ESTRUCTURAS DINÁMICAS UNIDIMENSIONALES

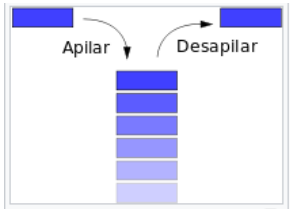
Una forma, entre muchas, de clasificar las estructuras dinámicas, es en unidimensionales y multidimensionales similarmente a como hicimos con los arrays:

- unidimensional, cada nodo está enlazado con un único nodo. Ejemplos: listas, conjuntos, pilas, colas, …
- multidimensional cada nodo puede estar enlazada con muchos nodos. Ejemplos: tablas hash(diccionarios python/mapas Java), árboles, grafos, …

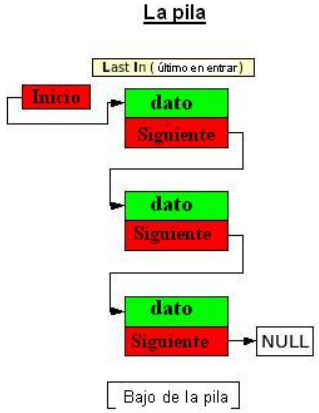
La lista es la estructura unidimensional más importante y ya le echamos un vistazo en el boletín anterior. En este boletín veremos pilas y colas "caseras" que son más que listas especializadas. Las tablas hash "caseras" son más complejas y no las abordaremos. En el próximo boletín veremos l"árboles"

UN TIPO ESPECIALIZADO DE LISTA: LA PILA.

La pila es un tipo de lista a la que se le aplica dos operaciones básicas: push (para insertar o apilar un elemento) y pop (para extraer desapilar un elemento). Su característica fundamental es que se mete y se saca por la cabeza de la pila con lo que se consigue el comportamiento LIFO (Last in First Out) como en "una pila de platos".



La pila de nuevo se puede implementar como un array o como una lista enlazada. Como una lista enlazada podemos visualizarla como sigue:



Ejercicio U6_B2B_E1: Implementa el siguiente interface Pila en una clase MiPila. Para solucionar el problema reutiliza la clase Nodo de la Lista enlazada

```
interface Pila{
    //inserta un elemento en la cabeza de la pila
    void push(int dato);

    //saca un elemento de la cabeza de la pila.
    int pop();
   public boolean esVacia() ;
}
```

Recuerda la clase Nodo

```
class Nodo {
    private Nodo sig;
    private int dato;


    public Nodo(int dato, Nodo sig) {
        this.dato = dato;
        this.sig = sig;
    }
```

```
    public void setSiguiente(Nodo sig) {
        this.sig = sig;
    }

    public Nodo getSiguiente() {
        return sig;
    }

    public int getDato() {
        return dato;
    }
}
```

y que funcione correctamente el siguiente main()

```
class App {
    public static void main(String[] args) {
        Pila mipila = new MiPila();
        mipila.push(1);
        mipila.push(2);
        mipila.push(3);
        mipila.push(4);
        mipila.push(5);
        while (!mipila.esVacia()) {
            System.out.println(mipila.pop());
        }
    }
}
```

Ejercicio U6_B2B_E2: Utilizando el código del ejercicio anterior, pero haciendo la pila de char, escribir un método que dada una expresión compruebe si los paréntesis están balanceados, observa que no es suficiente con contar que hay el mismo número de paréntesis '(' que ')'. Por ejemplo")4(" no es balanceada

```
class App {
    public static boolean parentesisBalanceados(String expresion){
      //utiliza una pila para resolver el problema
    }
    public static void main(String[] args) {
        String expresion="((2+3)/(3*(8-2))";
        System.out.println(parentesisBalanceados(expresion));
        expresion=")4(";
        System.out.println(parentesisBalanceados(expresion));
        expresion="(4)";
        System.out.println(parentesisBalanceados(expresion));
        expresion="(2+3)/(3*(8-2))";
        System.out.println(parentesisBalanceados(expresion));
    }
}
```

```
run:
false
false
true
true
```

Una forma fácil de solucionar el problema es que el método parentesisBalanceados() utilice una pila. Para ello vamos procesando cada caracter del String por orden, de izquierda a derecha. "Procesar" en este caso es leer el caracter del string y si detectamos que es un '(' hacemos un `push('(')` y si es un ')' un `pop()`.
¿Cómo se detecta que la expresión no está balanceada? Hay dos síntomas:

- En la mitad del proceso hay que hacer un pop pero la pila queda vacía.
- Se llega al final del String de forma que se procesa el último carácter y una vez procesado(puede implicar un push o pop) la pila no queda vacía lo que indica que hay más '(' que ')' y tampoco hay equilibrio.

UN ACEPTA RETO CON PILA: SUMA DE DÍGITOS

Muchos problemas de acepto el reto, aparecen en varias categorías, por ejemplo *suma de dígitos* aparece en recursividad y en estructuras de datos/pilas. Este ejercicio ya lo resolvimos y recuerda que aunque parece un problema aritmético realmente la forma más simple de resolverlo es basándose en Strings, por ejemplo

```
import java.util.Scanner;

public class App{


    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String numero="";
        while(!(numero=sc.next()).startsWith("-")){
            int suma=0;
            String resultado="";
            for(int i=0;i<numero.length();i++){
                resultado+=(i==numero.length()-1)?numero.charAt(i):numero.charAt(i)+" + ";
                suma=suma+Integer.parseInt(numero.charAt(i)+"");
            }
            resultado=resultado+" = "+suma;
            System.out.println(resultado);
        }
    }
}
```

Si me empeño en extraer los dígitos con aritmética (con operador / y %) la solución se complica un poco. Esta complicación se suaviza un poco si utilizo una pila.

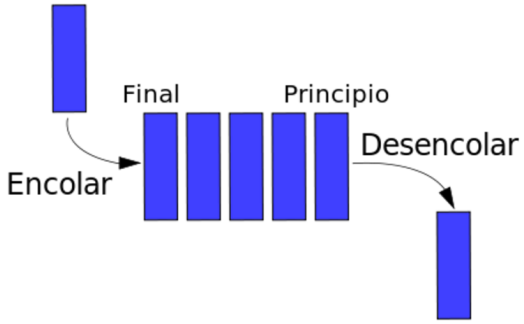
Ejercicio U6_B2B_E3: Conseguir aceptado en el problema de suma de dígitos pero utilizando OBLIGATORIAMENTE aritmética decimal para extraer los dígitos y una pila. Podemos acceder a las cifras de un número e ir introduciendo cifra a cifra en una pila de enteros básicamente con la siguiente idea

```
if (numero == 0) {
    pila.push(0);
} else {
    while (numero > 0) {
        pila.push(numero % 10);
        numero = numero / 10;
    }
}
```

Ahora simplemente quedaría ir vaciando la pila con pop pero ya vamos obteniendo los dígitos en el orden deseado para imprimir

OTRO TIPO ESPECIALIZADO DE LISTA: LA COLA.

Una cola es una Lista en la que sus elementos se introducen (Encolan) únicamente por un extremo que le llamamos "Final de la Cola" y se quitan (Desencolan) únicamente por el extremo contrario al que le llamamos "Frente de la Cola" o "Principio de la Cola". También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.



Ejercicio U6_B2B_E4: Implementa el interface Cola en la clase MiCola

Observa que ahora la cola maneja dos referencias: primero y último. Por definición de cola, *principio* se refiere al extremo por donde salen y *final* por donde entran

```
interface Cola {

   //inserta un elemento al final de la cola
   void encolar(int dato);

   //saca el primer elemento de la cola
   //el primer elemento es el más antiguo
   int desencolar();

   public boolean esVacia();
}

class MiCola implements Cola{
   Nodo primero=null;
   Nodo ultimo=null;
   etc...
}
```

pruébala con el siguiente main()

```
class App {

   public static void main(String[] args) {
      MiCola mc = new MiCola();
      mc.encolar(1);
      mc.encolar(2);
      mc.encolar(3);
      while (!mc.esVacia()) {
         System.out.println(mc.desencolar());
      }
   }
}

run:
1
2
3
```

ESTRUCTURAS DINÁMICAS Y RECURSIVIDAD

En muchas situaciones, utilizar recursividad para resolver algoritmos de estructuras de datos es muy apropiado y ventajoso. Veremos por ejemplo que árboles y recursividad van "cogidos de la mano". Antes de ver árboles, para recordar el uso de recursividad haremos un ejercicio con Listas donde la recursividad no es tan importante como en los árboles pero es más fácil de entender y por tanto más didáctico.
Haremos una serie de métodos que imprimen con println(), en la práctica real una estructura dinámica de carácter general como una lista no debería incluir en su código sentencias println(), pero relajamos esto por fines didácticos. También relajaremos esto mismo y por la misma razón cuando veamos árboles.

Ejercicio U6_B2B_E5: Recuerda el ejemplo MiListaEnlazada. Se pide que ahora el recorrido e impresión de la lista lo soporte la propia clase. Queremos recorrer la lista de 4 formas y debemos escribir por tanto 4 métodos:

- imprimirConFor()
- imprimirInvertidaConFor()
- imprimirConRec()
- imprimirInvertidaConRec()

y los probamos desde main()

```
class App {

   public static void main(String[] args) {
      MiListaEnlazada miLista = new MiListaEnlazada();
```

```
        miLista.insertar(8);
        miLista.insertar(88);
        miLista.insertar(888);
        miLista.imprimirConFor();
        miLista.imprimirInvertidaConFor();
        miLista.imprimirConRec();
        miLista.imprimirInvertidaConRec();
    }
}
```

que provoca la siguiente salida

```
run:
imprimirConFor .....
888 88 8
imprimirInvertidaConFor .....
8 88 888
imprimirConRec.....
888 88 8
imprimirInvertidaConRec.....
8 88 888
```