

INSTITUTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:

ESTRUCTURA DE DATOS

TEMA:

REPORTE DE PRACTICA TEMA 1 INTRODUCCION A LAS ESTRUCTURAS DE DATOS

NOMBRE DE LOS INTEGRANTES:

HERNÁNDEZ DÍAZ ARIEL - 21010195

PALACIOS MENDEZ XIMENA MONSERRAT - 21010209

CARRERA:

INGENIERIA INFORMATICA

GRUPO:

3a3A

FECHA DE ENTREGA:

17 DE FEBRERO DE 2023

INTRODUCCION

Una estructura de datos es un conjunto de elementos dispuestos en un orden específico. La mayoría de los algoritmos que desarrollaremos se centrarán en la repetición de un conjunto de acciones sobre una estructura de datos. Ahí radica la importancia de saber trabajar con las secuencias. Sin ir más lejos, en asignaturas anteriores ya habrás estudiado los esquemas de recorrido y búsqueda en una estructura de datos.

Pero además, aparte de la importancia del tratamiento de las estructuras de datos, en este módulo averiguaremos cómo se almacenan de manera que posteriormente podamos acceder a ellas secuencialmente en los datos ordenados y datos desordenados. Un tipo de datos consta de un conjunto de valores y de una serie de operaciones que pueden aplicarse a esos valores. Las operaciones deben cumplir ciertas propiedades que determinarán su comportamiento.

Las operaciones necesarias para trabajar con una secuencia son:

- Crear un array vacío.
- Insertar un elemento dentro del array.
- Borrar un elemento del array.
- Consultar un elemento del array.

El comportamiento que se establezca para cada una de las operaciones (¿Dónde se inserta un elemento nuevo? ¿Qué elemento se borra? ¿Qué elemento se puede consultar?) definirá el tipo de datos que necesitaremos. Un tipo abstracto de datos (abreviadamente TAD) es un tipo de datos al cual se ha añadido el concepto de abstracción para indicar que la implementación de la estructura de datos es invisible para los usuarios aplicada a los algoritmos de datos ordenados y desordenados.

Un TAD cualquiera constará de dos partes:

- Especificación. En la que se definirá completamente y sin ambigüedades el comportamiento de las operaciones del tipo.

- Implementación. En la que se decidirá una representación para los valores del tipo y se codificarán las operaciones a partir de esta representación.

Dado un tipo, podemos hallar varias implementaciones, cada una de las cuales deberá seguir la especificación del tipo.

De este modo, el único conocimiento necesario para usar un TAD es la especificación, ya que explica las propiedades o el comportamiento de las operaciones del tipo.

Las operaciones del TAD se dividirán en constructoras (devuelven un valor del mismo tipo) y consultoras (devuelven un valor de otro tipo).

Dentro de las operaciones constructoras, se denominarán generadoras las que formen parte del conjunto mínimo de operaciones necesarias para generar cualquier valor del tipo, mientras que el resto se denominarán modificadoras.

En este módulo nos centraremos en los tres TAD más típicos para representar las secuencias:

- Pilas
- Colas
- Listas

La explicación de cada tipo seguirá la misma estructura. Primero, se presentará el tipo de una manera intuitiva para entender el concepto. A continuación, se describirá el comportamiento de las operaciones del tipo de una manera formal. Y, para acabar, se desarrollará una implementación del tipo estructura de datos acompañada de algún ejemplo de algoritmo aplicando datos ordenados y desordenados.

MARCO TEÓRICO

La representación de la información es fundamental en ciencias de la computación y en informática.

El propósito principal de la mayoría de los programas de computadoras es almacenar y recuperar información, además de realizar cálculos.

De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y de los algoritmos que las manipulan constituye el núcleo central de la informática y de la computación.

Los lenguajes de programación tradicionales proporcionan tipos de datos para clasificar diversas clases de datos. Las ventajas de utilizar tipos en el desarrollo de software son:

- Apoyo y ayuda en la prevención y en la detección de errores.
- Apoyo y ayuda a los desarrolladores de software, y a la comprensión y organización de ideas acerca de sus objetos.
- Ayuda en la identificación y descripción de propiedades únicas de ciertos tipos.

Los tipos de Datos.

Los tipos son un enlace importante entre el mundo exterior y los elementos de datos que manipulan los programas. Su uso permite a los desarrolladores limitar su atención a tipos específicos de datos, que tienen propiedades únicas. Por ejemplo, el tamaño es una propiedad determinante en los arrays y en las cadenas; sin embargo, no es una propiedad esencial en los valores lógicos, verdadero (true) y falso (false).

Definición 1: Un tipo de dato es un conjunto de valores y operaciones asociadas a esos valores.

Definición 2: Un tipo de dato consta de dos partes: un conjunto de datos y las operaciones que se pueden realizar sobre esos datos.

En los lenguajes de programación hay disponible un gran número de tipos de datos. Entre ellos se pueden destacar:

- los tipos primitivos de datos.
- los tipos compuestos.
- los tipos agregados o abstractos o estructuras de datos.

Tipos Primitivos de Datos

Los tipos de datos más simples son los tipos de datos primitivos, también denominados datos atómicos porque no se construyen a partir de otros tipos y son entidades únicas no descomponibles en otros.

Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas. Estas propiedades diferencian un tipo de dato atómico de otro. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

Un tipo de dato atómico es:

- Un conjunto de valores, y
- Un conjunto de operaciones sobre esos valores.

Tipos de Datos Compuestos

Los datos compuestos son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado.

En algunas ocasiones los datos compuestos se conocen también como datos o tipos agregados. Los tipos agregados son tipos de datos cuyos valores constan de colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos.

Existen tres tipos agregados básicos:

- Arreglos (arrays) y Matrices (tablas)
- Registros
- Secuencias de texto o cadenas.

Tipos Agregados o Estructuras de Datos Abstractas

Una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Bajo esta óptica, cualquier representación de datos, incluso un número entero o un número de punto (coma) flotante almacenado en el computador, es una sencilla estructura de datos.

En un sentido más específico, una estructura de datos es una organización o estructuración de una colección de elementos de datos. Así, una lista ordenada de enteros almacenados en un array es un ejemplo de dicha estructuración.

"Una estructura de datos es una agregación de tipos de datos compuestos y atómicos en un conjunto con relaciones bien definidas. Una estructura significa un conjunto de reglas que contienen los datos juntos."

Las estructuras de datos pueden estar anidadas: se puede tener una estructura de datos que conste de otras.

Estructura de datos es:

Una combinación de elementos en la que cada uno es o bien un tipo de dato u otra estructura de datos. Un conjunto de asociaciones o relaciones (estructura) que implica a los elementos combinados.

Tipos de datos abstractos (TDA)

Un Tipo de dato abstracto (en adelante TDA) es un conjunto de datos u objetos al cual se le asocian operaciones. El TDA proporciona una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en cómo están implementadas dichas operaciones. Esto quiere decir que un mismo TDA

puede ser implementado utilizando distintas estructuras de datos y proporcionar la misma funcionalidad.

El paradigma de orientación a permite el encapsulamiento de los datos y las operaciones mediante la definición de clases e interfaces, lo cual permite ocultar la manera en cómo ha sido implementado el TDA y solo permite el acceso a los datos a través de las provistas por la interfaz.

TDA lista

Una lista se define como una serie de N elementos E_1, E_2, \dots, E_N , ordenados de manera consecutiva, es decir, el elemento E_k (que se denomina elemento k -ésimo) es anterior al elemento E_{k+1} . Si la lista contiene 0 elementos se denomina lista vacía.

Las operaciones que se pueden realizar en la lista son: insertar un elemento en la posición k , borrar el k -ésimo elemento, un elemento dentro de la lista buscar y preguntar si la lista está vacía.

Una manera simple de implementar una lista es usar un arreglo. Sin embargo, las operaciones de inserción y borrado de elementos en arreglos son ineficientes, puesto que para insertar un elemento en la parte media del arreglo es necesario mover todos los elementos que se encuentran delante de él, para hacer espacio, y al borrar un elemento es necesario mover todos los elementos para ocupar el espacio desocupado. Una implementación más eficiente del TDA se logra utilizando listas enlazadas.

A continuación se presenta una implementación en Java del TDA utilizando listas enlazadas y sus operaciones asociadas:

- `estaVacia()` : devuelve verdadero si la lista está vacía, falso en caso contrario.
- `insertar(x, k)` : inserta el elemento x en la k -ésima posición de la lista.
- `buscar(x)` : devuelve la posición en la lista del elemento x .
- `buscarK(k)` : devuelve el k -ésimo elemento de la lista.
- `eliminar(x)` : eliminar de la lista el elemento x .

En la implementación con listas enlazadas es necesario tener en cuenta algunos detalles importantes: si solamente se dispone de la referencia al primer elemento, el añadir o remover en la primera posición es un caso especial, puesto que la referencia a la lista enlazada debe modificarse según la operación realizada. Además, para eliminar un elemento en particular es necesario conocer el elemento que lo antecede, y en este caso, ¿Qué pasa con el primer elemento, que no tiene un predecesor?

Para solucionar estos inconvenientes se utiliza la implementación de lista enlazada con nodo cabecera. Con esto, todos los elementos de la lista tendrán un elemento anterior, puesto que el anterior del primer elemento es la cabecera. Una lista vacía corresponde, en este caso, a una cabecera cuya referencia siguiente es nula.

La interfaz de este TDA proporciona las siguientes operaciones:

- *apilar(x): inserta el elemento x en el tope de la pila (push en inglés).
- *desapilar(): devuelve el elemento que se encuentra en el tope de la pila y lo elimina de ésta (pop en inglés).
- *tope(): devuelve el elemento que se encuentra en el tope de la pila, pero sin eliminarlo de ésta (top en inglés).
- estaVacia(): retorna verdadero si la pila no contiene elementos, falso en caso contrario (isEmpty en inglés).
- Nota: algunos autores definitivamente desapilar como sacar el elemento del tope de la pila sin devolverlo.

MANEJO DE MEMORIA

Todas las variables, arreglos y objetos en general tienen una duración determinada en el transcurso de un programa. Son creados y destruidos para su uso y después para que la memoria sea liberada para que la utilicen otros objetos.

En java existen tres formas de usar la memoria para almacenar valores, son:

a) Memoria estática: Es la utilizada por variables globales y las declaradas de tipo static. Estos objetos tienen enfocado la misma dirección de memoria desde el comienzo hasta el final del programa.

(variables globales, tipo static)

b) Memoria automática: Es la utilizada por argumentos en una función y por las variables locales. Cada entrada en la función crea estos objetos y son destruidos al salir de ella.

(variables locales, argumentos en una función).

c) Memoria Dinámica: Es también llamado almacenamiento libre porque en este caso el programador es el que solicita memoria para crear los objetos y es el responsable de liberar la memoria cuando ya no la necesita para ser reutilizada.

(objetos de una clase)

La reserva y liberación para variables globales, estáticas, locales y argumentos son realizados de forma involucreada por el programa, la única que requiere intervención del programador es la reserva y liberación de memoria dinámica.

ANALISIS DE ALGORITMOS

Es una parte importante de la Teoría de complejidad computacional más amplia, que provee estimaciones teóricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Estas estimaciones resultan ser bastante útiles en la búsqueda de algoritmos eficientes.

A la hora de realizar un análisis teórico de algoritmos es común calcular su complejidad en un sentido asintótico, es decir, para un tamaño de entrada suficientemente grande. La cota superior asintótica, y las notaciones omega (cota inferior) y theta (caso promedio) se usan con esa finalidad. Por ejemplo, la búsqueda binaria decimos que se ejecuta en una cantidad de pasos proporcional a un logaritmo, en $O(\log(n))$, coloquialmente "en tiempo logarítmico". Normalmente las estimaciones asintóticas se utilizan porque diferentes implementaciones del mismo algoritmo no tienen por qué tener la misma eficiencia. No obstante la eficiencia de

dos implementaciones "razonables" cualesquiera de un algoritmo dado están relacionadas por una constante multiplicativa llamada constante oculta.

La medida exacta (no asintótica) de la eficiencia a veces puede ser computada pero para ello suele hacer falta aceptar supuestos acerca de la implementación concreta del algoritmo, llamada modelo de computación. Un modelo de computación puede definirse en términos de un ordenador abstracto, como la Máquina de Turing, y/o postulando que ciertas operaciones se ejecutan en una unidad de tiempo. Por ejemplo, si al conjunto ordenado al que aplicamos una búsqueda binaria tiene n elementos, y podemos garantizar que una única búsqueda binaria puede realizarse en un tiempo unitario, entonces se requieren como mucho $\log_2 N + 1$ unidades de tiempo para devolver una respuesta.

Las medidas exactas de eficiencia son útiles para quienes verdaderamente implementan y usan algoritmos, porque tienen más precisión y así les permite saber cuánto tiempo pueden suponer que tomará la ejecución. Para algunas personas, como los desarrolladores de videojuegos, una constante oculta puede significar la diferencia entre éxito y fracaso.

Las estimaciones de tiempo dependen de cómo definamos un paso. Para que el análisis tenga sentido, debemos garantizar que el tiempo requerido para realizar un paso esté acotado superiormente por una constante. Hay que mantenerse precavido en este terreno; por ejemplo, algunos análisis cuentan con que la suma de dos números se hace en un paso. Este supuesto puede no estar garantizado en ciertos contextos. Si por ejemplo los números involucrados en la computación pueden ser arbitrariamente grandes, dejamos de poder asumir que la adición requiere un tiempo constante (usando papel y lápiz, compara el tiempo que necesitas para sumar dos enteros de 2 dígitos cada uno y el necesario para hacerlo con enteros de 1000 dígitos).

Las tablas hash de direccionamiento abierto pueden almacenar los registros directamente en el array. Las colisiones se resuelven mediante un sondeo del array, en el que se buscan diferentes localidades del array (secuencia de sondeo) hasta

que el registro es encontrado o se llega a una casilla vacía, indicando que no existe esa llave en la tabla.

Ejemplo de direccionamiento abierto.

Las secuencias de sondeo más utilizadas son:

1. Sondeo lineal

En el que el intervalo entre cada intento es constante (frecuentemente 1). El sondeo lineal ofrece el mejor rendimiento del caché, pero es más sensible al aglomeramiento.

2. Sondeo cuadrático

En el que el intervalo entre los intentos aumenta linealmente (por lo que los índices son descritos por una función cuadrática). El sondeo cuadrático se sitúa entre el sondeo lineal y el doble hasheo.

3. Doble hasheo

En el que el intervalo entre intentos es constante para cada registro pero es calculado por otra función hash. El doble hasheo tiene pobre rendimiento en el caché pero elimina el problema de aglomeramiento. Este puede requerir más cálculos que las otras formas de sondeo.

EJEMPLOS DE UN TDA

Una calculadora es un ejemplo de un TDA que maneja objetos numéricos y las operaciones aritméticas sobre dichas cantidades.

Usa el sistema decimal para las cantidades y realiza operaciones de suma, resta, multiplicación, etc. Sin embargo, ¿Sabes cómo una calculadora representa las cantidades internamente? ¿En binario? ¿Decimal? , ¿Palitos? o ¿piedritas?. NO!, no lo sabemos y tampoco nos falta para usar la calculadora.

Un sistema de numeración es un ejemplo de un tipo de dato abstracto que representa el concepto de cantidad.

Los siguientes son ejemplos de sistemas de numeración:

- Romano: I, V, X, L, C, D, M;
- decimales: 0, 1, 2, 3, ..., 9;
- Maya: _ _ . _

Las características más importantes son:

- “crear una protección de las entidades representadas”, es decir: Oculta la representación e implementación de la entidad y sus operaciones. No sabemos cómo la calculadora representa las cantidades ni como realiza la operación de sumar.

- -Sólo se manipula a través de sus operaciones.

Nosotros solo podemos realizar las operaciones que la calculadora le permite realizar y nada más.

¿Cómo logramos crear dicha protección?. Según la fórmula de Wirth definimos un programa así:

Programa = Datos + Algoritmos

Si podemos separar e identificar en un algoritmo las instrucciones que manipulan los datos de las instrucciones que indican control, entonces podemos reescribir la fórmula como:

- Programa = Datos + Algoritmos-Datos + Algoritmo-Control
- Donde TDA = Datos + Algoritmos-Datos, y por tanto la fórmula queda así:
- Programa = TDA + Algoritmo de Control

Ventajas de uso de un TDA

- Herramienta para resolver problemas (nivel de abstracción)
- Independencia entre la forma de representar la información y la solución del problema → portabilidad de la solución.
- Favorece la supervisión, verificación y depuración de programas.
- Contribuye a la flexibilidad de los cambios.

COMPETENCIA(S) ESPECÍFICA(S):

Conoce y comprende las diferentes estructuras de datos, su clasificación y forma de manipularlas para buscar la manera más eficiente de resolver problemas.

RECURSOS, MATERIALES Y EQUIPO

- Computadora
- Java
- Lectura de los materiales de apoyo del tema 1 (PADLET)
- Notas de clase (problemas resueltos en clase y materiales de trabajo de la profesora)

CLASE DATOS ORDENADOS

```
public class datoOrd {
```

En esta clase se tienen 5 métodos, los cuales nos servirán para ingresar, buscar, eliminar, imprimir y modificar elementos de un arreglo, estos se explicarán más adelante. Para esto vamos a declarar 2 atributos privados, el primero será un arreglo el cual nos ayudara a manejar las operaciones anteriores, y el segundo es el subíndice:

```
private int datos[];  
private byte p;
```

Igual creamos un constructor el cual en el parámetro recibe el tamaño para dimensionar (reservar), dentro del constructor creamos el arreglo tomando el tamaño (que recibió en el parámetro), seguido de esto, "p" (subíndice) -1 para validar si está vacío el array.

```

public datoOrd(byte tam) {

    datos= new int[tam];
    p=-1;

}

```

METODO VALIDA VACIO

En este método de tipo booleano nos va a regresar un true o un false si el array esta vacío. En caso de que no este vacío nos dará chance de hacer cualquier operación antes mencionada. Este es el método de validaVacio:

```

public boolean validaVacio()
{
    return (p==-1);
}

```

METODO IMPRIME DATOS

Bueno, dicho método lo que hará es que imprimirá los datos antes ingresados al array, solo será usar un ciclo for para que "i" llegue hasta la última posición, y así concatene la posición y el dato para cada posición, así finalmente retorne la cadena que se concateno y la muestre en pantalla:

```

public String imprimeDatosOrd()
{
    String cad="";
    for (int i = 0; i <=p; i++) {
        cad+=i+"["+datos[i]+"] "+"\\n";
    }
    return "\\n"+cad;
}

```

METODO BUSCARSECORDENADO

En este método lo que se busca es que busque de forma ordena algún elemento existente en el array, en caso contrario nos dirá que el dato que buscamos no existe, de tal forma que el código queda de esta manera:

```

public byte buscarSecOrdenado(int dat) {
    byte i=0;
    while(i<=p && datos[i] < dat)
        i++;
    return (byte) ((i>p || datos[i] > dat)?-i:i);
}

```

METODO ELIMINAR ORDENADO

Aquí vamos a eliminar un elemento del arreglo, por medio del parámetro ya que este nos pedirá un valor. Ya que lo encontró va a eliminar el subíndice y por obvia razón consigo el elemento, y después lo que hará será como recorrer los elementos. Este es el método:

```

public void eliminarOrd(byte pos) {
    for(int k=pos; k<=p; k++) {
        datos[k]=datos[k+1];
    }
    p--;
}

```

METODO RECORRER ORDENADO

```

public void recorrePos(byte pos) {
    for(int j= p+1; j>pos; j--) {
        datos[j]=datos[j-1];
    }
}

```

Este método recorre de forma ordenada, ya que en caso de eliminar va a recorrer las posiciones de forma ordenada. Por esa razón la J decrementa.

METODO AGREGAR ORDENADO

```

public void AgregaOr() {
    if(validaVacio()){
        datos[p+1]= ToolsPanel.leerInt("Ingresa un valor");
        p++;
    }else{
        int dato = ToolsPanel.leerInt("Escribe el valor a insertar");
        byte pos = buscarSecOrdenado( dato);
        if(pos>=0)
            ToolsPanel.imprimeErrorMsje("Dato existente");
        else{
            pos*=-1;
            recorrePos((byte) pos);
            p++;
        }
        datos[pos] = dato;
    }
}

```

El método agregar ordenado lo que hará es que ingresemos datos en caso de que el arreglo este vacío, en caso contrario nos seguirá pidiendo que ingresemos más valores, pero aquí curioso es que en caso de ingresar un mismo valor nos arrojará que el dato ya existe, por ejemplo en la primera vez ingresamos el número 1, y para la otra vuelta volvemos a digitar el 1, pero en caso de ingresar un número diferente lo añade al arreglo y hace el llamado de **recorrePos** para recorrer el arreglo de forma ordenada, no sin antes incrementar el subíndice.

METODO MODIFICAR ORDENADO

```

public void modificarOrd(byte existe) {
    int dato;
    if(existe==0)
    {
        if(existe<=(Integer)datos[0]){
            do
            {
                dato = ToolsPanel.leerByte("Ingresa un valor menor a: " +
                    datos[existe + 1]);
            }
            while (dato >= (Integer) datos[existe*(-1)+1]);
            datos[existe*(-1)] = dato;
        }
    }
}

```

Como parámetro usaremos una variable auxiliar llamada existe, esta hará que para cuando nosotros queramos cambiar/modificar el primer elemento de la posición 0 (por eso cuando existe sea exactamente igual a cero, quiere decir que cuando el

elemento se encuentre en la posición 0 lo ubicara y automáticamente lo modificara) este mismo lo cambiara, eso si menor a la posición. Por ejemplo tenemos una lista de esta manera:

[0] 1

[1] 2

[2] 3

[3] 4

[4] 5

Si nosotros queremos modificar el 1, entonces va a pedir que lo cambiemos a un valor menor a existe +1 (esta parte va a incrementar el valor que recibió en el parámetro, por decir si EXISTE VALE 1, ESTONCES SUMA 1 +1, POR LO CUAL CUANDO PIDE UN NUMERO MENOR TIENE QUE SER MENOR AL DEL ELEMENTO QUE VA A SER CAMBIADO). **SI INGRESAMOS UN DATO ERRONEO NOS VOLVERA A SALIR LA MISMA PANTALLA.**

```
else
{
    do
    {
        dato = ToolsPanel.leerByte("Ingresa un valor menor a: '" +
            datos[existe + 1] +
            "' mayor a: '" + datos[existe - 1] + "'");
    }
    while (dato >= (Integer) datos[existe + 1] &&
        existe <= (Integer) datos[existe - 1]);
    datos[existe] = dato;
```

Activar W

Por otro lado, en caso de que queramos cambiar alguno de los datos del centro se hará lo mismo, solo que en esta parte se pedirá que el nuevo dato sea menor a lo que hay en **EXISTE** y también que sea mayor a lo que hay en **EXISTE**, **SI INGRESAMOS UN DATO ERRONEO NOS VOLVERA A SALIR LA MISMA PANTALLA.** Por último, si queremos modificar el ultimo dato del arreglo, tiene que ser mayor a lo que hay en **EXISTE** entonces si es así lo cambiará, sino seguirá pidiéndonos que ingresemos un valor mayor que hay en **EXISTE**.

```

    }
}
else
{
    do
    {
        dato = ToolsPanel.leerByte("Ingresa un valor mayor a: " +
        datos[existe - 1]);
    }
    while (dato <= (Integer) datos[existe - 1]);
    datos[existe] = dato;
}
}

```

CLASE MENU ORDENADO

Para mostrar uno y cada uno de los métodos anteriores vamos a hacer el uso de un menú, empezaremos con que vamos a crear un objeto llamado **Obj** de la misma clase, seguido de esta vamos a darle un tamaño de 10 (como sabemos que en la clase de **datos ordenados** en el constructor, nos pedía un tamaño en el parámetro), podemos darle otro tamaño de 15, 20 o 25, etc.

```

public class Menu {
    public static void menu3(String menu) {
        String sel="";
        datoOrd obj = new datoOrd((byte)10);
        byte pos=0;
    }
}

```

Después hacemos un **do-while** y dentro de este un **switch** para que este mismo se repita las veces que sean hasta que demos la opción de salir. Explicaremos el primer caso:

CASO AGREGAR ORDENADO

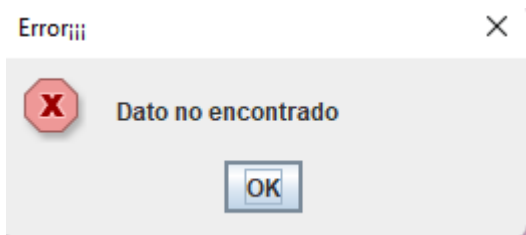
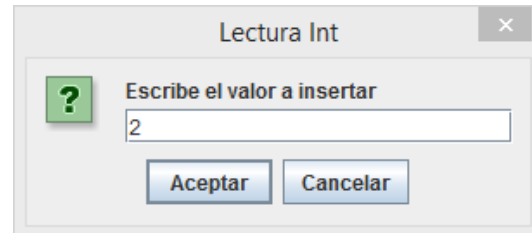
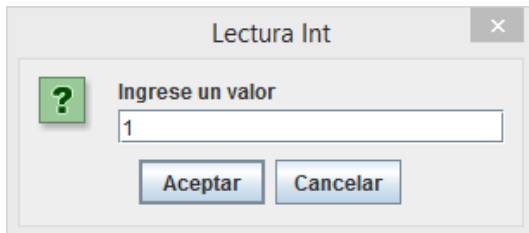
```

do {
    sel=boton(menu);
    switch(sel){
        case "AgregarOr":
            obj.AgregaOr();
    }
}

```

Aquí tenemos al método agregar ordenado, por lo visto este mismo ya lo tenemos en otra clase, así que solo por medio del objeto (**obj**) lo mandamos a llamar para insertar elementos al array.

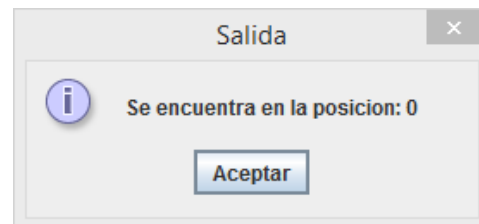
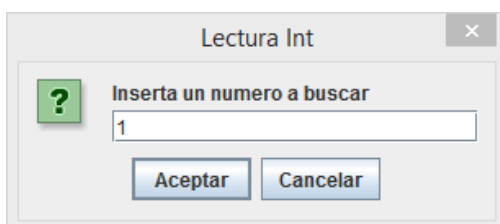
EJECUTABLES DE AGREGAR ORDENADO

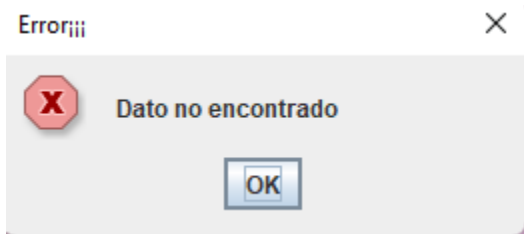
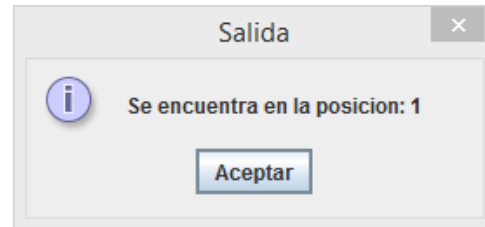
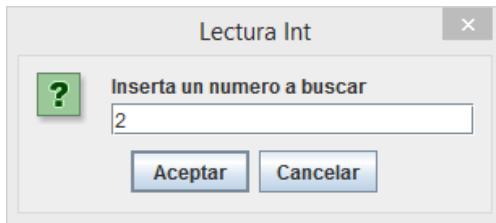


```
case "BuscarOr":
    if(obj.validaVacio()) {
        ToolsPanel.imprimeErrorMsje("Array vacio");
    }
    else {
        pos = obj.buscarSecOrdenado(ToolsPanel.leerInt("Inserta un numero a buscar"));
        if(pos>=0) {
            ToolsPanel.imprime("Se encuentra en la posicion: " +pos);
        }else {
            ToolsPanel.imprimeErrorMsje("Dato no encontrado");
        }
    }
    break;
```

Como primera instancia, vamos a validar si el array esta vacío, en caso contrario insertamos el numero que vamos a buscar en el arreglo, si POS llega ser mayor a 0 nos dirá en que posición se encuentra, sino el dato no existe o no se encontró.

EJECUTABLES BUSCAR ORDENADOS



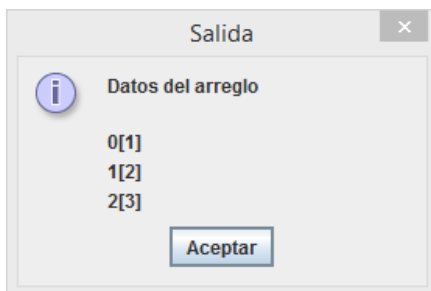


CASO IMPRIMIR ORDENADOS

```
case "Imprimir":  
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");  
    else ToolsPanel.imprime("Datos del arreglo\n"+obj.imprimeDatosOrd());  
  
    break;
```

Aquí volvemos a validar si esta vacío, si lo esta nos dirá que esta vacío, en caso contrario nos va a mostrar los datos del arreglo (para eso necesitamos llamar al método de imprimeDatos que esta en otra clase)

EJECUTABLES IMPRIME DATOS



CASO ELIMINAR ORDENADOS

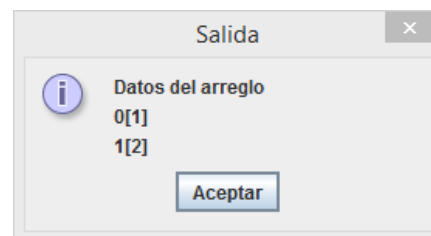
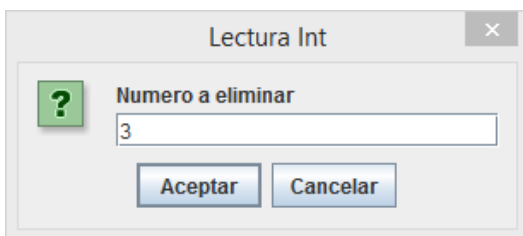
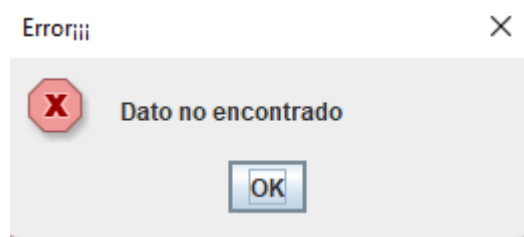
```

case "Eliminar":
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");
    else
    {
        pos = obj.buscarSecOrdenado(ToolsPanel.leerInt("Numero a eliminar"));
        if(pos>=0)
        {
            obj.eliminarOrd(pos);
            ToolsPanel.imprime("Datos del arreglo"+obj.imprimeDatosOrd());
        }
        else {
            ToolsPanel.imprimeErrorMsje("Dato no encontrado");
        }
    }
    break;

```

De nueva cuenta volvemos a validar si el array esta vacío, si esta vacío nos mandara un mensaje de que lo está, sino por medio de la variable **POS** nos pedirá ingresar el numero que vamos a eliminar entonces si **POS ES MAYOR O IGUAL A CERO**, mandamos a llamar al método de **eliminar ordenado** y este mismo recibe como parámetro a **POS** para eliminarlo. Por consiguiente, nos mostrara los datos del arreglo actualizados. Nota: en caso de que **POS** sea menor a cero o se pase del tope hasta donde se encuentra el último elemento del arreglo, es como nos dirá que el dato no existe o no se ha encontrado.

EJECUTABLES ELIMINA DATOS

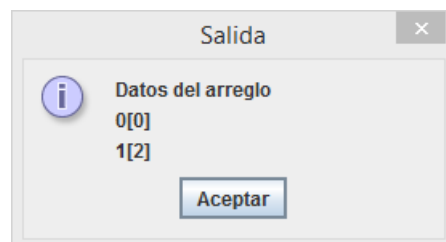
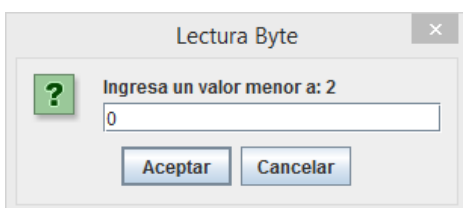
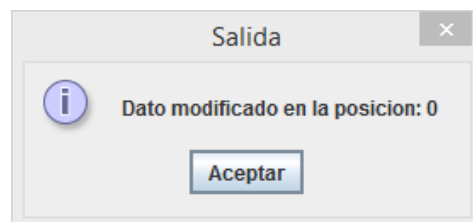
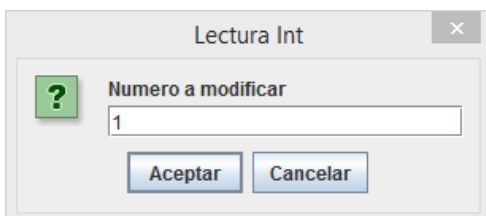


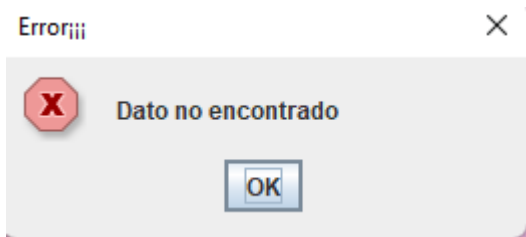
CASO MODIFICAR ORDENADOS

```
case "Modificar":
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");
    else
    {
        pos = obj.buscarSecOrdenado(ToolsPanel.leerInt("Numero a modificar"));
        if(pos >= 0)
        {
            ToolsPanel.imprime("Dato modificado en la posicion: " + pos);
            obj.modificarOrd(pos);
            ToolsPanel.imprime("Datos del arreglo"+obj.imprimeDatosOrd());
        }
        else ToolsPanel.imprimeErrorMsje("Dato encontrado");
    }
    break;
```

De nueva cuenta volvemos a validar si el array está vacío, si está vacío nos mandará un mensaje de que lo está, sino por medio de la variable **POS** nos pedirá ingresar el número que vamos a modificar entonces si **POS ES MAYOR O IGUAL A CERO**, antes de llamar al método vamos a mostrar la posición del dato que se modificó, ya ahora si mandamos a llamar al método de **modificar ordenado** y este mismo recibe como parámetro a **POS** para eliminarlo. Por consiguiente, nos mostrara los datos del arreglo actualizados. Nota: en caso de que **POS** sea menor a cero o se pase del tope hasta donde se encuentra el último elemento del arreglo, es como nos dirá que el dato no existe o no se ha encontrado.

EJECUTABLES MODIFICA DATOS





CASO SALIR

```
case "Salir":  
    break;
```

Por último en este caso, solo si queremos Salir la elegimos y automáticamente el programa termina.

Finalmente en el **MAIN** llamamos al menú, seguido de los nombres de cada **CASE**

```
public static void main(String[] args) {  
  
    menu3("AgregarOr, BuscarOr, Imprimir, Eliminar, Modificar, Salir");  
  
}  
}
```

Como algo importante tenemos también el **botón**, el cual se encarga de mostrar la ventana del menú

```
public static String boton(String menu) {  
    String valores[]=menu.split(",");  
    int n;  
    n = JOptionPane.showOptionDialog(null, "SELECCIONA DANDO CLICK ",  
        " M E N U", JOptionPane.NO_OPTION, JOptionPane.QUESTION_MESSAGE, null,  
        valores, valores[0]);  
    return ( valores[n]);  
}
```

CLASE DATOS DESORDENADOS

Para nuestro ejemplo empezaremos creando una clase llamada dato simple ,para luego crear un arreglo llamado datos, y una variable privada de tipo byte p, y empezar con la serie de métodos.

El siguiente método llamado DatoSimple con una variable de tipo byte llamada tan y con el arreglo se agrega dentro de este la variable tan , y p disminuye en 1,este método nos sirve para tener datos dentro del arreglo.

```
public class datosDesordenados {  
    private Integer datos[];  
    private byte p;  
    public datosDesordenados(byte tam)  
    {  
        datos = new Integer[tam];  
        p=-1;  
    }  
}
```

El siguiente método Valida vacío ,como su nombre nos indica nos ayuda a indicar si el arreglo está vacío ,retornando nos que p es exactamente igual a -1

```
public boolean validaVacio()  
{  
    return (p==-1);  
}
```

El método Almacenar dato nos ayuda a almacenar datos dentro del arreglo, en el cual si p qué es lo que está en el arreglo es menor a los datos dentro del arreglo el arreglo ira aumentando y podremos ingresar dato ,y si no se mandara un mensaje de que el arreglo está lleno


```

public void almacenarDatos()
{
    if(p< datos.length)
    {
        datos[p+1] = ToolsPanel.leerInt("Escribe un numero");
        p++;
    }
    else
        ToolsPanel.imprimeErrorMsje("Arreglo Lleno.");
}

```

Para imprimir los datos que se encuentran dentro del arreglo ocupamos el siguiente método en el cual utilizamos una nueva variable de tipo carácter llamada **cad** ,para luego entrar a un for donde inicializando i con 0 y mientras i sea menor a p ,incrementará en i ,para luego sumar lo que se encuentra y guardándola en **cad** donde agregamos lo que hay en el arreglo datos en la posición , y saliendo del for imprime los datos que se encuentran dentro del arreglo y ahora en **cad**.

```

public String imprimeDatos()
{
    String cad="";

    for (int i = 0; i<=p; i++)
    {
        cad+= "[" + datos[i] + "]" + "\n";
    }
    return "\n" + cad;
}

```

El siguiente método nos ayuda a crear una búsqueda Secuencial , con el objeto dat y se crea la variable byte i con 0 ,para entrar a un while donde y mientras i sea menor a lo que hay en p y i sea diferente a lo que hay dentro del arreglo en la posición de i, e incrementara en i y retorna el objeto y dentro de este el arreglo en la posición i devolviendo un -2

```

public byte busquedaSecuencial(Object val)
{
    byte i=0;
    while(i<=p && !val.equals(datos[i]))
        i++;
    return(i<=p)? i:-1;
}

```

Para eliminar datos dentro del arreglo utilizaremos el siguiente método con la variable de tipo byte pos, y j inicializada con 0 y entra al ciclo for donde si j es igual a la posición del número el cual queremos eliminar entonces j será menor a lo que hay en p y j incrementa y mandando a llamar al método le agregamos lo que tenemos el j que es el nuevo dato, y p decrementara

```

public void eliminaDatos(int pos)
{
    for (int j=pos; j<= p; j++)
    {
        datos[j]= datos[j+1];
    }

    p--;
}

```

El último método nos ayudara a modificar un dato dentro del arreglo ,con la variable de tipo byte dat agregando un nuevo valor dentro del arreglo datos.

```

public void modificarDato(byte pos) {

    int val=0;
    val = ToolsPanel.leerInt("Ingrese el nuevo dato");
    for (int j = pos; j<= pos; j++){
        datos[j] = val;
    }
}

```

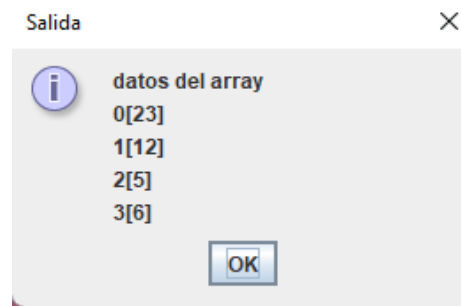
CLASE MENÚ DESORDENADO

Para el menú se mandan a llamar los métodos con los mensajes para agregar los métodos y posteriormente los imprime:

```
case "AgregarDesordenado":  
    obj.almacenarDatos();  
    break;  
  
case "Imprimir":  
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");  
    else ToolsPanel.imprime("Datos del arreglo\n"+obj.imprimeDatos());  
  
    break;
```

Activar Windows

EJECUTABLE DE AMBOS METODOS (ESTE SIRVE PARA VISUALIZAR LOS DATOS QUE INGRESAMOS)



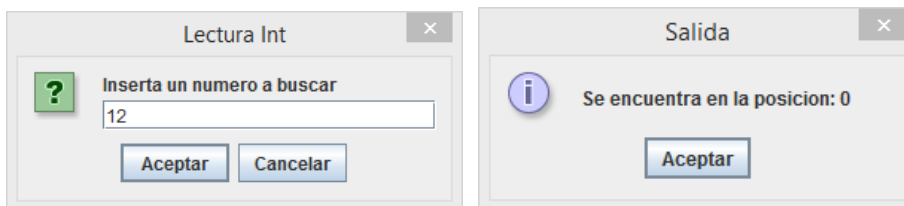
A excepción de buscar donde primero tenemos que verificar si el arreglo de encuentra vacío o lleno y si lo está nos mandará un mensaje de que el arreglo se encuentra vacío, y si no primero mandaremos a preguntar qué número queremos buscar a través de la variable pos que nos indica dónde se encuentra el dato y si pos es diferente a -2 nos mandará en que posición se encuentra el dato , y si no nos mandará el mensaje de que no se encuentra el dato dentro del arreglo.

```

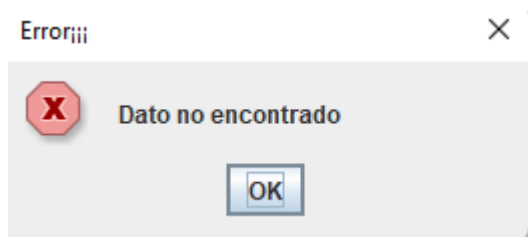
case "BuscarDesordenado":
    if(obj.validaVacio()) {
        ToolsPanel.imprimeErrorMsje("Array vacio");
    }
    else {
        pos = obj.búsquedaSecuencial(ToolsPanel.leerInt("Inserta un numero a buscar"));
        if(pos>=0) {
            ToolsPanel.imprime("Se encuentra en la posicion: " +pos);
        }else {
            ToolsPanel.imprimeErrorMsje("Dato no encontrado");
        }
    }
    break;

```

Aquí agregamos el número que deseamos buscar en este caso el número 12 que si se encuentra dentro del arreglo



y si ingresamos un número que no se encuentra dentro del arreglo como el 3 nos mandara este mensaje



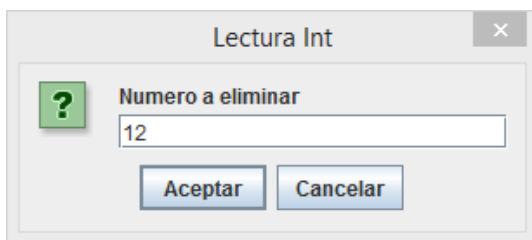
El siguiente que es la excepción es el método de Eliminar donde al igual que buscar nos valida si el arreglo esta vacío y si no lo está ocupando el método de búsqueda para saber si el dato que queremos eliminar se encuentra dentro del arreglo ,se eliminará el dato y si no lo está nos manda el mensaje que el número no se encuentra dentro del arreglo.

```

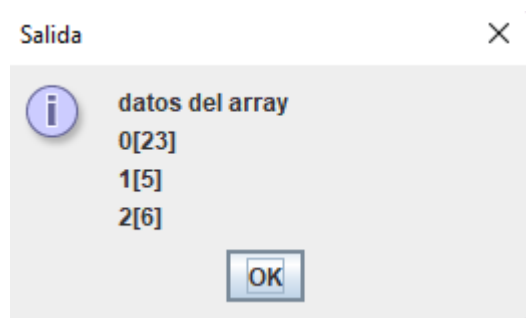
case "Eliminar":
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");
    else
    {
        pos = obj.busquedaSecuencial(ToolsPanel.leerInt("Numero a eliminar"));
        if(pos>=0)
        {
            obj.eliminaDatos(pos);
            ToolsPanel.imprime("Datos del arreglo"+obj.imprimeDatos());
        }
        else {
            ToolsPanel.imprimeErrorMsje("Dato no encontrado");
        }
    }
    break;

```

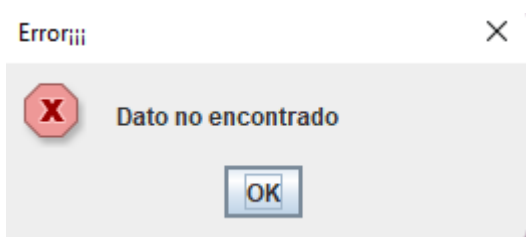
En este caso deseamos eliminar el número 12 que está dentro del arreglo



Y si el número si se encuentra dentro se eliminará



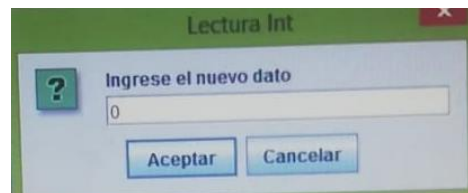
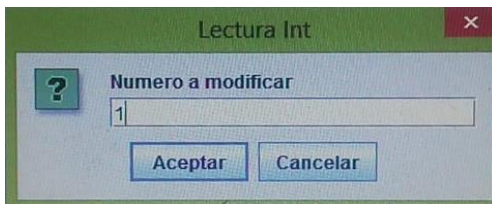
Y si el número no se encuentra dentro del arreglo nos mandara también este mensaje y no se elimina nada



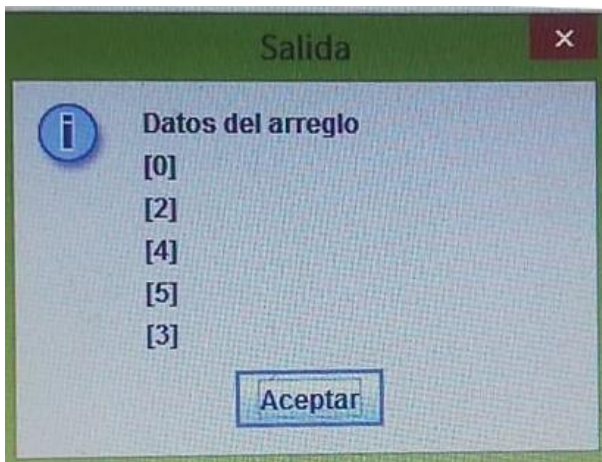
También el método de modificar ocupa lo mismo que el método de eliminar solo que en esta ocasión se reemplaza el dato eliminado por un dato nuevo y entonces el dato será modificado

```
case "Modificar":
    if(obj.validaVacio()) ToolsPanel.imprimeErrorMsje("Array vacio");
    else
    {
        pos = obj.búsquedaSecuencial(ToolsPanel.leerInt("Numero a modificar"));
        if(pos>=0)
        {
            ToolsPanel.imprime("Dato modificado en la posicion: " +pos);
            obj.modificarDato(pos);
            ToolsPanel.imprime("Datos del arreglo"+obj.imprimeDatos());
        }
        else ToolsPanel.imprimeErrorMsje("Dato encontrado");
    }
    break;
```

Para este caso en particular modificaremos otra serie de dato siendo el numero 1 Y será reemplazado por el nuevo dato en este caso 0



Dándonos así la lista ya modificada



CONCLUSION

Con lo anterior expuesto en esta práctica, se define que el tener claros los conceptos y la información a trabajar, además de generar un análisis exhaustivo de las necesidades que debe cumplir el programa y porque medio se van a cumplir, son características fundamentales previo a la creación de código, si no se tiene claro que se va a diseñar, no se puede tener claro cómo se va a realizar.

El tener un lenguaje correcto para aplicar a los programas es parte integral de estos, las desventajas de desconocer un lenguaje o no conocer lo que lo distingue de otros lenguajes es un desfavorecimiento para el programador. Por eso se usa JAVA lenguaje de programación, debido a que permite tener una compilación mucho más rápida y un uso para el usuario nada tedioso. Aunque no se genera un entorno gráfico muy cómodo para el usuario, este solo se encarga de ingresar la información para obtener el informe que necesita por lo cual en realidad no es un punto en contra el del lenguaje.

El uso de las multilistas favorece las labores de programación, con estas se agiliza profundamente el tener que hacer validaciones para el ingreso de datos y demás líneas de código que solo generan acumulación, además de hacer que el programa sea compilado mucho más lento.

BIBLIOGRAFIA

- Introducción a las Estructura de Datos - UBO - Estructuras de Datos. (n.d.). Google.com. Retrieved April 13, 2023, from <https://sites.google.com/site/edatosubo/1-introduccion-a-las-estructura-de-datos>
- No title. (n.d.-a). Zyrosite.com. Retrieved April 13, 2023, from <https://estructuradedatosbalv.zyrosite.com/about-me-copy-copy-7kZ-zPol-F8>
- No title. (n.d.-b). Zyrosite.com. Retrieved April 13, 2023, from <https://estructuradedatosbalv.zyrosite.com/1-4-2-memoria-dinamica-copy-tM8l0gv1Uz>

- No title. (n.d.-c). Zyrosite.com. Retrieved April 13, 2023, from <https://estructuradedatosbalv.zyrosite.com/about-me-copy-copy-ehjecMOMuA>
- Tipos de datos abstractos. (n.d.). Uchile.Cl. Retrieved April 13, 2023, from <https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/>
- (N.d.). Uoc.edu. Retrieved April 13, 2023, from <https://openaccess.uoc.edu/bitstream/10609/6922/6/Estructuras%20de%20datos%20b%C3%A1sicas.pdf>