ESTRUCTURA DE DATOS

ABSTRACCIÓN: permite tratar la complejidad, entendida como exceso en el número de detalles.
Se trabaja de menor a mayor nivel de detalle (de complejidad)

ESPECIFICACIÓN
valores
⊕
operaciones
→ IMPLEMENTACIÓN

"interfaz" pública
Representación privada "encapsulamiento" ⊕ "ocultación"

Tipos abstractos de datos y programación orientada a objetos:

(T.A.D) = MODELO DE DATOS + OPERACIONES
Especificación TAD's

(OBJETO) = ESTRUCTURA DE DATOS + METODOS
Implementación TAD's

Especificación del TAD: { Especificación formal (lenguaje algebr)
{ Pseudocódigo (puede no ser necesaria)

Implementación del TAD: { Definir los tipos de datos
{ Especificar el lenguaje algorítmico

NOTACIÓN ASINTÓTICA O: es una función del tamaño de la entrada de datos (n) utilizada para hacer referencia al tiempo máximo de ejecución T(n) de un programa para una entrada de tamaño (n).
↳ Es muy importante la velocidad de crecimiento de la función, y la influencia del tamaño de las entradas en el tiempo de ejecución.
RESUMEN: Te dice como de bueno o malo es un algoritmo a medida que crece en tamaño.
CONTRATO: especificación formal que define las reglas y expectativas entre diferentes partes de un sistema (como clases, módulos o servicios). Establece qué funciones o métodos deben estar disponibles, qué tipo de datos deben aceptar y devolver y qué condiciones deben cumplirse.
ÁMBITO: se refiere a la región del código donde una variable, función u objeto es accesible. Define qué partes del código pueden ver y usar ciertas variables.
INTERFAZ: define un conjunto de métodos que una clase debe implementar.

ESTRUCTURAS LINEALES

Listas (colección ordenada de elementos) → Listas simplemente enlazadas
→ Listas doblemente enlazadas
→ Listas básicas

↳ Las listas tienen orden, pero no están ordenadas.
Puedo acceder a los elementos con un determinado num.

En un conjunto no hay orden.

LISTAS BÁSICAS

En las listas puedo:
- Añadir elementos
- Eliminar elementos
- Recoger elementos → Get()
- Insertar elementos → Set(posición, valor)

Tipos de colas
- FIFO
- LIFO

Migración: Proceso de mover o transformar datos, código (...) de un estado a otro. Se trata de modificar la estructura de una base de datos (agregar / eliminar / modificar tablas...) o mover datos de un sistema a otro.

ARRAY: Estructura de datos que almacena múltiples elementos del mismo tipo en una secuencia ordenada.

RAM: (Memoria de acceso aleatorio) es un tipo de memoria volátil que almacena datos temporales. (Se borra cuando se apaga el equipo).

(*) Set(pos, valor): nosotros al definir esta operación no hemos contado con que el valor pueda ser un gato. Si esto quería así no nos vale ~~(esta)~~ (est) para esta estructura de datos por lo que tendríamos que crear todo desde el inicio.

↳ Para que esto no ocurra establecemos una plantilla para poder operar con todo tipo de datos.

Definimos nuestro contrato por clases:

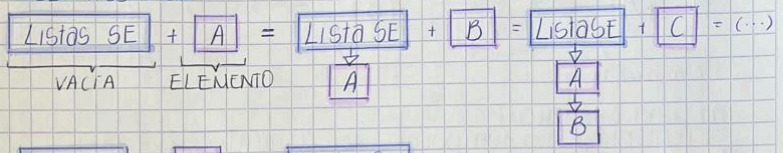
CLASES

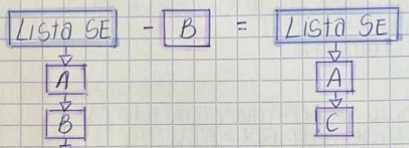
LISTA BÁSICA	
Array datos Nº de elementos Tam_máximo	ÁREA DE DATOS ATRIBUTOS (variables)
Add Insert Delete Set GetIndex Push GetValue Pop	FUNCIONES MÉTODOS

LISTAS SIMPLEMENTE ENLAZADAS: Estructura de datos donde cada elemento almacena un valor y una referencia (puntero) al siguiente elemento de la lista. A diferencia de un array, los elementos no están almacenados en posiciones contiguas en la memoria, sino que están enlazados entre sí mediante punteros. (Los elementos no están numerados).

LISTAS DINÁMICAS

IDEA ABSTRACTA:

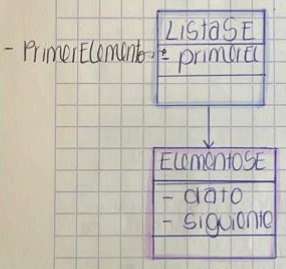


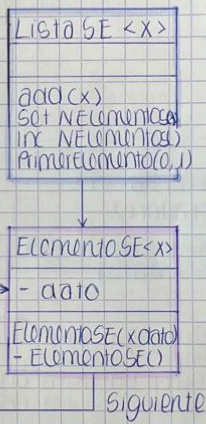


El elemento A es un elemento (<TipoDato>) y trabajo con él sin importar qué tipo de dato sea.

Cuando A se suma, pasa a estar dentro de un elemento de la lista, y cuando se añade B, al elemento que contiene "A" engancha el elemento que contiene "B"

UML:






```
Class Lista <TipoDato>
    tipoDato
{
datosE[*] = New tipoDato
                        [?]
```

Relación reflexiva

Debemos hacer un comando para definir cómo llamar a la lista, cómo añadir, cómo quitar...

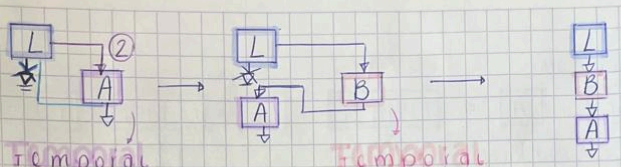
```
ListaSE <int> miLista = newListaSE < int>();
...
void miPrueba(){
    int a = new int(5);
    miLista.add(a);
}
```

this.primerElemento

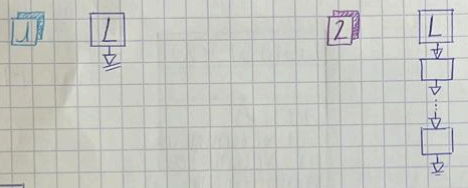
Código de la clase ElementoSE:

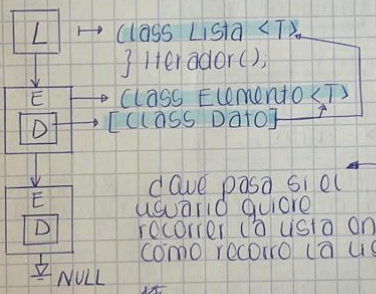
```
Class ElementoSE <X>{
    X dato;
    ElementoSE <X> siguiente;
    ElementoSE (x miDato){
        this.dato = miDato;
    }
    private ElementoSE(){}
}
```

```
Class ElementoSE <X> {
    int add(dato)
    ElementoSE<X> temporal =
        = new ElementoSE<X>(dato);
① Temporal.setSiguiente(*);
② set PrimerElemento(temporal);
    inc NElementos();
```




```
int add (x dato){
  ElementoSE<x> temporal = newElementoSE<x> dato;
① if (getPrimerElemento() == null){
    set PrimerElemento (temporal);
  }
② else {
    ElementoSE<x> aux = getPrimerElemento()
    while (aux.getSiguiente() <> null{
      aux = aux.getSiguiente();
    }
    aux.setSiguiente (temporal);
}
```





¿Cómo somos capaces de recuperar los elementos de una lista?
Voy a permitir al usuario recorrer la lista, que me vaya pidiendo elementos y yo se los voy dando.

¿Qué pasa si el usuario quiere recorrer la lista en paralelo? Como recorro la lista?: con un ITERADOR

ITERADOR: es un objeto que permite recorrer una secuencia de elementos (como listas, arrays...) uno a uno sin necesidad de conocer su estructura interna.

- ¿Qué tiene un iterador?:
  - un constructor
  - un "haySiguiente()" ← Metodo
  - un "getDato()"
- Los iteradores van a tener una interfaz:

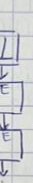
Pegine los metodos

```
interfaz Iterador<T>{
    haySiguiente();
    getDato();
}
```

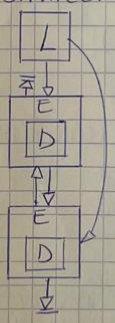
(interfaz)

```
class Iterador implements IIterador <T> {
  Iterador (Lista <T> l){
      Lista <T> miLista;
      Elemento <T> actual;
      Iterador (Lista <T> l){ → ESTADO
        this.miLista = l;
        this.actual = l.primerElemento;
      }
  }
  haySiguiente (){
      boolean haySiguiente (){
        return this.actual = null;
      }
  }
  getDato (){
      T getDato (){
        T temporal = this.actual.getDato();
        this.actual = this.actual.getSiguiente();
        return temporal;
      }
  }

  static void main(){
    Lista <String> l = new Lista <String> ();
    l.add ("Hola")
    l.add ("Mundo")
    IIterador <String> i = l.iterador();
    while (i.haySiguiente()){
      System.out.println(i.getDato());
    }
}
```



LISTAS DOBLEMENTE ENLAZADAS: es una estructura de datos similar a la simplemente enlazada solo que cada nodo tiene dos punteros:
- uno apunta al siguiente nodo
- otro apunta al anterior nodo

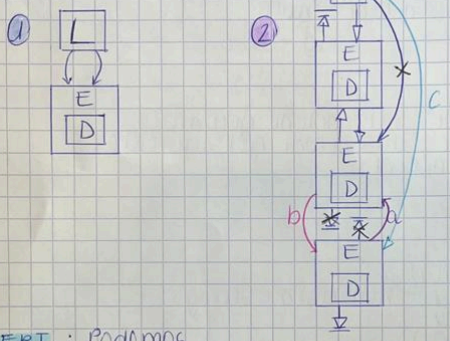



```
class ListaDE {
  ElementoDE primero
  ElementoDE ultimo
}
class ElementoDE
  T Dato
  ElementoDE anterior,
  ElementoDE siguiente,
}
```

ADD

```
void add (TD dato){
    ElementoDE<TD> elemento = new ElementoDE (dato);
    ig(this.primero == null){
        this.primero = elemento;
        this.ultimo = elemento;
    }
    else{
        a elemento.anterior = this.ultimo;
        b this.ultimo.siguiente = elemento;
        c this.ultimo = elemento;
    }
}
```

① LISTA VACIA
② LISTA CON ELEMENTOS



- INSERT : Podemos definirlo tanto en el iterador como en la lista:

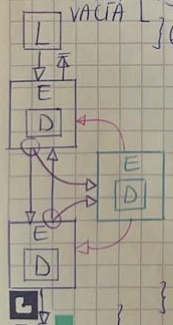
ITERADOR: insert(dato) → (Si hago esto el insert estará protegido El usuario no lo podrá ver)

```
void insert (TD dato){
    this.lista.insert(this.actual, dato);
}
```

LISTA

```
void insert(ElementoDE elemento, TD dato){
    ig(elemento == null){
        this.add(dato)
    }else{
        ElementoDE<TD> nuevoElemento = new ElementoDE(dato);
        ig(elemento == this.primero){
            elemento.anterior = nuevoElemento;
            nuevoElemento.siguiente = elemento;
            this.primero = nuevoElemento;
        }else{
            nuevoElemento.siguiente = elemento;
            nuevoElemento.anterior = elemento.anterior;
            nuevoElemento.anterior.siguiente = nuevoElemento;
            nuevoElemento.siguiente.anterior = nuevoElemento;
        }
    }
}
```

LISTA VACIA



- DELETE

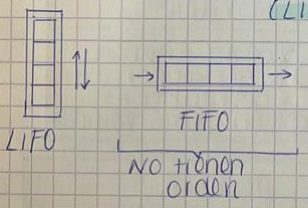
ITERADOR

```
void insert(TD dato){
    this.lista.insert(this.actual, dato);
}
void delete(){
    this.lista.delete(this.actual);
}
```

LISTA

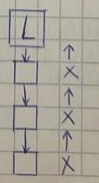
```
void delete(ELEMENTODE elemento){
    if(elemento != null){
        boolean primero = false, ultimo = false;
    }
    if(elemento.anterior == null){
        this.primero = elemento.siguiente;
        primero = True;
    }
    if(elemento.siguiente == null){
        this.ultimo = elemento.anterior;
        ultimo = True;
    }
    if(!primero){
        elemento.anterior.siguiente = elemento.siguiente;
    }
    if(!ultimo){
        elemento.siguiente.anterior = elemento.anterior;
} }
}
```

PILAS Y COLAS



(LIFO) PILAS: estructura de datos en la que el último elemento en entrar es el primero en salir (Last In, First Out).
Ej: Pila de platos
(Último elemento agregado primer en ser eliminado)
Operaciones principales:

push(el): Agrega al tope de la pila
pop(): elimina y devuelve " "
peek(): muestra el el " " " "



COLA(FIFO): estructura de datos donde el primer elemento en entrar es el primero en salir (First In, First Out).
Ej: Fila en el banco
Operaciones principales:

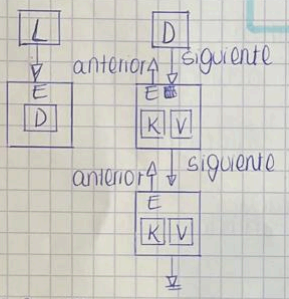
enqueue(el): Agrega un el al final de la cola
dequeue(): Elimina y devuelve el elemento en el frente de la cola

PILA:

```
class PilaLigo <T>{
    ListaDE<T> pila = new ListaDE();
    void push(T dato){ Lista.add(dato)}
    T pop(){
        T temporal = pila.ultimo.getDato();
        pila.delete(pila.ultimo);
    }
}
```

DICCIONARIOS:

<Key, Value>
<K, V>



DICCIONARIO: estructura de datos que almacena elementos en pares clave-valor. Permite acceder a valores a través de sus claves en lugar de utilizar un índice numérico. Sirve para búsquedas rápidas.

Para asignar las clases hay que tener en cuenta que ahora hay dos tipos de datos:

Diccionario<K, V>
ElementoD<K, V> Primero ElementoD<K, V> ultimo
boolean add(K,V) boolean insert(K,V) boolean delete(K,V) Lista<K> getKeys() Lista<V> getValues() boolean exists(K) V getValue(K) IteradorD getIteradorD() boolean setValue(K,V) # IteradorD Final(K) # HoradorD Final(IteradorD, K)

ElementoD<K, V>
ElementoD<K,V> siguiente ElementoD<K,V> anterior K indice V dato
boolean delete() K getKey() V getValue() boolean update Value(V)

IteradorD
Diccionario<K,V> nuDiccionario ElementoD<K,V> actual
boolean hasNext V next() V getKey() V getValue() # getActual()

Se programan como interfaces. Lo plantea sin orden.

COSAS NUEVAS: CEL LISTA ES igual a una lista doblemente enlazada)

```
Lista<K> getKeys(){
    Iterador<K,V> it = this.getIterador();
    Lista<K> claves = newLista<K>();
    while(it.hasNext()){
        it.next();
        claves.add(it.getKey());
    }
    return claves;
}
}
boolean existe(K clave){
    Iterador<K,V> it = this.getIteradorD();
    while(it.hasNext()){
        it.next();
        if(it.getKey() == clase){
            return true
        }
    }
} return false
```