Tema 1: Breve introducción a Java

- Java es un lenguaje de programación orientado a objetos.
- Se rige con la filosofía “Write one, run anywhere” (Es portable)
- Al compilar el código no se genera lenguaje máquina, sino un código intermedio conocido como Bytecode.
- La única máquina capaz de ejecutar Bytecode es la “Java Virtual Machine” (JVM).
- La JVM es especificada por Oracle y es implementada por varias empresas. Osea, existen distintas “marcas” de JVM (Oracle, IBM, …)
- Cada proceso Java se ejecuta en una JVM aparte.
- Java + JVM + Utilidades para programar, conforman una plataforma de software para el desarrollo de aplicaciones de usuario.

Existen varias plataformas en Java:

- SE (Standart Edition) Aplicaciones de escritorio.
- ME (Mobile Edition) Dispositivos móviles.
- EE (Enterprise Edition) Aplicaciones Servidor.

Java se distribuye de varias formas:

- JRE (Mínimo para ejecutar programas)
- JDK (JRE + SE)
- Servidores de aplicaciones (Implementaciones EE)

Paradigmas de programación

- En general, es una forma de ver/entender/modelar el mundo.
- En el ámbito del software, es un estilo fundamental de programación. Determina cómo el programa “ve” el mundo.
- Determina cómo debe ser usado el lenguaje por el programador.
- Altos lenguajes son multiparadigmas (no se hace una selección explicita del código).

Hay cuatro paradigmas principales:

Funcional
Lógico
Imperativo
Orientado a Objetos

Paradigma Lógico:

- El mundo se modela mediante predicados lógicos.
- Se aplican directamente principios matemáticos.
- Poco utilizado en aplicaciones comerciales.
- Prolog es el principal lenguaje.

Paradigma Funcional:

- El mundo se modela como funciones matemáticas.
- Lenguajes 100% funcionales (Lisp, Scheme).
- Permite declarar funciones, pasar funciones, pasar parámetros, retornar funciones, entre otros.
- Muchos lenguajes modernos han incorporado este paradigma por conveniencia.

Paradigma Imperativo:

- El mundo se modela como instrucciones, pasos, procedimientos.
- Ampliamente utilizados en aplicaciones comerciales.
- Tienden a ser muy eficientes (generalmente compilan en lenguaje máquina).
- Gramaticalmente simples.
- Se consideran como de nivel de abstracción bajos.
- Ejemplos: C/C++

Paradigma Orientado a Objetos:

- El mundo se modela como objetos del mundo real junto con su interacción.
- Muy naturales para los humanos.
- Facilitan la reutilización de código (Herencia).
- Mayoritariamente no generan lenguaje máquina.
- Poseen muchas estructuras sintácticas
- Ejemplos: Java, C++, Python, JavaScript.

Lenguaje Java

Colección: Matrices (puede contener datos primitivos "int, double, char" u objetos "Integer, Double, String", siempre y cuando sean del mismo tipo y Listas acepta Objects (Cualquier tipo de elemento, no hay que definir el tamaño de la mismo).

Conjunto: Es una construcción de colecciones que por definición contiene elementos únicos, es decir ninguno duplicado. Un conjunto se preocupa por la singularidad, no por el orden de los datos como en el caso de las colecciones tipo matriz(array) y tipo List.

Excepciones:

- **Tiempo de compilación:** Errores de sintaxis (olvidar poner punto y como, cerrar llaves, etc)
- **Tiempo de Ejecución:** Heredan de la clase Throwable
  - **Error:** Muy relacionado con problemas de hardware (escases de memoria, desbordamiento de la pila del sistema, etc. Es muy poco normal que ocurran.
  - **Exception:**
    - **IOException:** Clase que contiene excepciones verificadas (que un archivo no se encuentre, que se quiera leer y este vacié, entre otros). Requieren de try-catch
    - **RuntimeExcepcion:** Clases que contiene excepciones no verificadas (Dividir entre cero, agregar más elementos a una matriz con magnitud definida, almacenar un dato en un tipo de dato incorrecto, etc).

Nota: Hay clases que exigen el lanzamiento de excepciones (throws), esto es implementar las instrucciones try-catch. Por ejemplo, la clase ImageIO exige captar una posible excepción con try-catch, en todo caso dicha excepción seria verificada ya que pertenece a la excepción IOException.

Nota: Las excepciones de tipo o que heredan de RuntimeExcepcion no exigen captar la excepción en un try-catch, es decir como programador te permite considerar código adicional que evite que tal excepción ocurra y de ser así hacer algo al respecto.

throws y throw: throws se utiliza en la declaración de un método para indicar que dicho método podría lanzar una excepción, mientras que la instrucción throw se utiliza en cualquier parte del código para indicar que en tal parte en concreto se lanza una excepción.

¿Como crear excepciones?
Se debe crear una clase que herede de la clase Exception o cualquier otra subclase de excepciones (RuntimeException, IOException, etc). La clase creada debe tener mínimo 2 métodos constructores, uno sin argumentos y el otro con un argumento tipo String para informar la causa de la excepcion. Hay que considerar que, si la clase hereda de la clase Exception o IOException, al llamar el método que podrá usar la excepción creada, el mismo deberá estar dentro de un try – catch, caso contrario si la clase hereda de la clase RuntimeException.

Lectura y Escritura de archivos

FileWriter, BufferedWriter, PrintWriter: Se usan normalmente para escribir datos orientados a caracteres en un archivo. A diferencia de la clase FileOutputStream, no necesita convertir la cadena en una matriz de bytes porque proporciona un método para escribir la cadena directamente.

FileOutputStream: Convierte un String en bytes y escribe los bytes en un archivo.

RandomAccessFile: Permite escribir en cualquier parte del archivo (parecido a Python).

FileChannel: Para escribir archivos muy grandes.

UML: Lenguaje de modelo unificado. Conjunto de diagramas. Lectura y ponerlo en la bitácora.

Programación Orientada a Objetos

Paquetes: Mecanismo para encapsular clases, sub paquetes e interfaces. Son usados para:

- Prevención de conflictos de nombres, por ejemplo, pueden haber clases con el mismo nombre siempre y cuando estén en diferentes paquetes.
- Permite tener un mayor orden de las clases de acuerdo a sus funcionalidades, por ejemplo, las clases del paquete javafx, son específicas para el desarrollo de aplicaciones de escritorio y las clases del paquete java Lang (paquete por defecto de Java) como la clase Math permiten realizar operaciones lógicas y aritméticas.

Clase: Modelo donde se redactan las características comunes de un grupo de objetos.

Objeto: Es un ejemplar o instancia de una clase, tiene propiedades(atributos) y comportamientos(métodos).

Encapsulamiento: Principio en el cual las propiedades y comportamientos de un objeto se mantienen o no al margen de otros objetos.

Herencia: Mecanismo de POO que permite la reutilización de código. Ejemplo: Considere que se desea crear un objeto Empleado, si antes se creo un objeto Persona, el objeto Empleado perfectamente puede heredar todos los atributos y comportamientos del objeto Persona ya que el objeto Empleado es

una persona. A partir de ahí el nuevo objeto puede tener sus propios métodos y comportamientos como tener un salario, un ID, un jefe… cosas que un objeto Persona no tiene. Todas las clases heredan de la clase Object.

Polimorfismo: Principio de sustitución. Se puede utilizar un objeto de la subclase siempre que el programa espere un objeto de la superclase. Ejemplo si se desean conocer todas las características básicas de una persona (nombre, edad, país, fecha de nacimiento, altura, peso…) puedo llamar a dicho método Getter que lo realiza, sin embargo, en ese llamado perfectamente se puede indicar que en un lugar de dar las características básica de una persona, de las características de un empleado pues un empleado también es una persona.

Enlazado dinamico: Forma en la que Java distingue objetos que presentan o no herencia, relacionado con el polimorfismo.

Clase Abstracta: Clase que marca el diseno en la jerarquia de herencia. Normalmente es la clase mas generica de un programa con multiple herencia. Cuando se define un metodo abstracto (metodo que heredaran todas las subclases de esta superclase) la clase se convierte en una clase abstracta.

- Con que exista al menos un método abstracto, la clase que lo contiene debe declararse como una clase abstracta.
- Un método abstracto está obligado a definirse en cada subclase bajo la premisa de que todas las comparten, pero difieren en información. Ejemplo: imagine un método que devuelve la descripción general del objeto, un objeto persona, jefe y empleado deben heredar dicho método, sin embargo, la descripción general de cada uno de ellos varia ya que no son el mismo objeto, la información de dicho método varia.

TEMA: UML y patrones de diseño

SDLS (Software Development Life Cycle):

1. Recopilación de requerimiento
2. Análisis de requerimientos
3. Diseño de software
4. Implementación
5. Pruebas
6. Instalación y mantenimiento

Ingeniería de software: Código / Software de calidad (que funcione, sea mantenible, se ajuste al tiempo y presupuesto del proyecto, que sea reusable). La ingeniería de software esta titulada bajo SEI, ISO

Modelo de cascada – Waterfall: Crisis del software
Modelo en espriral: De este modelo nació Rational Unified Process que creo UML, otro es Agile (Screen, XP)

Diseño Orientado a Objetos

- Modelo para guiar al programador durante el proceso de diseño de software bajo el paradigma OO.
- Provee un conjunto de principios para calificar qué tan bien está el diseño de software.

Primera Fase: Modelado Conceptual

- Entender el dominio del problema y crear un modelo conceptual. Este modelo se puede hacer a criterio del diseñador
- Deber ser fácil de entender
- Storyboard
- ¿Cómo calza el software en el cuadro completo?

Segunda Fase: Análisis

- Construir historias de usuario / requerimientos
- User stories siguen un formato: “Yo como <rol> quiero ser capaz de <acción> con el fin de <objetivos>.”
- Se pueden crear prototipos para entender mejor el problema

Tercera Fase: Arquitectura

- Definir la estructura de solución para cumplir los requerimientos funcionales y de calidad
- Documentar las decisiones

Cuarta Fase: Diseño detallado

- Construir de forma iterativa un diseño detallado del sistema por construir
- Paso #1: Identifique las clases
    - Busque los sustantivos en las historias de usuario
    - Algunos sustantivos se convierten en clases. Otros se eliminan y otros se unen
    - Las clases deben tener una sola responsabilidad
- Paso #2: Identifique asociaciones
    - Identificar cómo interactúan las clases entre sí
    - Una asociación tiende a convertirse en un atributo
- Paso #3: Identifique atributos y métodos

Patrones de diseño

- Un patrón es una regularidad
- En el software hay regularidades o problemas recurrentes
- Reutilizar buenas soluciones es una buena práctica conocida por los ingenieros de software (basar nuevas soluciones en experiencias previas)

Patrón de diseño:

- Solución general para un problema común en un contexto dado
- Es una buena práctica
- Son conocidas por muchos profesionales a nivel mundial

Hay 3 categorías de patrones de diseño

- **Creacionales:** Determinan cómo crear objetos
- **Estructurales:** Cómo usar objetos entre sí mediante composición
- **De comportamiento:** Cómo comunicar objetos sin componerlos directamente

Creacionales	Estructurales	De comportamiento
Factory	Adapter	Intérprete
Builder	Bridge	Cadenas de responsabilidad
Singleton	Composite	Comando
Prototype	Decorator	Iterator
	Facade	Memento
	Proxy	Observer
	Abstract Factory	

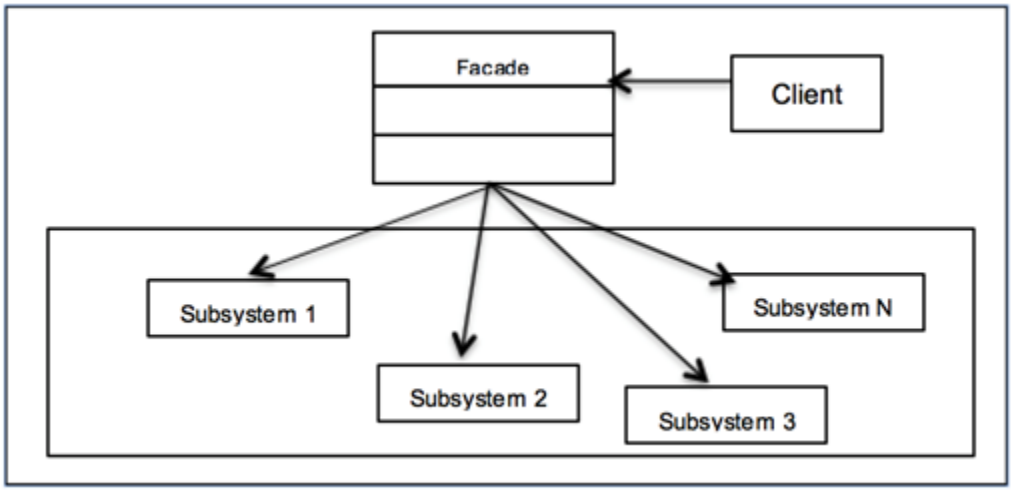
Patrón Factory: Crea una clase Factory que crea objetos concretos mediante un selector. El caller no conoce a las clases concretas.

```
interface Dog {
    void bark();
}
class Rottweiller implements Dog {
    @Override
    public void bark() {

    }
}
class Poodle implements Dog{
    @Override
    public void bark() {
```

```
    }
}
class Dog_Factory{
    public static Dog getDog(DogType d) throws Exception {
        if (d==DogType.BIG){
            return new Rottweiller();
        }
        else if (d==DogType.SMALL){
            return new Poodle();
        }
        else {
            throw new Exception("Unknown DogType");
        }
    }
}
enum DogType{
    SMALL,
    BIG
}
```

Patrón Facade: Crea una clase que funciona como fachada para un subsistema complejo. El caller interactúa con el Facade y no con el subsistema directamente.

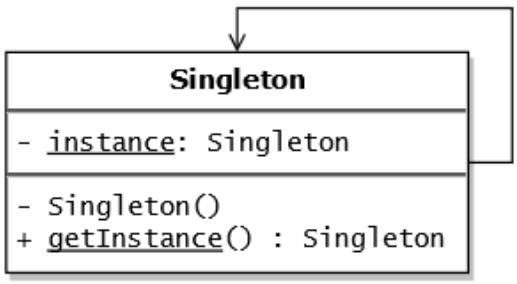


Patrón Singleton: Patrón que permite restringir la instanciación de una clase. Únicamente se puede instanciar una sola clase.

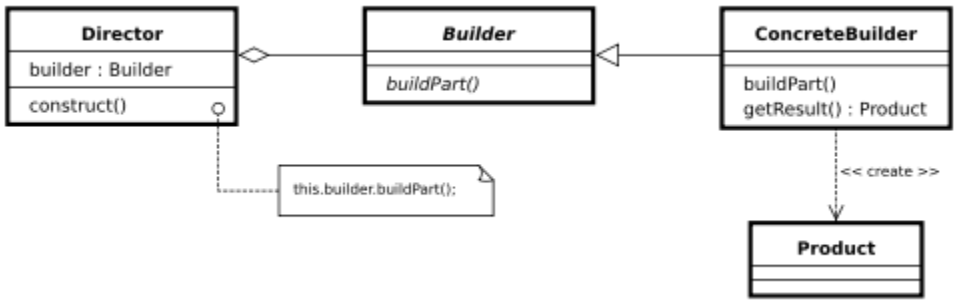
```
class Singleton {
    private static Singleton instance = null;
    private Singleton(){

    }
    public void doSomenthing(){

    }
    public static Singleton getInstance(){
        if (instance==null){
            instance = new Singleton();
        }
        return instance;
    }
}
```



Patrón Builder: Delega la construcción de una clase compleja a una clase complementaria.



TEMA: Estructuras de datos lineales

Tipos primitivos
Tipos referencia –> Extender el lenguaje

TDA: Tipo de datos abstractos

- Fundamento matemático
- Listas enlazadas (add, addFirst, addLast, remove)
- Arbóles
- Grafos
- Set / Conjuntos
- Maps / Diccionarios
- Colas y Pilas

Arrays

- Convención finita y bien definida de elementos de x tipo.
- Cada elemento está contiguo uno al otro en memoria

```
int[] a = new int[8];
a[0]=10;
a[7]=20;
System.out.println(a[3]);
int x=a[0];
```

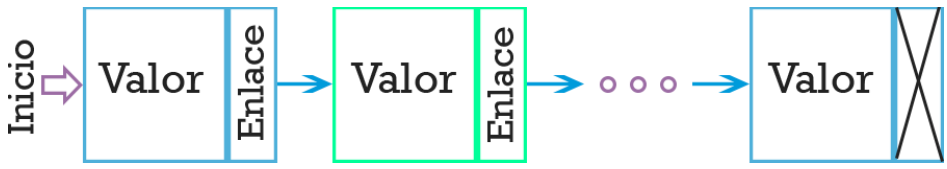
- Pros: Rápida lectura, rápido acceso;
- Contras: Eliminación/Inserción en posiciones aleatorias si no hay espacio, es inmutable.

Matrices

- Array multidimensional
- Filas y columnas: int[][] m = new int[2][2];
- No todos los elementos están contiguos.

Listas Enlazadas y Doblemente Enlazada

- Estructura de datos dinámicos
- Crece o decrece por demanda
- No está contiguo en memoria
- Lista secuencia de elementos
- La lista tiene una referencia al primer elemento, esta se protege
- El último nodo apunta a null
- Cada elemento es un nodo



- Un nodo es un objeto compuesto por dos objetos
  - El valor o dato que guarda.
  - Referencia al siguiente elemento de la lista

Pros de la lista enlazada:

- Crece y decrece por demanda
- Inserción en posiciones aleatorias es rápida

Contras de la lista enlazada:

- Lectura / búsqueda lenta

Implementación

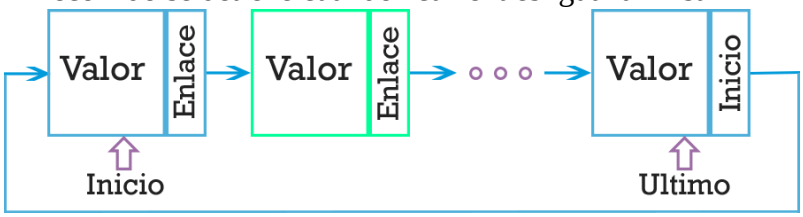
```
class Node {
    int dato;
    Node next;
    public Node(int dato){
        this.dato = dato;
        this.next = null;
    }
}

class LinkedList{
    private Node first;
    public LinkedList(){
        this.first = null;
    }
    public void addLast(int e){
        if (this.first == null){
            this.first = new Node(e);
        }
        else {
            Node current = this.first;
            while (current.next!=null){
                current= current.next;
            }
            current.next = new Node(e);
        }
    }
    public void addFirst(int e){
        if (this.first == null){
            this.first = new Node(e);
        }
        else {
            Node temp = new Node(e);
            temp.next = this.first;
            this.first = temp;
        }
    }
    public void deleteLast(){
        if (this.first==null){
            if (this.first.next == null){
                this.first = null;
            }
            else {
                Node temp = this.first;
                while (temp.next.next!=null){
```

```
                    temp = temp.next;
                }
                temp.next = null;
            }
        }
    }
}
```

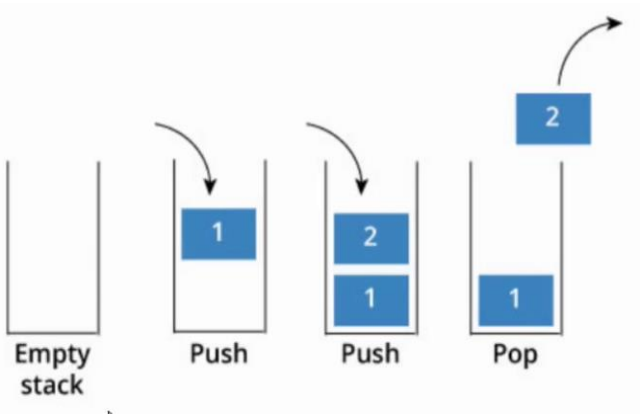
Listas Circulares
Es una lista en la que cada nodo tiene un sucesor y predecesor

- El sucesor del último, es el primer elemento
- El predecesor del primero, es el último
- Iniciando en cualquier nodo, se puede recorrer la lista completa
- El first apunta al último elemento
- Son útiles cuando se necesita acceder rápidamente al inicio / fin de la lista
- El recorrido se detiene cuando "current es igual al first"



Pilas

- Conjunto de elementos colocados uno sobre el otro
- Únicamente se puede acceder al último elemento insertado
- Siempre se inserta al final (el final se llama tope)
- Naturaleza LIFO (Last Input, First Output)



✓ Tiene 3 operaciones:
- Push: Insertar un element en el tope
- Pop: Eliminar elemento en el tope
- Peek: Ver elemento en el tope

- ✓ Acotación: Se puede implementar con listas o arrays

```
class Stack {
    private int maxsize;
    private Object[] stackarray;
    private int topPosition = -1;

    void push(Object newObject){
        if (topPosition < maxsize){
            stackarray[++topPosition] = newObject;
        }
    }

    Object pop(){
        return stackarray[topPosition--];
    }
}
```

TEMA: Algoritmos de ordenamientos

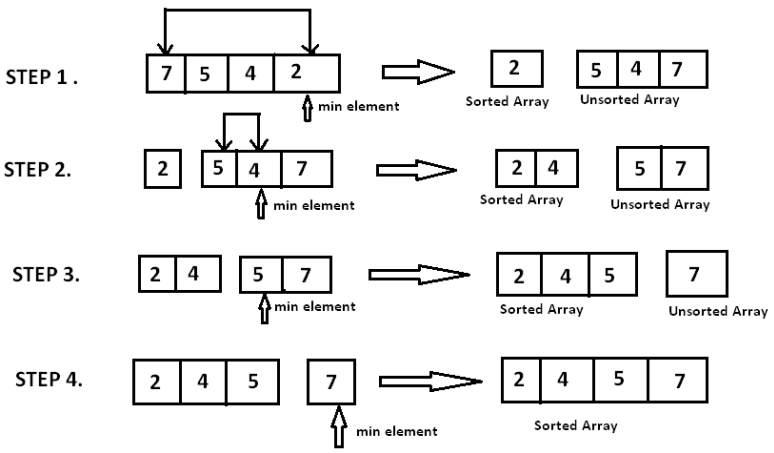
- ✓ Mantener listas de elementos ordenados es importante para facilitar búsquedas.
- ✓ Una lista de elementos desordenados no permite realizar búsquedas de memoria eficiente
- ✓ Hay muchos algoritmos de ordenamientos
- ✓ La meta es encontrar algoritmos cada vez más eficientes

Selection sort

Consiste en buscar el menor elemento, sacarlo de la lista, ponerlo en otra y repetir hasta que la lista original esté vacía.

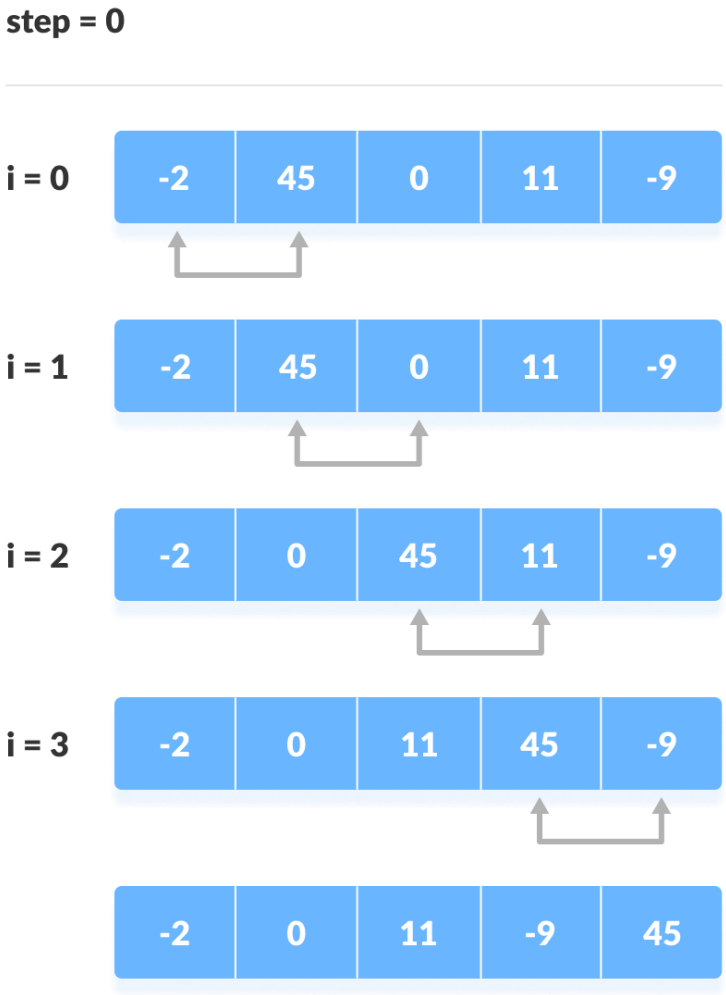
- Este enfoque es costoso en términos de espacio
- Se puede mejorar

```
static void selectionSort(){
    for (int current=0; current<end; current++){
        swap(current, minimo(current,end));
    }
}
```



Bubble sort

Consiste en “empujar” el mayor elemento hacia el final de la lista. Únicamente se comparan elementos adyacentes.

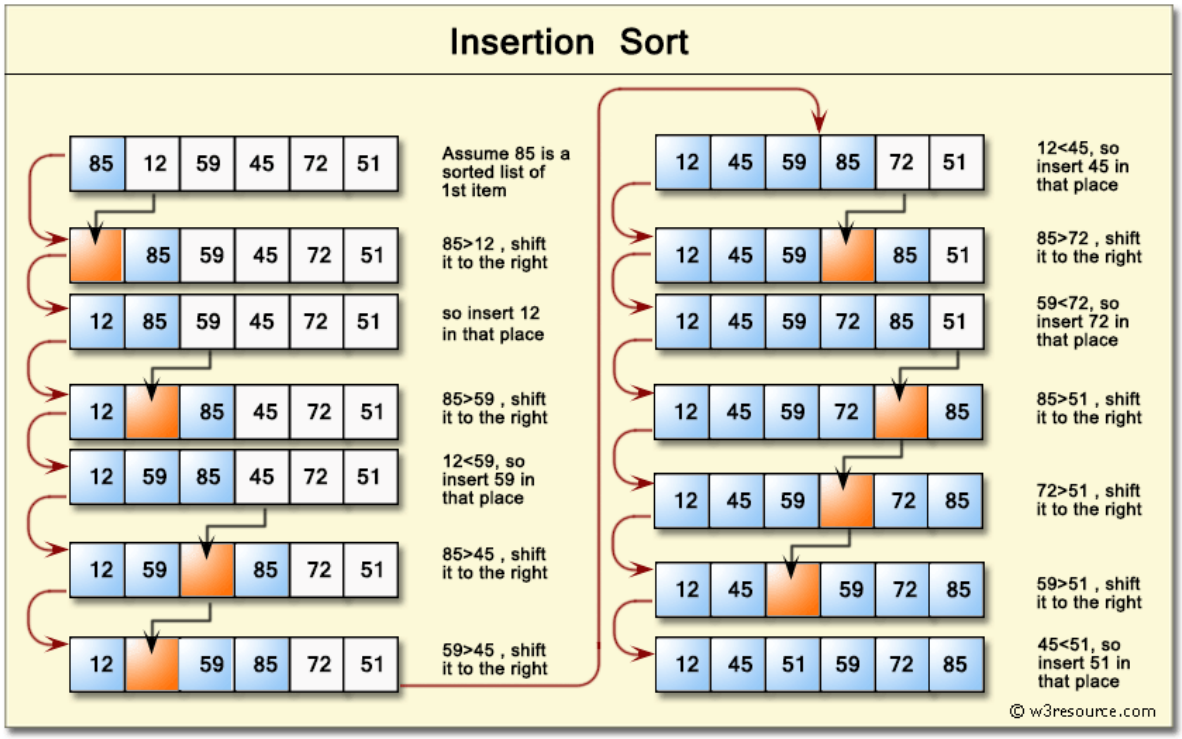


Insertion sort

- ✓ La lista se divide en dos partes:
  - Elementos ordenados
  - Elementos no ordenados
- ✓ El algoritmo saca el primer elemento de la lista desordenada y lo inserta en la lista ordenada
- ✓ No existen dos listas aparte, la división es “lógica”
- ✓ Al insertar en la lista ordenada, se va corriendo los elementos hasta encontrar la posición que le corresponde al elemento por insertar

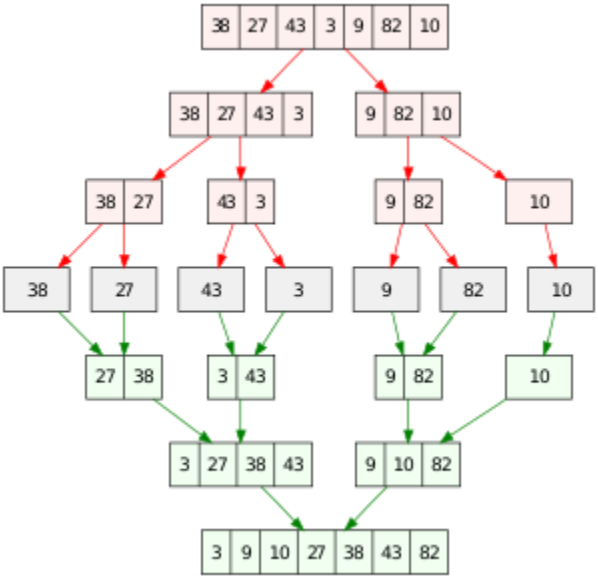
```
void insertionSort(int[] A){
    int in = 0;
    int out = 0;
    for (out=1; out<A.length; out++){
        int temp = A[out];
        in = out;
        while (in>0) && A[in-1]>=temp){
            A[in] = A[in-1];
            --in;
```

```
            }
            A[in] = temp;
        }
    }
```



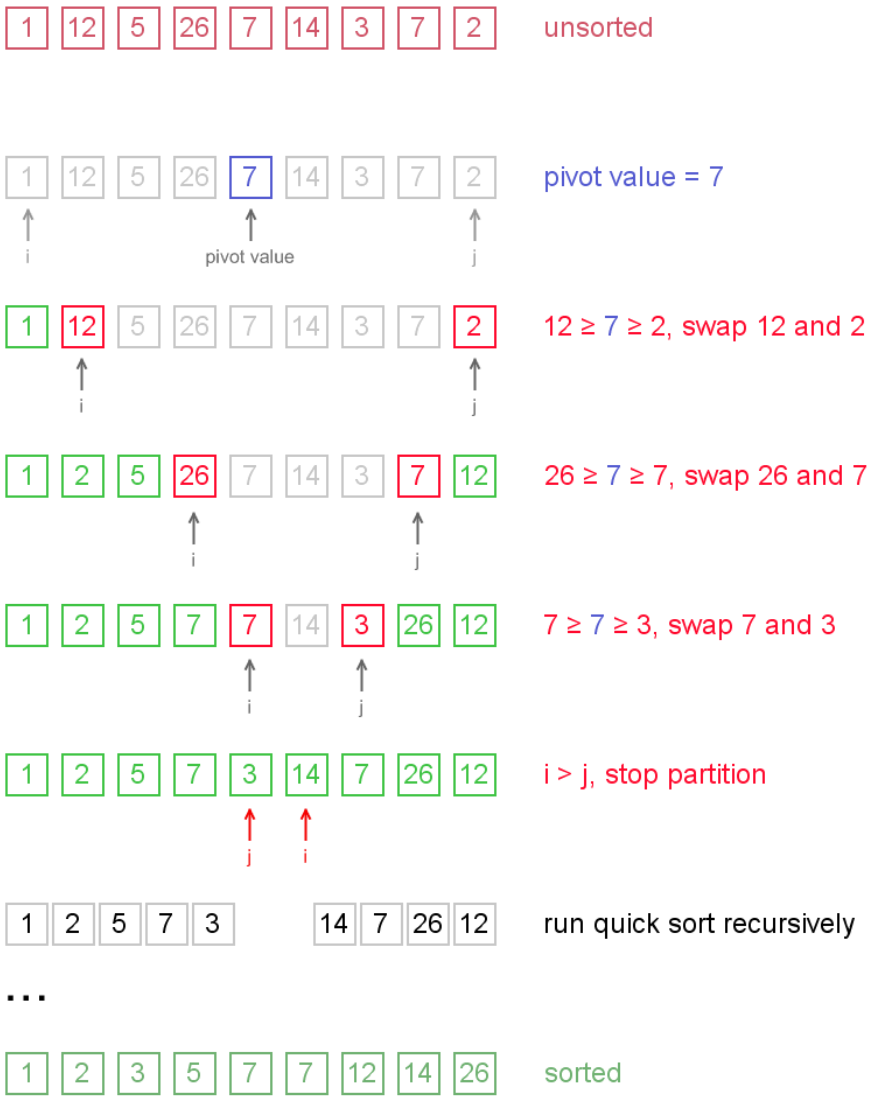
Merge sort

- ✓ Parte el array en dos
- ✓ Llama recursivamente en cada mitad
- ✓ Cuando la recursión llega a array de un elemento, se devuelve y empieza a unir cada subarray



Quick sort

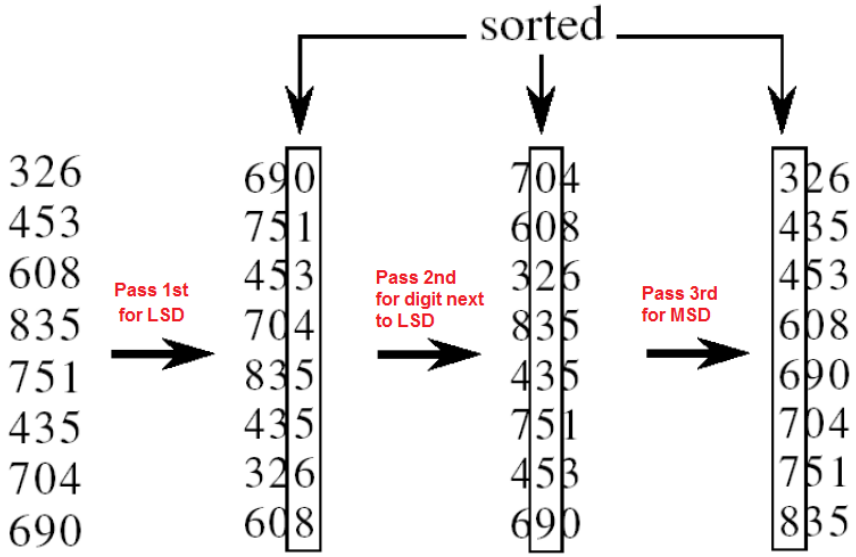
- ✓ El más popular de los algoritmos de ordenamientos
  - El más rápido en la mayoría de casos
  - Particiona el arreglo en dos y se llama recursivamente en cada mitad
- ✓ El paso inicial es calcular el pivote. El pivote es el elemento central del arreglo.
- ✓ Iterativamente compara los elementos previos y posteriores al pivote contra el valor del pivote
- ✓ Hace swap de los elementos para colocarlos en la mitad que le corresponde
- ✓ Cuando los índices se cruzan, se recorre en cada mitad



Radix sort

- ✓ Utiliza un enfoque distinto al resto de algoritmos de ordenamientos vistos hasta el momento
- ✓ Los algoritmos vistos consideran la llave (elemento de la lista) como una unidad indivisible
  - Por ejemplo: 100 se considera como el número 100
- ✓ Sin embargo, Radix sort divide la llave en cada uno de sus dígitos y ordena por cada uno de estos

- ✓ Radix: base (raíz del sistema numérico)
- ✓ Si se ordena números en base 10, radix = 10
- ✓ Si se ordena números en base 2, radix = 2



Binary search

- ✓ Busca un elemento en la lista / array ordenada
- ✓ Compara el elemento central del array contra el elemento buscado
- ✓ Si el central es igual al buscado, termina. Sino :
  - Central > buscado, se busca en la mitad inferior
  - Central < buscado, se busca en la mitad superior

```
public boolean binarySearch(int[] list, int start, int end, int elem) {
    int index = (start + end) / 2;
    if (list[index] == elem) {
        return true;
    } else if (list[index] > elem) {
        binarySearch(list, start, index, elem);
    } else if (list[index] < elem) {
        binarySearch(list, index + 1, end, elem);
    } else {
        return false;
    }
}
```

TEMA: Arboles binarios

Un árbol binario es una estructura de datos jerárquica, es decir la forma en la que se conectan los elementos permiten identificar niveles de importancia.

Los árboles se componen de nodos. Cada nodo contiene un dato y una o más referencias a otros nodos.

En un árbol binario, cada nodo tiene a lo sumo dos nodos.

Existen árboles n-arios en los que los nodos pueden tener más de dos nodos

Un árbol binario de búsqueda BST cumple con la condición:

- ✓ El hijo derecho es mayor que la raíz
- ✓ El hijo izquierdo es menos que la raíz
- ✓ Estas condiciones se cumplen en cualquier parte del árbol

Un BST provee búsqueda rápidas y eliminaciones / inserciones eficientes

¿Cómo se define un nodo en Java?

```
class Nodo {
    int element;
    Nodo right;
    Nodo left;
}
```

¿Cómo se define un árbol?

```
class BST{
    Nodo root = null;
    boolean isEmpty(){
        return root == null;
    }
}
```

¿Cómo se busca en el árbol?

```
public boolean contains(int e){
    return this.contains(e, root);
}
private boolean contains(int element, Node current){
    if (current == null){
        return false;
    }
    else if (element<current.element){
        return contains(element, current.left);
    }
    else if (element>current.element){
        return contains(element, current.right);
    }
```

```
    else {
        return true;
    }
}
```

¿Cómo encontrar el mayor / menor?

```
public Nodo findMin(){
    this.findMin(this.root);
}
private Nodo findMin(Nodo current){
    if (current==null){
        return null;
    }
    else if (current.left==null){
        return current;
    }
    else {
        return findMin(current.left);
    }
}
```

¿Cómo se inserta en un árbol?

```
public void insert(int e){
    root = this.insert(e, this.root);
}
private Nodo insert(int e, Nodo current){
    if (current==null){
        return new Nodo(e);
    }
    else if (e<current.element){
        current.left = insert(e, current.left);
    }
    else{
        current.right = insert(e, current.right);
    }
    return current;
}
```

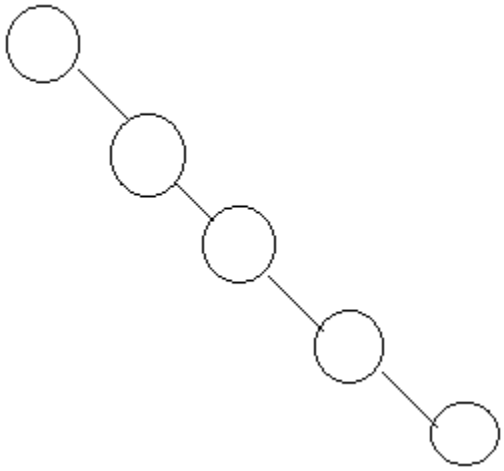
¿Cómo se elimina un element?

Hay 3 posibilidades:

1. Si el nodo por eliminar es una hoja: En este caso simplemente se “corta” la referencia
2. Si el nodo por eliminar tiene una solo hijo: En este caso se “salta” el nodo por eliminar y su hijo se reemplaza.
3. Si el nodo por eliminar tiene dos hijos: En este caso:
    a. Se busca el elemento menor en el subárbol derecho o el mayor en el subárbol izquierdo
    b. Se copia el elemento encontrado en “a” en el nodo por eliminar
    c. Se ejecuta el proceso de eliminación a partir de la posición en la que se copió el elemento encontrado en “a”

Arbol AVL

- ✓ Uno de los supuestos de los árboles es que proveen búsquedas rápidas
- ✓ Dependiendo del orden de inserción se puede obtener un árbol como:

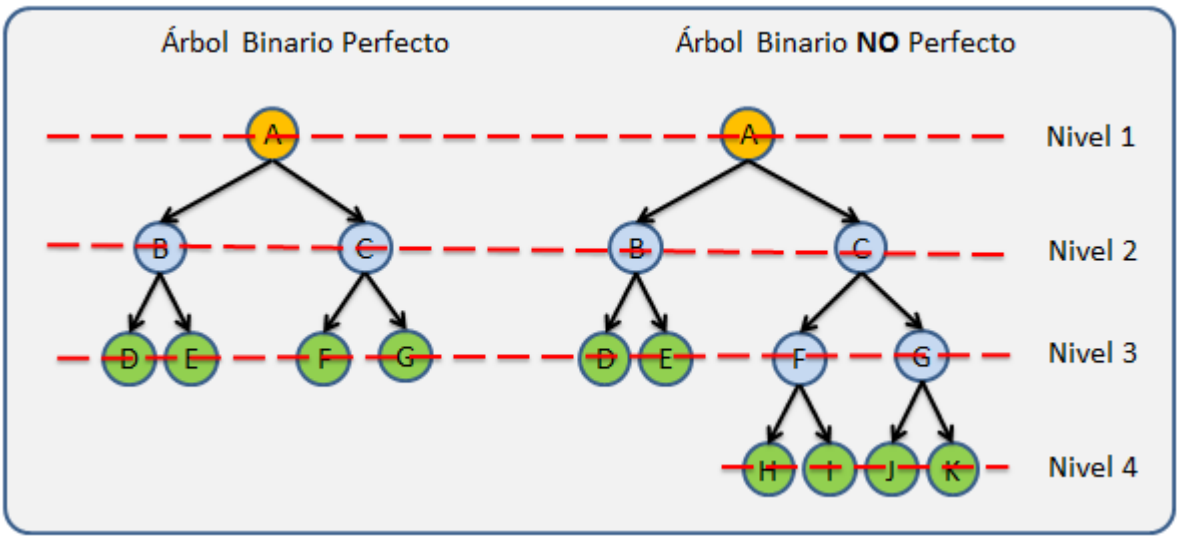


¿Qué sucede en este caso?

R/ Se pierde la rapidez para buscar y se reduce a una búsqueda secuencial

AVL significa: Adelson-Velskii-Landis

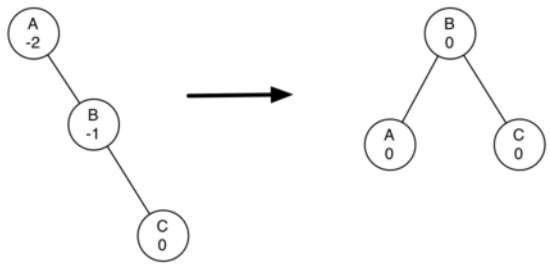
- ✓ Es un árbol binario de búsqueda con una condición de balance para asegurar que la profundidad sea óptima:
  - La altura en la izquierda no puede diferir más de 1 con respecto a la derecha
- ✓ Niveles de un árbol:



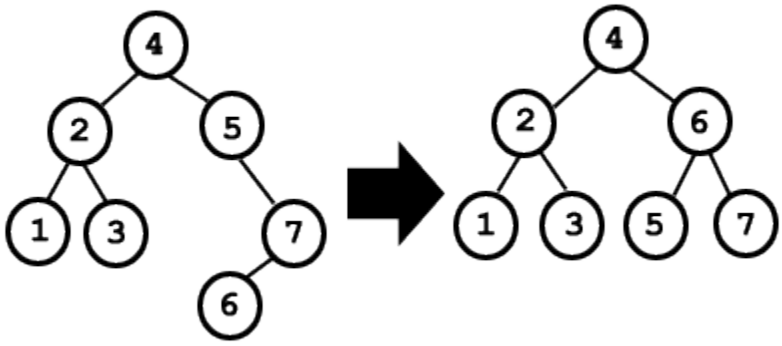
- ✓ La altura de un árbol es el nivel máximo más 1
- ✓ Cada nodo tiene el factor de balance como atributo

¿Cómo se inserta en un AVL?

- ✓ Igual que en un BST
- ✓ Para resultar en una violación del balance
- ✓ En caso de violación, se aplica una rotación
- ✓ Hay varios casos:
  - o II-DD: Se resuelve con una rotación sencilla

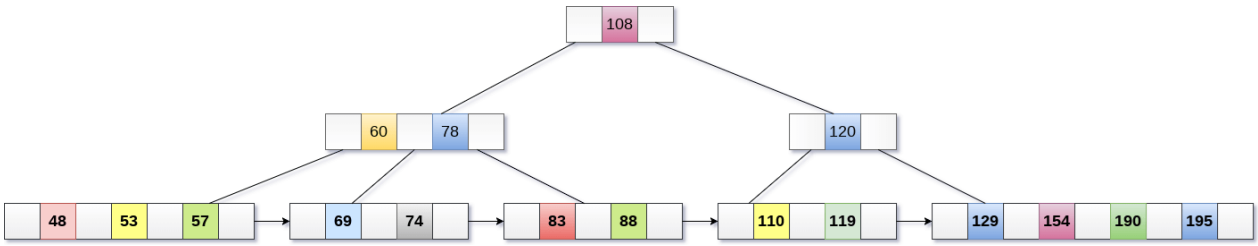


  - o ID-DI: Se resuelve con rotaciones dobles



Arboles B:

- ✓ Son árboles n-arios (Pueden tener más de dos hijos)
- ✓ Optimizados para estar almacenados en el disco. Los árboles vistos hasta el momento asumen que están en memoria ampliamente
- ✓ No es factible si hay muchos datos.
- ✓ El problema de tener una estructura en disco es la latencia asociada a estos
- ✓ Los discos son sumamente lentos en comparación a la memoria principal.
- ✓ Los B-trees están optimizados para no tener tantas operaciones en el disco
- ✓ Un B-tree gráficamente



- ✓ Un B-tree de un orden tiene las siguientes características:
  - Todas las hojas están al mismo nivel
  - Todos los nodos tienen a lo sumo m ramas y m-1 claves
  - Como mínimo cada nodo (excepto la raíz) tienen al menos m/2 ramas

Arboles de expresión:

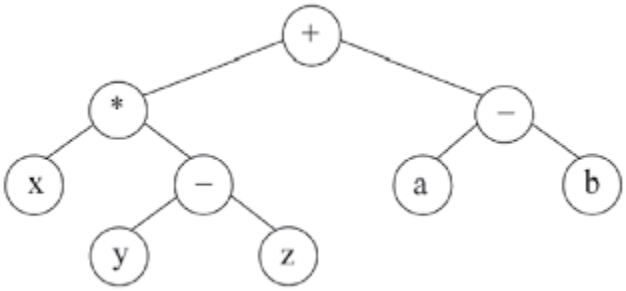
Una expresión es una secuencia de “tokens” (componentes léxicos que siguen una estructura específica)

- Operadores / palabras claves

Un árbol de expresión es un árbol con las siguientes características:

- Cada hoja es un operando
- Cada raíz son operadores
- Cada subárbol es una subexpresión

Los compiladores usan árboles de expresiones extensamente



Para convertir el árbol a expresión, simplemente se debe recorrer el árbol con alguna de las siguientes formas:

- Infijo (hijo izq, raíz, hijo derecho)
- Postfijo (izq, derecho, raíz)
- Prefijo (raíz, izquierdo, derecho)

```
infijo(nodo){
     if(nodo != null){
         if(nodo==operador){
                 print("(");
         }
         infijo(nodo.izq);
```

```
        print(nodo.value);
        infijo(nodo.derecho);
        if(nodo == operador){
         print(")");
        }
    }
```

TEMA: Estructura de datos generales

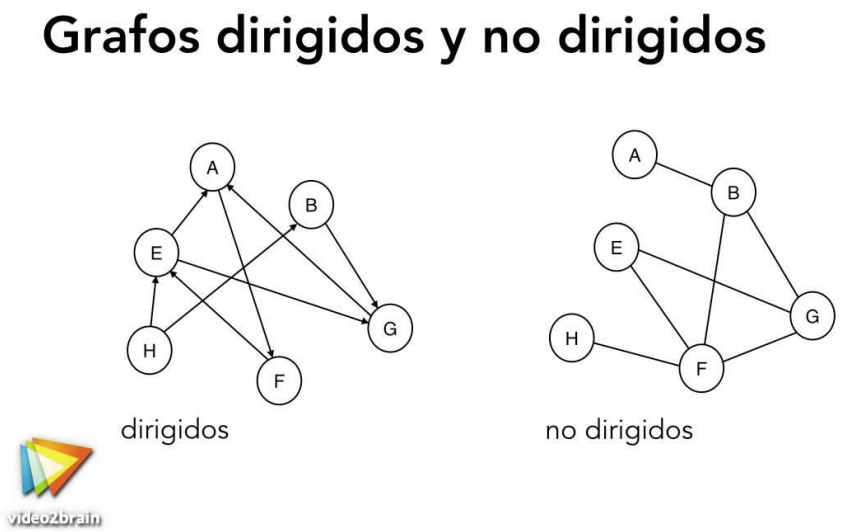
Los grafos son estructuras de datos generales aplicables en muchas áreas como:

- Sociología
- Química
- Geografía
- Criminología

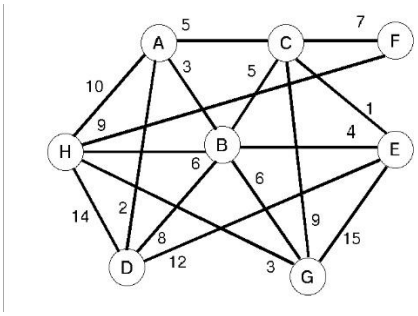
Un grafo G agrupa entidades que representan conceptos.

Se compone de vértices/nodos que representan cada entidad y de aristas/arcos que representan relacionados entre los nodos.

Un grupo puede ser dirigido o no dirigido. El ejemplo anterior es un grafo no dirigido. El siguiente es un grafo dirigido.



Los arcos pueden tener magnitudes, creando un grafo con peso



El grado de un nodo es la cantidad de conexiones en las que participa

En un grafo dirigido, hay nodo saliente y entrante

Un camino P={V0,V1,V2,…., Vn}

Un ciclo es un camino que empieza y termina en el mismo punto

Un grafo es acíclico si no tiene ciclos

Un DAG (Directed Aciclic Graph): es un nodo dirigido sin ciclos

¿Cómo se implementa un grafo?

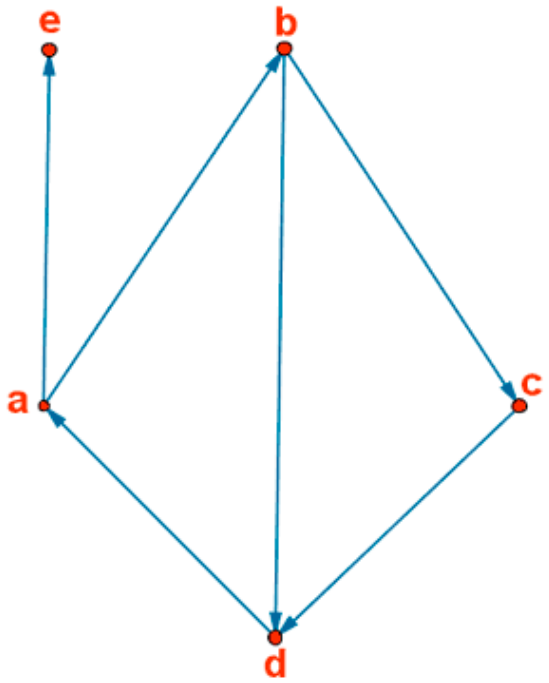
R/ Hay dos formas:

- Matriz de adyacencias
- Lista de adyacencias

Matriz de adyacencias: Dado un grafo de n vértices, se construye una matriz de nxn donde cada elemento aij tiene uno de los siguientes valores:

```
* 1 si hay arco de Vi-Vj
* 0 en caso contario
```

Una lista de adyacencia es una lista enlazada donde cada elemento es un nodo de grafos. Cada elemento tiene una lista con sus conexiones



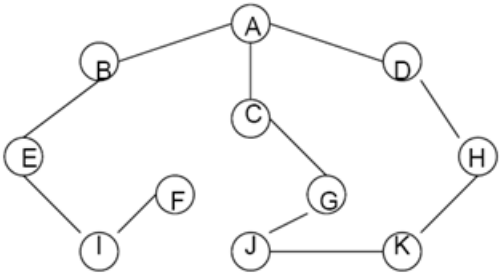
	a	b	c	d	e
a	0	1	0	0	1
b	0	0	1	1	0
c	0	0	0	1	0
d	1	0	0	0	0
e	0	0	0	0	0

Elegir entre uno y otra forma depende de:

- Algoritmos por aplicar
- Densidad del grafo
- ALTA-Matriz
- BAJA-lista

¿Cómo se recorre un grafo?

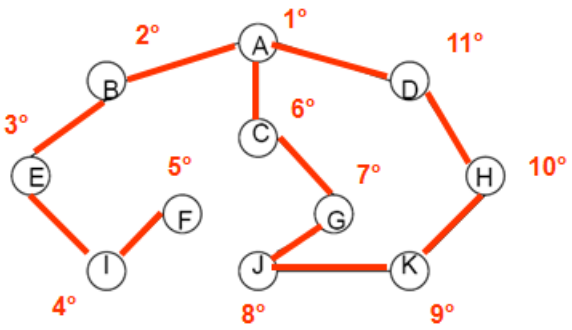
R/ Se recorre desde un nodo específico y hasta recorrer todos los nodos alcanzable de las siguientes dos formas:



Recorrido de un grafo en anchura.

A-B-C-D-E-G-H-I-J-K-F

Pre-Orden



Recorrido de un grafo en profundidad

Solución: A-B-E-I-F-C-G-J-K-H-D

La ruta más corta entre un nodo y el resto

- Es uno de los problemas más comunes de los grafos
- Edgar Dijskta propone un algoritmo que calcula la ruta más corta.

  * Grafo por peso
  * Grafo dirigido

Pasos

- Asignar a todos los nodos infinito menos al inicial
- Calcular el valor a todos los nodos a partir de A.
- Selecciona el nodo menor y nos movemos a este. Se recalcula las distancias usando el nuevo nodo como "puente".
- Se sigue seleccionando el menor no visitado y se recalcula.

Algoritmo de Washall (Cierre transitivo)

- Permite determinar si hay camino entre cualquier par de nodos del grafo.
- Calcule la matriz de caminos de un grafo G de n vértices representando la matriz de adyacencia A
- Define una secuencia de matrices Po, P1,...,Pn. Los elementos de cada matriz Pk[i,j] tienen un cero si no hay camino, o 1 si hay camino de Vi a V, através de Vk.

P0	**1**	**2**	**3**	**4**	**5**
1	1	1	0	0	0
2	0	1	0	1	1
3	1	1	1	0	1
4	1	0	0	1	1
5	0	0	0	1	1

- A partir de la matriz Pk-1

P1	**1**	**2**	**3**	**4**	**5**
1	1	1	0	0	0
2	0	1	0	1	1
3	1	1	1	0	1
4	1	**1**	0	1	1
5	0	0	0	1	1

```
* Hay camino de V4 a V2 a través de V1
```

P2	**1**	**2**	**3**	**4**	**5**
1	1	1	0	**1**	**1**
2	0	1	0	1	1
3	1	1	1	1	1
4	1	1	0	1	1
5	0	0	0	1	1

```
* Así sucesivamente hasta llegar a P5
```

Algoritmo de camino más corto para todos los vértices

- Se podría aplicarDijkstra para todos los vértices pero sería ineficiente.

- El agoritmo de FLoyd-Washall es más directo para calcular los rutas óptimas para todos los vértices.

- El grafo se representa con la matriz de pesos. Si no hay arco/ arista se conserva infinito.

- La diagonal es infinito.

- Al igual que Warshall, genera n matrices de nxn. En cada paso, se incorpora un nuevo vértice y se evalúa a ratos.

- De igual forma, se generan n matrices predecesoras.

```
Wo[i,j]   ----> Cij peso del arco de V1 A v
              ----> inf si no hay arco
```

W1[i,j] = mínimo (D0[i,j], D0[i,1], D0[1 ,j])

WK[i,j] = min (Dk-1[i,j], Dk-1[i,k] + Dk-1[k,j])