

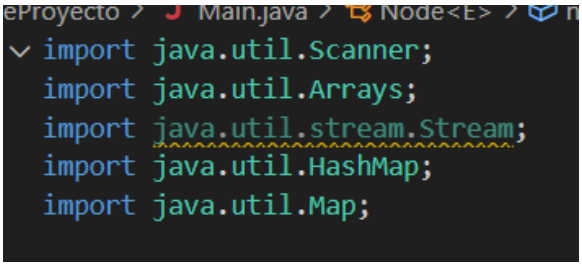
En el método cambiarHermanos, se comparan ambos clientes y se ve cual va primero, si el cliente detrás del segundo tiene menos artículos, se cambian entre estos, si no, permanecen tal cual entraron.

```java
//Cambiar hermanos
public void cambiarHermanos(int indexPapa){
    int hermanoIzq = setHijoIzq(indexPapa);
    int hermanoDer = setHijoDer(indexPapa);
    if(clientes[hermanoIzq].prioridad > clientes[hermanoDer].prioridad){
        Node nombreDer = clientes[hermanoDer];
        Node nombreIzq = clientes[hermanoIzq];
        clientes[hermanoDer] = nombreIzq;
        clientes[hermanoIzq]= nombreDer;
        return;
    }
    return;
}
```

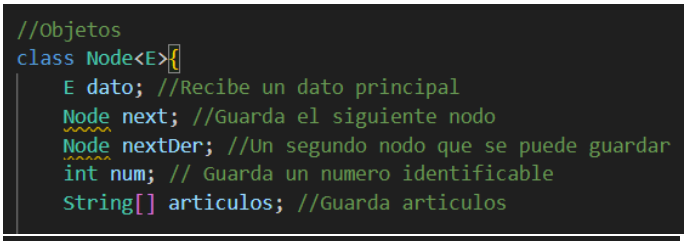
PARTE II

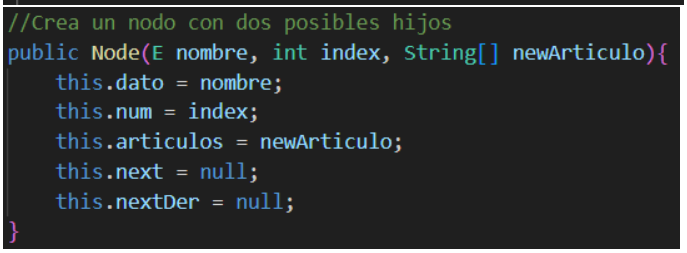
En este apartado, está registrado la continuación del programa para una ferretería. Se agregaron las estructuras de arboles binarios, recursividad con algoritmos “Divide y vencerás”, tablas hash, métodos de ordenamiento y búsqueda, y grafos.

Se agregaron las librerías de stream, hash y map, para el trabajo con las listas.



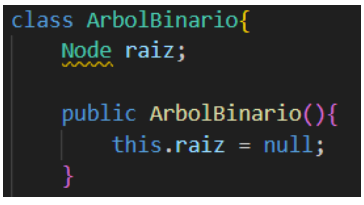
La clase Node fue actualizada con dos atributos más, un arreglo y un nuevo apuntador, los cuales se usan en arboles binarios. Tambien se le agrego un constructor el cual trabaja como un Nodo de la estructura arboles.





Arbol binario

La clase del árbol binario, tienen un único atributo, raíz, el cual guarda el primer Nodo, el papá de los demás.



El método **agregarProveedor**, recibe el nombre, índex y los artículos que provee, trabaja como un método recursivo, con un caso base, que la raíz no tenga dato.

Las dos condicionales, filtran los números sobre a que lado debe de ir y si este es el ultimo Nodo del árbol.

```java
//Agregamos un nuevo nodo al arbol
public void agregarProveedor(Node raizActual, String proveedor, int index, String[] articulos){
    //Si la raiz esta vacia, la establecemos como raiz
    if(raiz == null){
        Node nuevoProveedor = new Node(proveedor, index, articulos);
        raiz = nuevoProveedor;
        return;
    }
    //Verificamos el nuevo index sea mayor
    if(raizActual.num > index){
        if(raizActual.next == null){ //Si llegamos al ultimo hijo izquierdo
           raizActual.next = new Node(proveedor, index, articulos);//Se agrega
            return;
        }
        agregarProveedor(raizActual.next, proveedor, index, articulos);//Exploramos el proximo hijo izquierdo
    //Verificamos que el index sea menor
    }else if( raizActual.num < index){
        if(raizActual.nextDer == null){//Que no haya hijo derecho
            raizActual.nextDer = new Node(proveedor, index, articulos);
            return;
        }
        agregarProveedor(raizActual.nextDer, proveedor, index, articulos);//Exploramos el proximo hijo derecho
    }
}
```

Hay dos métodos que se ciclan buscando el padre de un índex dado. Uno trabaja con Nodos del lado izquierdo y el otro con el lado derecho.

```java
//Buscar al num pequenno del lado izquierdo y regresar al papa
 public Node sucesorPequenno(Node indexActual){
    Node papa = indexActual;
    while(indexActual.next != null){
        papa = indexActual;
        indexActual = indexActual.next;
    }
    return papa;
}
//Buscar al num grande del lado derecho y regresar al papa
 public Node sucesorGrande(Node indexActual){
    Node papa = indexActual;
    while(indexActual.nextDer != null){
        papa = indexActual;
        indexActual = indexActual.nextDer;
    }
    return papa;
}
```

El método **eliminarProveedor** elimina un Nodo a partir de un índex. Tiene un caso base por si el índex no existe y dos filtros el cual busca si el índex buscado es mayor o menor.

```java
//Eliminar un elemento
public void eliminarProveedor(Node indexAnterior, Node indexActual, int num){
    //Pasamos por todos los index y no se encontro
    if(indexActual == null){
        System.out.println(x:"---El proveedor no existe---");
        return;
    }
    //El index es menor
    if(indexActual.num > num){eliminarProveedor(indexActual, indexActual.next, num);return;}
    //El index es mayor
    if(indexActual.num < num){eliminarProveedor(indexActual, indexActual.nextDer, num); return;
```

Si no entra, significa que el índex fue encontrado, por lo que se busca si el nodo a borrar cumple con alguno de los casos: **es la raíz**, **es un nodo hoja**, **nodo con un solo hijo** o **un nodo con dos hijos**.

Si es la raíz, se buscara el nodo más pequeño del lado derecho y se actualizaran los valores y se dejara de apuntar a la anterior raíz.

```java
}else{
    if(indexActual == raiz){
        //pero es la raiz
        if(raiz.next == null && raiz.nextDer == null){raiz=null;return;} //No tiene hijos

        //Verificamos que haya hijo derecho
        if(indexActual.nextDer != null){
            Node papaPeque = sucesorPequenno(indexActual.nextDer);
            Node nuevoNodo = papaPeque;
            Node viejoNodo = indexActual;
            this.raiz = nuevoNodo;
            if(viejoNodo.next != null){
                this.raiz.next = viejoNodo.next;
            }
            if(viejoNodo.nextDer != this.raiz){
                this.raiz.nextDer = viejoNodo.nextDer;
                papaPeque.next = null;
            }
            preorden(raiz);
            return;
```

Si no existe el lado derecho, se hará lo mismo del lado izquierdo, pero buscando el nodo más grande.

```java
    }else{
        //Buscamos al papa del nodo mas grande
        Node papaGrande = sucesorGrande(indexActual.next);
        Node nuevoNodo = papaGrande.nextDer;
        Node viejoNodo = indexActual;
        //actualizamos la raiz a este nuevo nodo
        this.raiz = nuevoNodo;
        //Quitamos la referencia del padre
        if(viejoNodo.next != this.raiz){
            this.raiz.next = viejoNodo.next;
            papaGrande.nextDer = null;

        }
        preorden(raiz);
        return;
    }
}
```

Si es un nodo hoja, simplemente eliminaremos el apuntador del nodo padre.

```
//Caso 1: Nodo hoja
else if(indexActual.next == null && indexActual.nextDer == null){
    //Si es el hijo izquierdo
    if(indexAnterior.next == indexActual){
        indexAnterior.next = null;
        return;
    }else{ //Si es el hijo derecho
        indexAnterior.nextDer = null;
        return;
    }
}
```

Si el nodo tiene un hijo, primero verificamos de qué lado está el hijo para saber que apuntador usaremos. También es necesario saber quien es el padre para actualizar su apuntador.

Por último, eliminaremos el nodo actualizándolo.

```
//Caso 2: Nodo con un hijo izquierdo
else if(indexActual.next != null && indexActual.nextDer == null){
    //Es el hijo izquierdo
    if(indexActual.next == indexActual){
        indexAnterior.next = indexActual.next;
        return;
    }else{ //El hijo esta a la derecha
        indexAnterior.nextDer = indexActual.next;
        return;
    }
//Caso 2: Nodo con un hijo derecho
}else if(indexActual.next == null && indexActual.nextDer!=null){
    //El nieto esta a la izquierda
System.out.println(x:"hijo derecho");
    if(indexAnterior.next == indexActual){
        indexAnterior.next = indexActual.nextDer;
        return;
    }else{ //El nieto esta a la derecha
        indexAnterior.nextDer = indexActual.nextDer;
        return;
    }
```

Si el nodo tiene dos hijos, buscaremos en el lado derecho, el nodo más grande, actualizamos los apuntadores y borramos el index.

```
//Caso 3: Nodo con dos hijos
}else{
    //Verificamos que haya hijo derecho
    if(indexActual.nextDer != null){
        //Buscamos el num pequennoy guardamos el anterior nodo
        Node papaPequeno = sucesorPequenno(indexActual.nextDer); //buscamos al papa del ultimo nodo
        Node nuevoNodo = papaPequeno.next;
        Node viejoNodo = indexActual;

        //El viejo nodo esta a la izquierda de papa
        if(indexAnterior.next == viejoNodo)indexAnterior.next = nuevoNodo;
        //El viejo nodo esta a la derecha de papa
        else indexAnterior.nextDer = nuevoNodo;

        //Actualizamos los hijos del nuevo nodo
        if(viejoNodo.next != null){
            nuevoNodo.next = viejoNodo.next;
        }
        //Cuidamos que no sea un hijo directo
        if(viejoNodo.nextDer!=nuevoNodo){
            nuevoNodo.nextDer= viejoNodo.nextDer;
            papaPequeno.next = null;
        }
        return;
```

Si no hay nodos del lado derecho (caso prácticamente imposible), se hará lo mismo del lado izquierdo.

```
    //Subimos al hijo mas grande izq
    }else{
        Node papa = sucesorGrande(indexActual.next);
        Node nuevoNodo = papa.nextDer;
        Node viejoNodo = indexActual;
        //Buscamos el No mas grande de

        //Verificamos que el viejo nodo este del izquierdo
        if(indexAnterior.next == viejoNodo) indexAnterior.next = nuevoNodo;//actualizamos
        //Esta en el lado derecho
        else indexAnterior.nextDer = nuevoNodo;//actualizamos

        if(viejoNodo.next != nuevoNodo){
            nuevoNodo.next = viejoNodo.next;
            papa.nextDer = null;
        }
    }
}
```

Usamos tres métodos para imprimir el árbol, con preorden, inorden y postorden.

```
public void preorden(Node proveedor){
    if(raiz==null){System.out.println(x:"No hay proveedores");return;}
    System.out.println("\t"+proveedor.num+": "+proveedor.dato+" => "+proveedor.articulos[0]+", "+proveedor.articulos[1]);
    if(proveedor.next!= null)preorden(proveedor.next);
    if(proveedor.nextDer!=null)preorden(proveedor.nextDer);
}
public void inorden(Node proveedor){
    if(raiz==null){System.out.println(x:"No hay proveedores");return;}
    if(proveedor.next!= null)preorden(proveedor.next);
    System.out.println("\t"+proveedor.num+": "+proveedor.dato+" => "+proveedor.articulos[0]+", "+proveedor.articulos[1]);
    if(proveedor.nextDer!=null)preorden(proveedor.nextDer);
}
public void postorden(Node proveedor){
    if(raiz==null){System.out.println(x:"No hay proveedores");return;}
    if(proveedor.next!= null)preorden(proveedor.next);
    if(proveedor.nextDer!=null)preorden(proveedor.nextDer);
    System.out.println("\t"+proveedor.num+": "+proveedor.dato+" => "+proveedor.articulos[0]+", "+proveedor.articulos[1]);
}
```

Para buscar el índex de un proveedor, usaremos un método recursivo que recorrerá con la búsqueda de postorden

```
public void buscarProveedor(int x, Node proveedorActual){
    //Primer caso base, no existe
    if(proveedorActual == null){System.out.println(x:"No existe este index"); return;}
    //Buscamos en modo postOrden
    if(proveedorActual.next!= null)buscarProveedor(x, proveedorActual);
    if(proveedorActual.nextDer!=null)buscarProveedor(x, proveedorActual);
    if(proveedorActual.num == x){
        System.out.println("\t"+proveedorActual.num+": "+proveedorActual.dato+" => "+proveedorActual.articulos[0]+", "+proveedorActual.articulos[1]);
        return;
    }
}
```

Ordenamiento y búsqueda

La clase de ordenamiento tiene dos atributos, un arreglo y un contador que controla el tamaño.

El constructor, declara el tamaño como vacío y se establece el tamaño del arreglo.

```
//Ordnamiento y busqueda
class Ordenamiento{//Lista de cosas que faltan en la ferreteria por prioridad
    String[][] arreglo;
    int size;
    public Ordenamiento(){//Constructor
        this.size = -1;
        this.arreglo = new String[50][2];
    }
```

El método burbuja, usa el tipo de ordenamiento que pone los índex la prioridad en que se encuentra el arreglo.

```java
public void burbuja(){//Ordenamiento por burbuja
    for(int i = 0; i<size; i++){
        for(int j = 0; j< size-1; j++){//Cuida los ciclos que no pase los que ya flotaron
            int par1 = Integer.parseInt(arreglo[j][0]);//Guardamos la prioridad
            int par2 = Integer.parseInt(arreglo[j+1][0]);//Guardamos la priodidad
            if(par1 > par2){
                String[] numTemp = arreglo[j];
                arreglo[j]=arreglo[j+1];
                arreglo[j+1]=numTemp;
            }
        }
    }
    mostrar();
}
```

EL método agregar, aumenta el índex y agrega los datos y el método mostrar, se apoya de un ciclo para mostrar cada elemento del arreglo.

```java
public void agregar(String articulo, String prioridad){
    ++size;
    this.arreglo[size][0] = prioridad;
    this.arreglo[size][1] = articulo;
    burbuja();
}
```

```java
public void mostrar(){
    if(size == -1){System.out.println(x:"--Sin pendientes"); return;}
    for(int i=0; i<size; i++){
        System.out.println(arreglo[i][0]+": "+arreglo[i][1]);
    }
}
```

Tablas hash

El primer metodo, es un constructor el cual crea el mapa.

El método **agregarUbicacion** el cual agrega al mapa la categoría de las herramientas y una breve descripicón de dónde está.

El método **consultarUbicacion** busca una categoría a partir de su nombre, si existe, se regresan los valores del mismo, si no, se da aviso de que no existe.

```java
class hashAlmacen {
    // La Tabla Hash donde la Clave es el nombre del material (String)
    // y el Valor es la ubicación en el almacén (String).
    private Map<String, String> ubicacionesMateriales;

    public hashAlmacen() {
        // Inicializa el HashMap
        this.ubicacionesMateriales = new HashMap<>();
    }
    public void agregarUbicacion(String material, String ubicacion) {
        // El método put() inserta el par clave-valor en la tabla hash.
        ubicacionesMateriales.put(material.toUpperCase(), ubicacion);
    }
    public String consultarUbicacion(String material) {
        String materialKey = material.toUpperCase();

        // El método get() accede directamente al valor usando la clave.
        // Esto es lo que proporciona la velocidad O(1) de la tabla hash.
        if (ubicacionesMateriales.containsKey(materialKey)) {
            return "Ubicación de " + material + ": " + ubicacionesMateriales.get(materialKey);
        } else {
            return "⚠️ Material \"" + material + "\" no encontrado en el sistema.";
        }
    }
}
```

Recursividad

Dentro de la clase **main**, está el método estatico el cual recive un arreglo de Strings, el cual va diviendo en arerglos más pequeños para ordenarlos desde la cadena más grande hasta la más pequeña.

```java
//Hacer entregas pequennas
//Dividimos un arreglo de todas las entregas que se haran
//Se entrega primero los mas largos
public static String[] recursividad(String[] entregas1) {
    if (entregas1.length <= 1) {
        return entregas1;
    }

    int mitad = entregas1.length / 2;

    // 1. División recursiva
    // Se llama a entregas, que devuelve un sub-arreglo ORDENADO.
    String[] primerGrupoOrdenado = recursividad(Arrays.copyOfRange(entregas1, from:0, mitad));
    String[] segundoGrupoOrdenado = recursividad(Arrays.copyOfRange(entregas1, mitad, entregas1.length));

    // 2. Fusión (Merge)
    // Se fusionan los dos sub-arreglos ordenados y se devuelve el resultado.
    return merge(primerGrupoOrdenado, segundoGrupoOrdenado);
}
```

El método de merge, recive y regresa un arreglo, su función es juntar los sub arreglos ya ordenados.

```java
// Método de Fusión (Merge) que hace la mayor parte del trabajo de ordenamiento
private static String[] merge(String[] arr1, String[] arr2) {
    int i = 0, j = 0, k = 0; // i para arr1, j para arr2, k para el resultado
    String[] resultado = new String[arr1.length + arr2.length];

    // Mientras haya elementos en ambos arreglos...
    while (i < arr1.length && j < arr2.length) {
        // Comparamos longitudes. Queremos el MÁS LARGO primero (orden descendente).
        if (arr1[i].length() >= arr2[j].length()) {
            resultado[k++] = arr1[i++];
        } else {
            resultado[k++] = arr2[j++];
        }
    }
    // Copiar los elementos restantes de arr1 (si los hay)
    while (i < arr1.length) {
        resultado[k++] = arr1[i++];
    }
    // Copiar los elementos restantes de arr2 (si los hay)
    while (j < arr2.length) {
        resultado[k++] = arr2[j++];
    }
    return resultado;
}
```

Grafos

El metodo **mostrarPedidos** muestra por un ciclo, los elementos del arreglo.

```java
//mostramos
    public static void mostrarPedidos(String[] entregas){
        for(int iter = 0; iter < entregas.length; iter++){
            System.out.print(entregas[iter]+", ");
        }
    }
```

El método **acomodarn** recive una matríz y con 3 arreglos, verifica los valores e intercambia por valores pequeños.

```java
public static float[][] acomodorar(float[][] grafo){
    for(int intermedio=0; intermedio<grafo.length; intermedio++){
        for(int oriInicio = 0; oriInicio<grafo.length;oriInicio++){
            for(int oriFinal=0; oriFinal<grafo.length; oriFinal++){
                if(grafo[oriInicio][intermedio] + grafo[intermedio][oriInicio] < grafo[oriInicio][oriFinal] && grafo[oriInicio][intermedio]!=0){
                    grafo[oriInicio][oriFinal] = grafo[oriInicio][intermedio] + grafo[intermedio][oriInicio];
                }
            }
        }
    }
    return grafo;
}
```

Main

Se inicializan los objetos que se usaran.

```java
ArbolBinario proveedores = new ArbolBinario();
//Cajas
Ordenamiento cajas = new Ordenamiento();
//almacen
hashAlmacen almacen = new hashAlmacen();
//Memoria de la anterior mejor ruta
float[][] grafo =null;
```

El menú es alargado con las nuevas opciones, los cuales inician a partie del 6.

```java
while(menuPrincipal){
    System.out.println("\n\n===Ferreteria Luca's Son=== \n"+
                       "1. Inventario.\n"+
                       "2. Cliente en fila\n"+
                       "3. Empleados\n"+
                       "4. Imprimir ticket\n"+
                       "5. Proveedores\n"+
                       "6. Entrega compleja\n"+
                       "7. tablas hash\n"+
                       "8. Lista de compra\n"+
                       "9. Mapa\n"+
                       "10. Salir\n");
    int menu = read.nextInt(); read.nextLine();
```

El menú de proveedores y la primer opción

```java
case 5: //Menu de proveedores
    System.out.print("\n----Proveedores----\n"+
                     "1. Ver proveedores\n"+
                     "2. Ingresar proveedor\n"+
                     "3. Eliminar proveedor por index\n"+
                     "4. Buscar proveedor por index\n"+
                     "=>");
    int menuProveedores = read.nextInt();read.nextLine();
    switch(menuProveedores){
```

```java
case 1://Mostrar proveedores
    System.out.println(x:"---Inorden---");
    proveedores.inorden(proveedores.raiz);
    System.out.println(x:"---Postorden---");
    proveedores.postorden(proveedores.raiz);
    System.out.println(x:"---Preorden---");
    proveedores.preorden(proveedores.raiz);
    break;
```

Al agregar, se piden los datos y se convierten al tipo de dato necesario.

```java
case 2://Agregar proveedor
    System.out.print(s:"Ingrese el nombre: "); String nombreProveedor = read.nextLine();
    System.out.print(s:"Ingrese el index: "); int numProveedor = read.nextInt(); read.nextLine();
    String[] articulosProveedor = new String[2];
    System.out.print(s:"Ingrese 2 articulos que provea: ");articulosProveedor[0] =read.nextLine();
    System.out.print(s:"=> "); articulosProveedor[1] = read.nextLine();
    proveedores.agregarProveedor(proveedores.raiz, nombreProveedor, numProveedor, articulosProveedor)
    break;
```

Eliminar y buscas es a partir del index, por lo que ambos solicitan un número y se llama a su respectivo método.

```java
case 3://Eliminar proveedor
    System.out.print("==Elimnar proveedor por index===\n"+
                     "Ingrese el index: ");
    int indexProveedor = read.nextInt();read.nextLine();
    proveedores.eliminarProveedor(indexAnterior:null, proveedores.raiz, indexProveedor);
    break;
case 4://Buscar proveedor
    System.out.print("==Buscar proveedor por index===\n"+
                     "Ingrese el index: ");
    indexProveedor = read.nextInt();read.nextLine();
    proveedores.buscarProveedor(indexProveedor, proveedores.raiz);
```

El menú 6 recive strings describiendo los pedidos que tienen. Se llama a un médoso el cual los ordena y los muestra.

```
case 6://Entrega compleja - Recursividad
//Solicitamos los datos
    System.out.print("\n---Division de entregas complejas---\n"+
                     "Ingrese cuantos pedidos son: ");
    int pedidos = read.nextInt();read.nextLine();
    String[] entregas = new String[pedidos];
    System.out.println(x:"Ingrese lo que entregara");
    for(int iter = 0; iter<entregas.length; iter++){
        System.out.print(s:"=>");
        entregas[iter]=read.nextLine();
    }

//Ordenamos los datos
    System.out.println(x:"El problema acomodado");
    mostrarPedidos(recursividad(entregas));
    break;
```

Las tablas hash, de la opción 7, tienen un menú el cual agrega una nueva ubicación de alguna categoría y tambien se puede buscar a traves del índex.

```
case 7://Tablas hash
    System.out.print("\n---Almacen---"+
                     "\n1. Agregar ubicacion"+
                     "\n2.buscar ubicacion\n=>");
    int menuHash = read.nextInt();read.nextLine();
    switch(menuHash){
        case 1:
            System.out.print(s:"Ingrese los tipos de herramientas(Ejem: Carpi)");
            String material = read.nextLine();
            System.out.print(s:"Ingrese una breve descripcion de donde esta\n=>");
            String ubicacion = read.nextLine();
            almacen.agregarUbicacion(material, ubicacion);
            break;
        case 2:
            System.out.print(s:"Ingrese que tipos de materiales busca.(Ejem: Carpinteria)\n=>");
            material = read.nextLine();
            System.out.println(almacen.consultarUbicacion(material));
            break;
    }
```

La opción 8 crea una una lista de articulos que se deben comprar, pero esta lista cuenta con una prioridad, por lo que siempre apareceran los articulos más urgentes de comprar. Si se necesita ver la lista, solo se escoge la primera opción.

```java
case 8: //Cajas
    System.out.print("\n---Lista de cosas a comprar---\n"+
                     "1. Mostrar\n"+
                     "2. Agregar a la fila\n"+
                     "=>");
    int menuCajas = read.nextInt(); read.nextLine();
    switch (menuCajas) {
        case 1:
            System.out.println(x:"\n---LISTA---");
            cajas.mostrar();
            break;
        case 2:
            System.out.print(s:"\tIngrese el articulo:\n\t=>");
            String articulo = read.nextLine();
            System.out.print(s:"\tIngrese la prioridad\n\t=>");
            int priori = read.nextInt(); read.nextLine();

            String prioridad = Integer.toString(priori);
            cajas.agregar(articulo, prioridad);
            break;
        default:
            break;
    }
```

Los grafos, de la opcion 9, tiene la primera opcion para ver los la ruta anterior, el cual se imprime a traves de un ciclo, la segunda opción pide los puntos a los que se debe ir, se solicita que se indique cuantos kilometros hay de un punto a otro o si no hay forma de llegar, esta matríz es entregada a un método.

```java
case 9: //Mapa
    System.out.print("---RUTAS DE CAMIONES---\n"+
                     "1.Ver ruta anterior\n"+
                     "2. Nueva ruta\n"+
                     "=>");
    int menuMapa = read.nextInt();
    switch (menuMapa) {
        case 2:
            System.out.println(x:"¿Cuantos puntos de parada son?");
            int paradas = read.nextInt();
            grafo = new float[paradas][paradas];
            for(int nodos = 0; nodos<grafo.length; nodos++){
                for(int i=0; i<grafo.length; i++){
                    System.out.println("El punto "+(nodos+1)+" tiene camina al punto "+(i+1)+
                                       "\n(No) Ingrese 0. (Si) Ingrese los kilometros");
                    float aristas = read.nextFloat();
                    grafo[nodos][i] = aristas;
                }
            }
            grafo = acomodorar(grafo);
        break;
```

```java
case 9: //Mapa
    System.out.print("---RUTAS DE CAMIONES---\n"+
                     "1.Ver ruta anterior\n"+
                     "2. Nueva ruta\n"+
                     "=>");
    int menuMapa = read.nextInt();
    switch (menuMapa) {
        case 2:
            System.out.println(x:"¿Cuantos puntos de parada son?");
            int paradas = read.nextInt();
            grafo = new float[paradas][paradas];
            for(int nodos = 0; nodos<grafo.length; nodos++){
                for(int i=0; i<grafo.length; i++){
                    System.out.println("El punto "+(nodos+1)+" tiene camina al punto "+(i+1)+
                                       "\n(No) Ingrese 0. (Si) Ingrese los kilometros");
                    float aristas = read.nextFloat();
                    grafo[nodos][i] = aristas;
                }
            }
            grafo = acomodorar(grafo);
        break;
```

Teniendo un resultado así:

```
Si se ha formado un cliente, ingrese su nombre:
=>Sharon
Cantidad de articulos que comprara:
=>2
Primer cliente a la fila
```

```
Si se ha formado un cliente, ingrese su nombre:
=>Luca
Cantidad de articulos que comprara:
=>5
```

```
Si se ha formado un cliente, ingrese su nombre:
=>Maria
Cantidad de articulos que comprara:
=>7
```

```
Si se ha formado un cliente, ingrese su nombre:
=>Sheyla
Cantidad de articulos que comprara:
=>15
```

```
Si se ha formado un cliente, ingrese su nombre:
=>Julian
Cantidad de articulos que comprara:
=>3
```

En mostrar, se permite imprimir a todos los clientes con un ciclo el cual accede al arreglo e imprime a todos los clientes en el orden de atención.

```
case 2:
    System.out.println(x:"===Clientes===");
    fila.mostrar();
```

```
public void mostrar(){
    if(size == 0){
        System.out.println(x:"Vacio...");
    }
    for(int i = 1; i<=size; i++){ ···
}
```

En tickets, estos ya están creados y ya tienen datos dentro, pues han sido actualizados conforme los clientes se forman. Todo el proceso se hace a partir de la clase Pila.

```
case 4:
    System.out.println("\n===TICKETS===\n"+
                       "1. Imprimir ultimo ticket.\n"+
                       "2. Eliminar ticket.\n"+
                       "3. Imprimir todos los tickets");
    int menuTickets = read.nextInt();read.nextLine();
```

En la clase se crean los atributos que contendrá el ticket más reciente, un arreglo donde estarán todos los tickets, la capacidad de este que ya esta definida, una lista que se actualiza y un numero que permite las veces que rotaran los empleados como una iteración.

```
//Tickets !!
class Pilas{//LIFO
    int superficie;
    String[] dato;
    int capacity;
    ListasEnlazadas empleados;

    int rotacion=0;
```

```
public Pilas(int capacity){
    this.capacity = capacity;
    this.dato = new String[capacity];
    this.superficie = -1;
}
```

Existen dos metodos los cuales verifican si esta vacio el arreglo para evitar errores.

```
public boolean isEmpty(){return superficie == -1;}
public boolean isFull(){return superficie == capacity-1;}
```

Y con push se puede agregar un string al arreglo el cual registra al cliente y quien le atendió de los empleados gracias al método getName de listasEnlazadas.

```
public void push(ListasEnlazadas empleados, String cliente){
  if(isFull()){
      System.out.println(x:"Limite de tickets alcanzados");
      return;
  }
  if(empleados.isCabezaEmpty()){
      System.out.println(x:"No hay empleados para atender");
      return;
  }
  dato[++superficie] = ("Cliente "+ cliente +" atendido por "+empleados.getName(this.rotacion))
  this.rotacion++;

}
```

Para imprimir el primer ticket, solo se verifica si no está vacío el arreglo, y se toma el atributo Superficie para imprimir el índice del último dato.

```
public void peek(){//Ultimo ticket
    if(isEmpty()){
        System.out.println(x:"Sin ventas registradas.");
        return;//Error
    }
    System.out.println(dato[superficie]);
    return;
}
```

En la primera opción está el ultimo cliente, que con el método peek, puedes ver el ultimo ticket registrado.

```
case 1:
    System.out.print(s:"Ultimo cliente \n=>");
    tickets.peek();
    break;
```

Con la segunda opción de eliminar, se elimina el ultimo registro dándole una resta a la superficie. Con esto se deja de acceder a ese ultimo registro.

```java
case 2:
    System.out.println(x:"===Eliminar el ultimo ticket===");
    tickets.pop();
    break;
```

```java
public void pop(){
        if(isEmpty()){
            System.out.println(x:"Sin ventas registradas");
            return; //Indicador de error
        }
        System.out.println(dato[superficie--]);
         return;
    }
```

Cuando se muestras todos los empleados, estos inician desde el más reciente hasta el más antiguo a través de un ciclo que empieza desde la superficie y termina en el ultimo dato.

```java
public void mostrar(){//ver todos lso tickets
    for(int i=superficie; i>=0; i--){
        System.out.println(dato[i]);
    }
}
```

```
Ultimo cliente
=>Cliente Julian atendido por Claudia
```

```
===TICKETS===
Cliente Julian atendido por Claudia
Cliente Sheyla atendido por Alfredo
Cliente Maria atendido por Claudia
Cliente Luca atendido por Alfredo
Cliente Sharon atendido por Alfredo
```

```
===Eliminar el ultimo ticket===
Cliente Julian atendido por Claudia
```

```
Ultimo cliente
=>Cliente Sheyla atendido por Alfredo
```

Proveedores

```
----Proveedores----
1. Ver proveedores
2. Ingresar proveedor
3. Eliminar proveedor por index
4. Buscar proveedor por index
=>2
Ingrese el nombre: Key
Ingrese el index: 10
Ingrese 2 articulos que provea: llaves
=> Destornillador
```

```
Ingrese el nombre: Juli
Ingrese el index: 15
Ingrese 2 articulos que provea: cinta
=> escaleras
```

```
Ingrese el nombre: macarena
Ingrese el index: 5
Ingrese 2 articulos que provea: Hojas
=> lapices
```

```
---Inorden---
        5: macarena => Hojas, lapices
        10: Key => llaves, Destornillador
        15: Juli => cinta, escaleras
---Postorden---
        5: macarena => Hojas, lapices
        15: Juli => cinta, escaleras
        10: Key => llaves, Destornillador
---Preorden---
        10: Key => llaves, Destornillador
        5: macarena => Hojas, lapices
        15: Juli => cinta, escaleras
```

```
4. Buscar proveedor por index
=>4
==Buscar proveedor por index===
Ingrese el index: 5
        5: macarena => Hojas, lapices
```

```
----Proveedores----
1. Ver proveedores
2. Ingresar proveedor
3. Eliminar proveedor por index
4. Buscar proveedor por index
=>3
==Elimnar proveedor por index===
Ingrese el index: 15
```

```
---Inorden---
        5: macarena => Hojas, lapices
        10: Key => llaves, Destornillador
---Postorden---
        5: macarena => Hojas, lapices
        10: Key => llaves, Destornillador
---Preorden---
        10: Key => llaves, Destornillador
        5: macarena => Hojas, lapices
```

Lista de entregas

```
---Division de entregas complejas---
Ingrese cuantos pedidos son: 4
Ingrese lo que entregara
=>Escaleras
=>Martillos
=>Clavos
=>Cinta metrica
El problema acomodado
Cinta metrica, Escaleras, Martillos, Clavos,
```

Tablas hash / Dirección de los articulos por categoría

```
---Almacen---
1. Agregar ubicacion
2.buscar ubicacion
=>1
Ingrese los tipos de herramientas(Ejem: Carpi)Carpinteria
Ingrese una breve descripcion de donde esta
=>Pasillo 2, estanteria 3
```

```
---Almacen---
1. Agregar ubicacion
2.buscar ubicacion
=>2
Ingrese que tipos de materiales busca.(Ejem: Carpinteria)
=>Carpinteria
Ubicación de Carpinteria: Pasillo 2, estanteria 3
```

Lista de compras

```
---Lista de cosas a comprar---
1. Mostrar
2. Agregar a la fila
=>2
        Ingrese el articulo:
        =>Escaleras
        Ingrese la prioridad
        =>10
1: Martillos
```

```
---Lista de cosas a comprar---
1. Mostrar
2. Agregar a la fila
=>2
        Ingrese el articulo:
        =>Clavos
        Ingrese la prioridad
        =>5
1: Martillos
10: Escaleras
```

```
---Lista de cosas a comprar---
1. Mostrar
2. Agregar a la fila
=>1

---LISTA---
1: Martillos
10: Escaleras
```

Rutas / Grafos

```
---RUTAS DE CAMIONES---
1.Ver ruta anterior
2. Nueva ruta
=>2
¿Cuantos puntos de parada son?
3
El punto 1 tiene camina al punto 1
(No) Ingrese 0. (Si) Ingrese los kilometros
0
El punto 1 tiene camina al punto 2
(No) Ingrese 0. (Si) Ingrese los kilometros
7
El punto 1 tiene camina al punto 3
(No) Ingrese 0. (Si) Ingrese los kilometros
0
El punto 2 tiene camina al punto 1
(No) Ingrese 0. (Si) Ingrese los kilometros
7
El punto 2 tiene camina al punto 2
(No) Ingrese 0. (Si) Ingrese los kilometros
0
El punto 2 tiene camina al punto 3
(No) Ingrese 0. (Si) Ingrese los kilometros
8
El punto 3 tiene camina al punto 1
(No) Ingrese 0. (Si) Ingrese los kilometros
0
El punto 3 tiene camina al punto 2
(No) Ingrese 0. (Si) Ingrese los kilometros
8
El punto 3 tiene camina al punto 3
(No) Ingrese 0. (Si) Ingrese los kilometros
0
```

```
9
---RUTAS DE CAMIONES---
1.Ver ruta anterior
2. Nueva ruta
=>1
0.0     1.0     1.0
1.0     0.0     1.0
1.0     1.0     0.0
```

Conclusión

La estructura de datos es principalmente la forma en poder manejarlos como una base que conserva y guarda, aunque principalmente se dedica a la modificación y accesibilidad a estos.

Cada uno depende de su funcionamiento para saber cómo y con qué tipos de ejemplos se puede usar, que de hecho, pueden mezclarse entre estos, pues la mayoría puede manejarse con un arreglo y tambien con los enlaces(primer tema que vimos).

Aquello que me llamo más la atención de las estructuras, es que no necesariamente son nuevos objetos o métodos para trabajar con los mimos datos, si no más bien el seguimiento de reglas, ya sea por condicionales o por métodos recursivos los cuales hacen que los datos trabajen y se comporten de cierta forma.

Con las estructuras que más me quedare, es con las listas enlazadas y los tipos de ordenamiento, pues son soluciones que ya había pensado en algún momento, sin embargo, no los había sabido aplicar o no sabía de su existencia.

El método que más me costo entender fue el de hash table, por lo que sera la unica estructura que me llevare de tarea y la estructura que más problemas me trajo, fue el de arboles binarios, pues en mi código aun se puede notar que es la estructura, que por más largo que fue, tiene varios usos que considero básicos y errores que, aunque no me parecen de gran importancia para un ejemeplo a baja escala, en un caso real, podría llegar a ser la piedra en el zapato.