

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos: POO y EDD

Profesor: Hans Löbel

Cuando hablamos de un curso, ¿en qué estamos pensando?

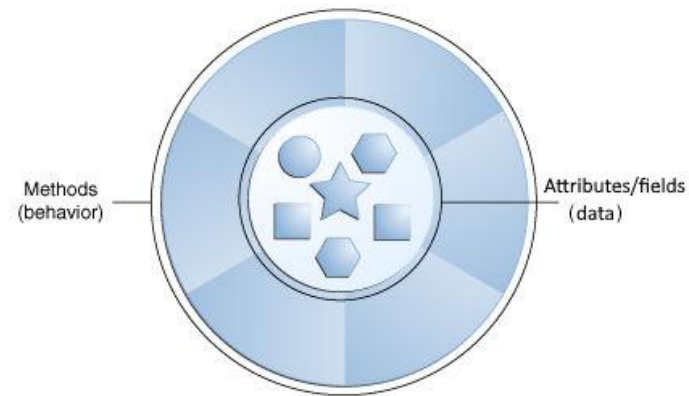


Todas estas maneras de “modelarlo” representan distintas abstracciones del concepto curso, cada una más o menos adecuada para distintas tareas.

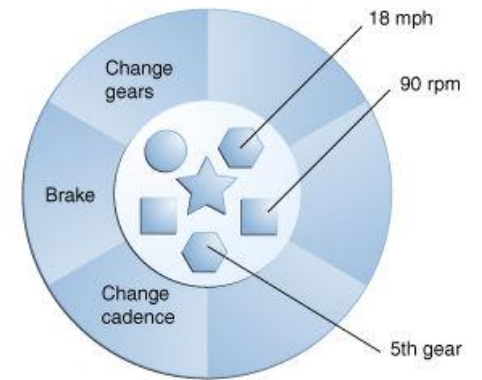
Objetos de software combinan ambas ideas

En el desarrollo de software, un objeto es una colección de **datos** que además tiene asociado **comportamientos**.

- Datos: **describen** el estado y/o composición de los objetos. Se les conoce como **atributos** o **campos** del objeto.
- Comportamientos: **representan acciones** que realiza el objeto, o realizan sobre él, que pueden generar cambios en su estado. Se les conoce como **métodos** del objeto.



(a) A software object



(b) Bicycle modelled as a software object

Ejemplo: datos y comportamiento

| Clase: Auto | |
|------------------|---------------------------------------|
| Datos | Comportamiento |
| Marca | Calcular próxima mantención |
| Modelo | Calcular distancia a alguna dirección |
| Color | Pintar de otro color |
| Año | Realizar nueva mantención |
| Motor | |
| Kilometraje | |
| Ubicación actual | |

¿Qué es entonces OOP?

La programación orientada a objetos (OOP) implica que los programas estarán orientados a modelar las funcionalidades a través de la interacción entre objetos por medio de sus datos y comportamiento.

Para definir un objeto, creamos una plantilla llamada **clase**

Cada objeto es una **instancia** de la clase Auto

Objeto 1



Objeto 2



Objeto 3



Clase **Auto**

```
1 class Departamento:
2     def __init__(self, _id, mts2, valor, num_dorms, num_banos):
3         self._id = _id
4         self.mts2 = mts2
5         self.valor = valor
6         self.num_dorms = num_dorms
7         self.num_banos = num_banos
8         self.vendido = False
9
10    def vender(self):
11        if not self.vendido:
12            self.vendido = True
13        else:
14            print("Departamento {} ya se vendió".format(self._id))
```

```
1 d1 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
2 print(d1.vendido)
3 d1.vender()
4 print(d1.vendido)
5 d1.vender()
```

False

True

Departamento 1 ya se vendió


```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

False

False

True

True

Un concepto fundamental es el de **interfaz** de un objeto

Existen atributos de los objetos que no necesitan ser visualizados ni accedidos por los otros objetos con que se interactúa.



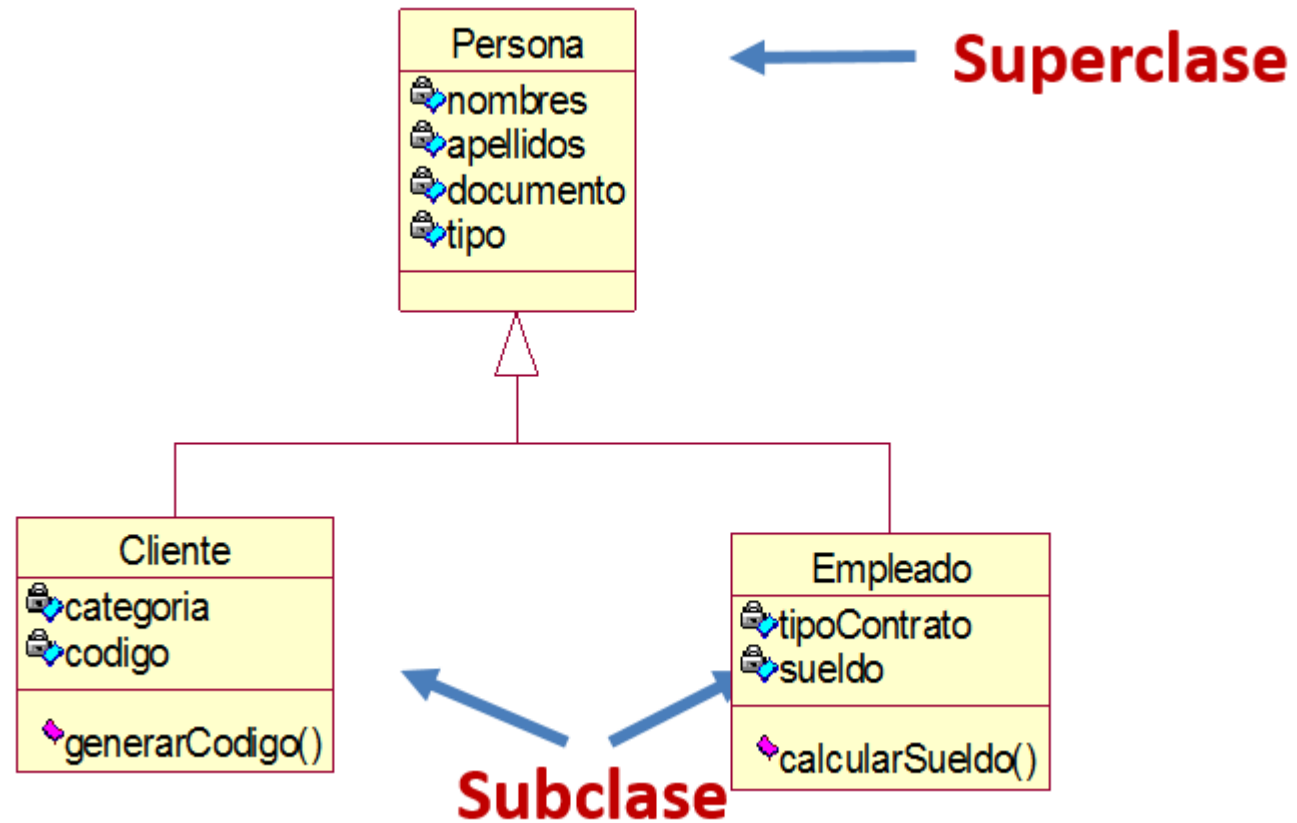
Un concepto fundamental es el de **interfaz** de un objeto



| Interface |
|---|
| Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel |
| Current channel Volume level |

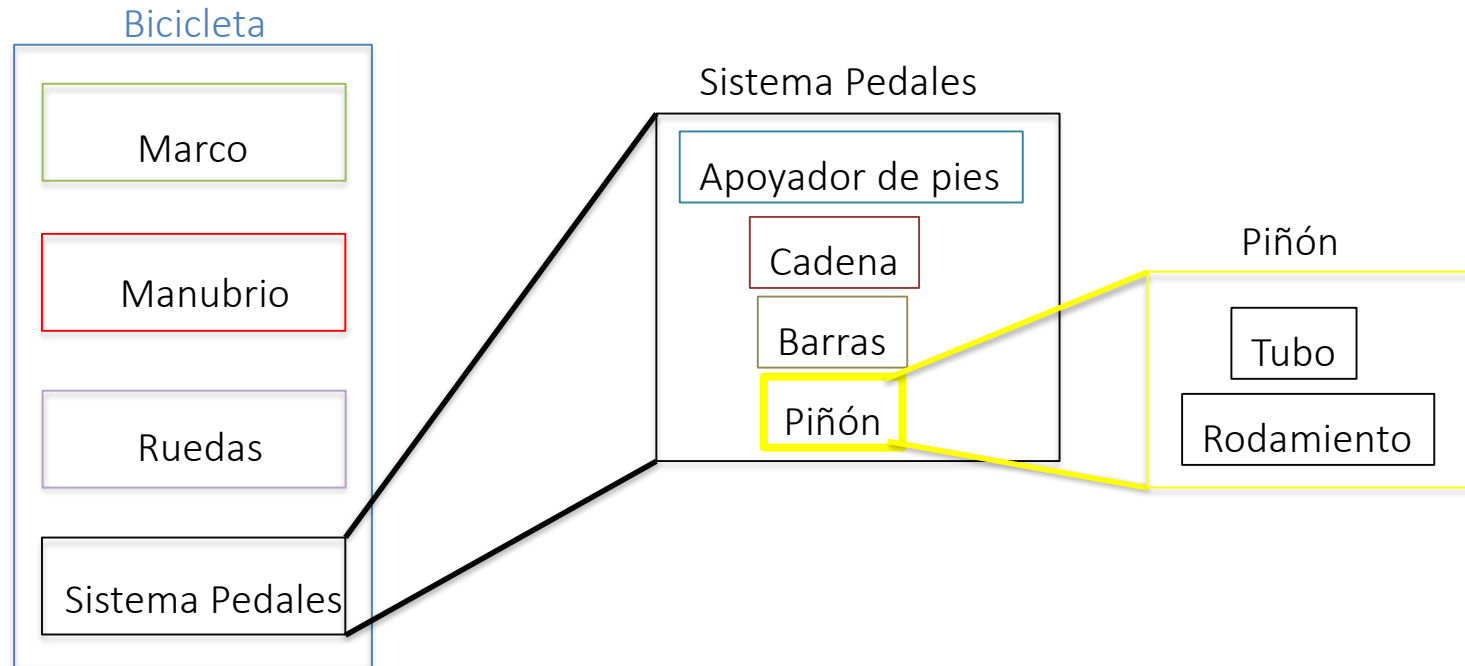
```
1 class Televisor:
2     ''' Clase que modela un televisor.
3     '''
4
5     def __init__(self, pulgadas, marca):
6         self.pulgadas = pulgadas
7         self.marca = marca
8         self.encendido = False
9         self.canal_actual = 0
10
11     def encender(self):
12         self.encendido = True
13
14     def apagar(self):
15         self.encendido = False
16
17     def cambiar_canal(self, nuevo_canal):
18         self._codificar_imagen()
19         self.canal_actual = nuevo_canal
20
21     def __codificar_imagen(self):
22         print("Estoy convirtiendo una señal eléctrica en la imagen que estás viendo.")
```

Herencia nos permite modelar clases similares sin reescribir todo de nuevo



```
1 import numpy as np
2
3 class Variable:
4     def __init__(self, data):
5         self.data = np.array(data)
6
7     def representante(self):
8         pass
9
10 class Ingresos(Variable):
11     def representante(self):
12         return np.mean(self.data)
13
14 class Comuna(Variable):
15     def representante(self):
16         ind = np.argmax([np.sum(self.data == c) for c in self.data]) # el que mas se repite
17         return self.data[ind]
18
19 class Puesto(Variable):
20     categorias = {'Gerente': 1, 'SubGerente': 2, 'Analista': 3,
21                  'Alumno en Practica': 4} # class (or static) variable
22
23     def representante(self):
24         return self.data[np.argmin([Puesto.categorias[c] for c in self.data])]#la categoria mas alta acorde con el diccionario
```

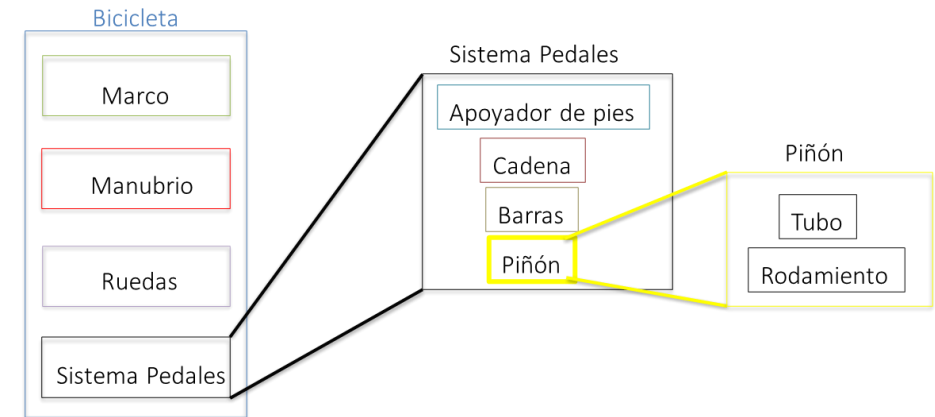
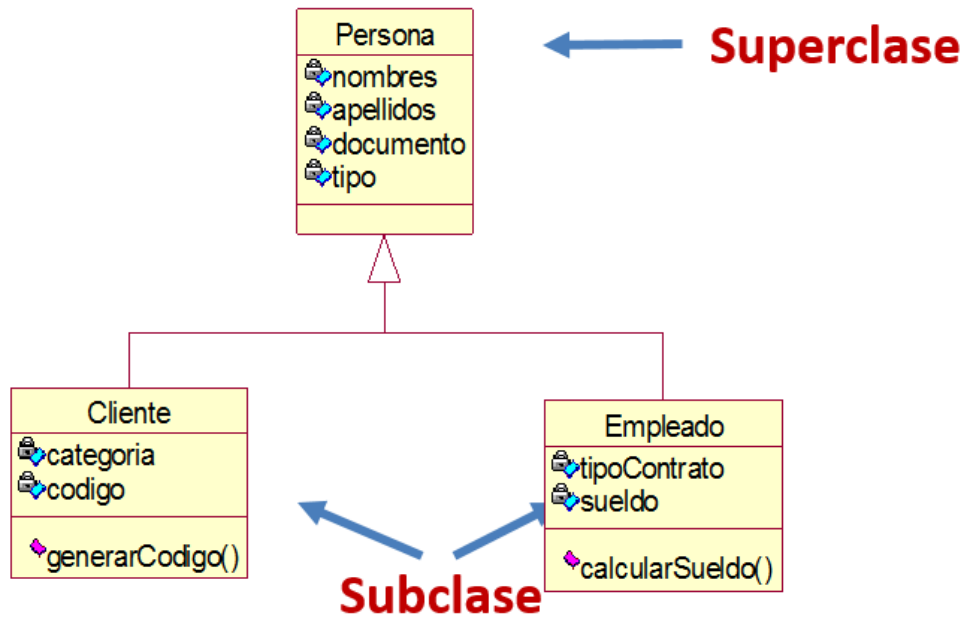
También es posible modelar **objetos** como atributos de otros objetos, mediante **agregación o composición**



Agregación: atributo existe de manera independiente al contenedor

Composición: atributo no puede existir de independiente del contenedor

¿Cómo se comparan herencia y agregación/composición?



¿Cómo se comparan herencia y agregación/composición?

- NO TIENEN MUCHO QUE VER EN REALIDAD!
- Si bien ambos son mecanismo para modelar, estructuralmente difieren de manera fundamental.
- **Herencia** busca facilitar la **especialización** de las clases, sin requerir repetir código.
- **Agregación y composición** buscan aumentar el nivel de **abstracción** de las clases, al permitir tipos de dato complejos (otras clases) como atributos.

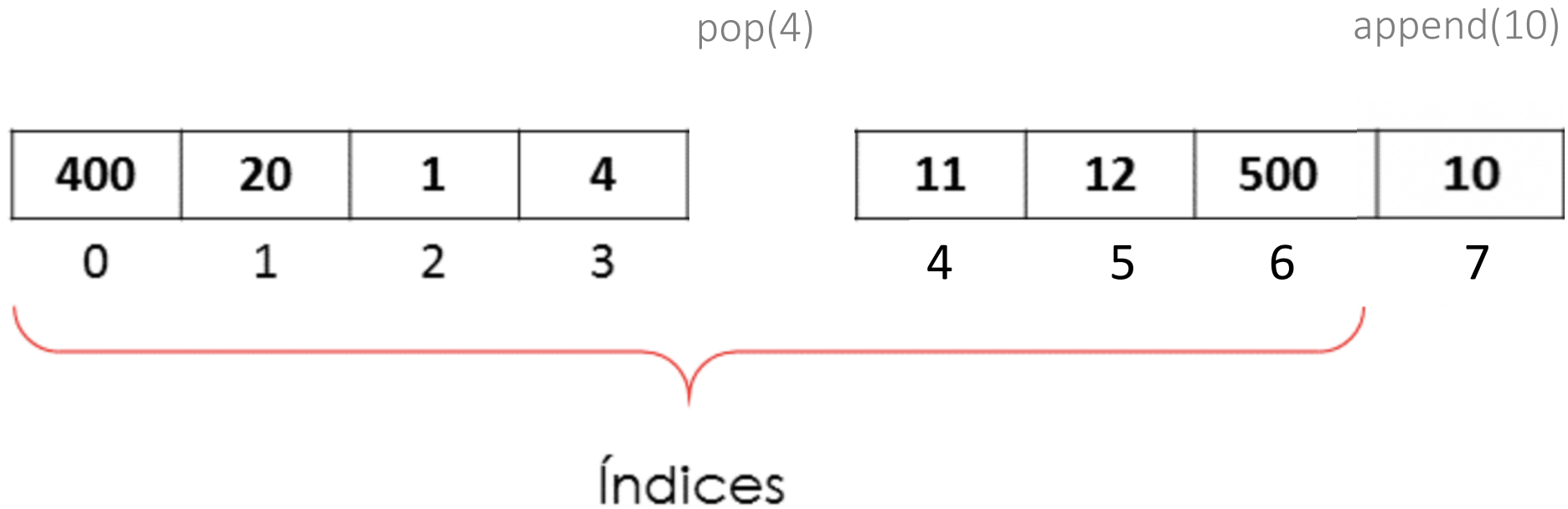
Movámonos ahora a las estructuras de datos (EDD)

Son **tipos de dato especializados**, diseñados para **agrupar, almacenar o acceder** a la información de manera más **eficiente** que un tipo de dato básico (como int, float, etc). Algunos ejemplos son los siguientes:

- Clases
- Listas
- Tuplas
- Diccionarios
- Árboles

Listas


- Las listas son estructuras que guardan datos de forma **ordenada**.
- Son mutables (modificables).



Tuplas

- Similares a las listas, permiten manejar datos de forma ordenada.
- Al igual que las listas, se accede a los datos mediante índices basados en el orden que fueron ingresados.
- A diferencia de las listas, son **inmutables**.

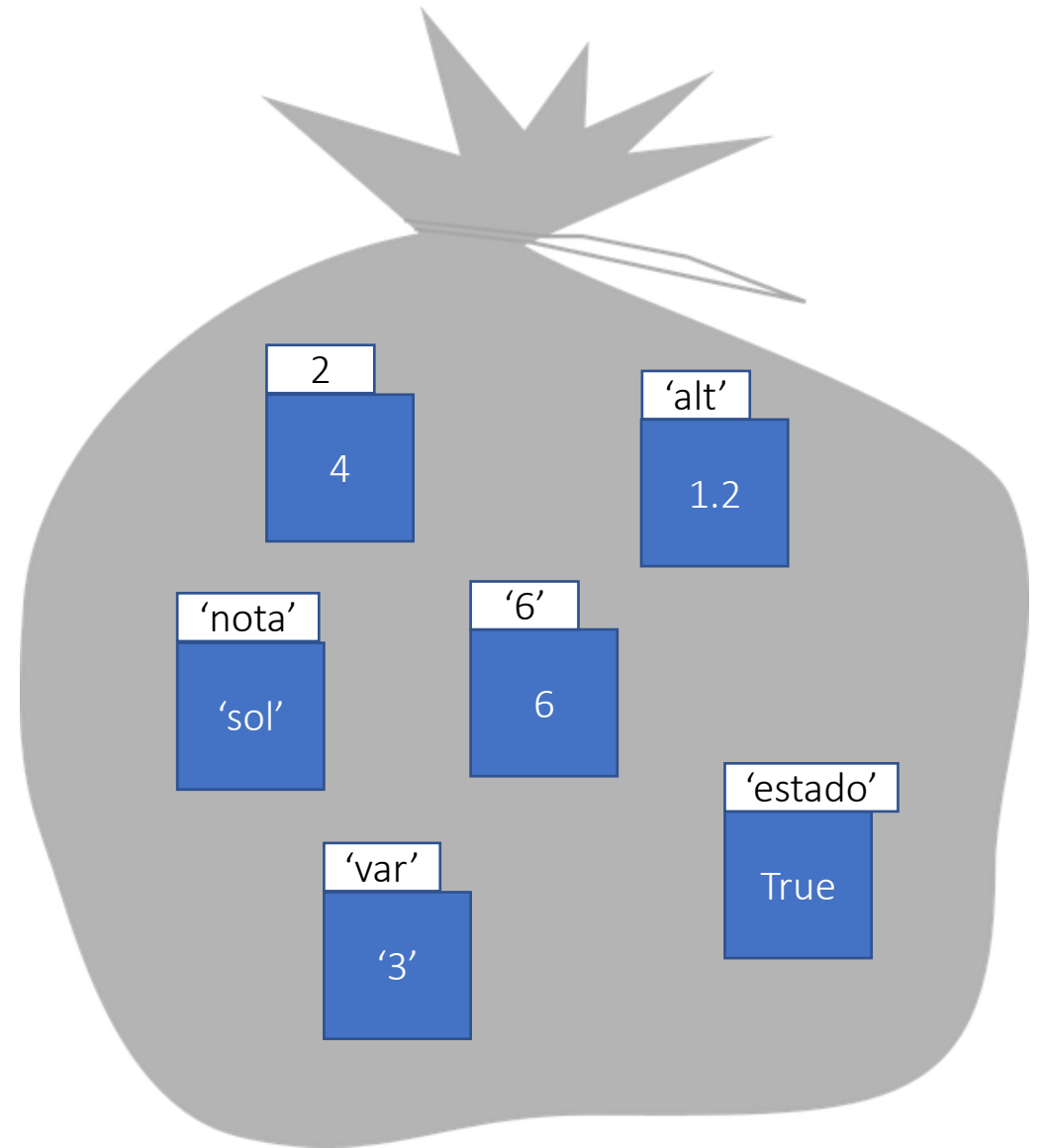
| | | | | | | | |
|------------|-----------|----------|----------|-----------|-----------|-----------|------------|
| 400 | 20 | 1 | 4 | 10 | 11 | 12 | 500 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |



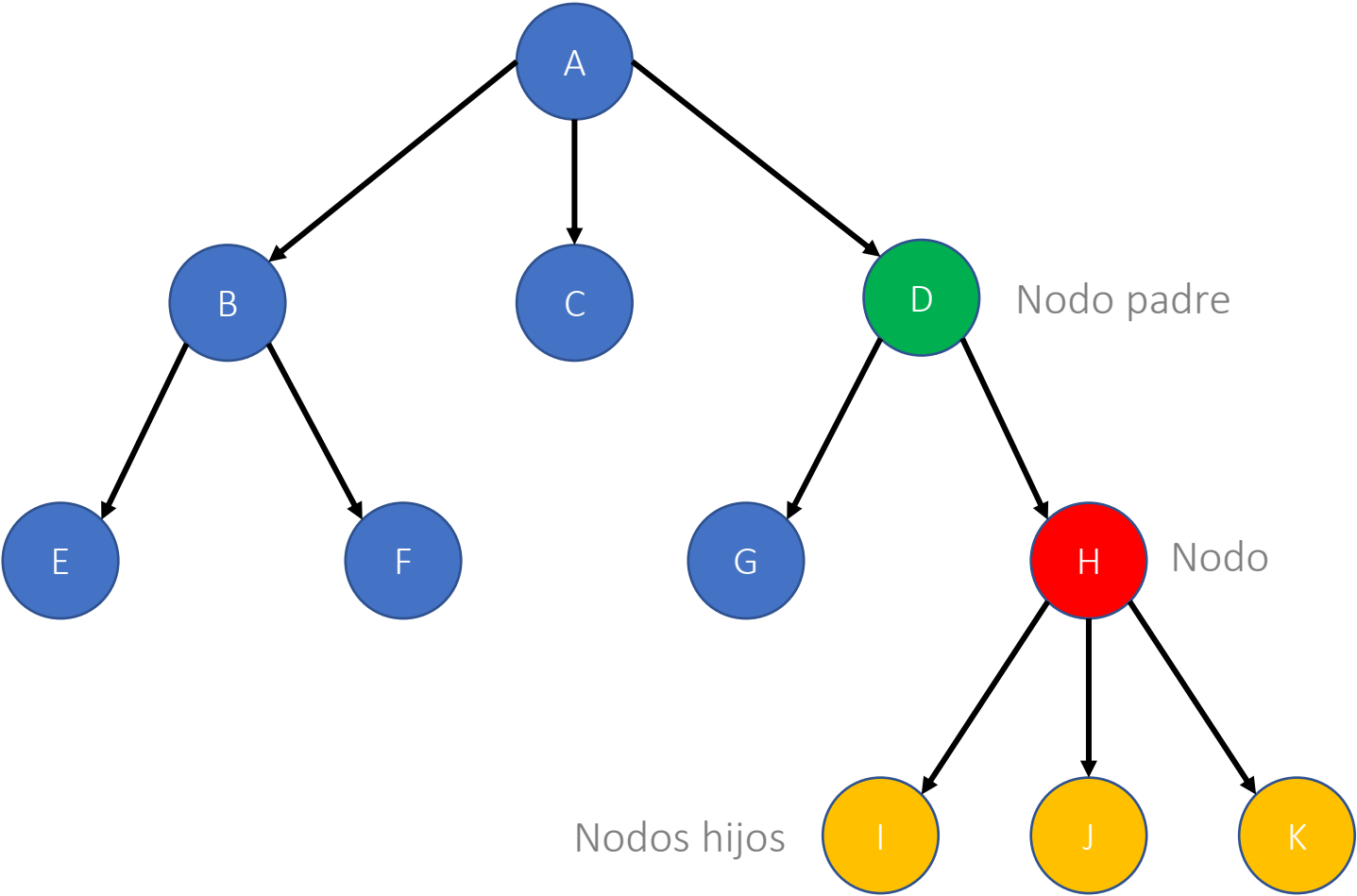
Índices

Diccionarios

- Permiten almacenar datos basados en una asociación de pares de elementos, a través de una relación **llave-valor**.
- Acceso a valores a través de la llave es instantáneo, no se necesita realizar una búsqueda (análogo a un índice).
- Se prefiere a una lista cuando el caso de uso más común no implica revisar todos los elementos, sino solo algunos fácilmente encontrables a través de la llave.

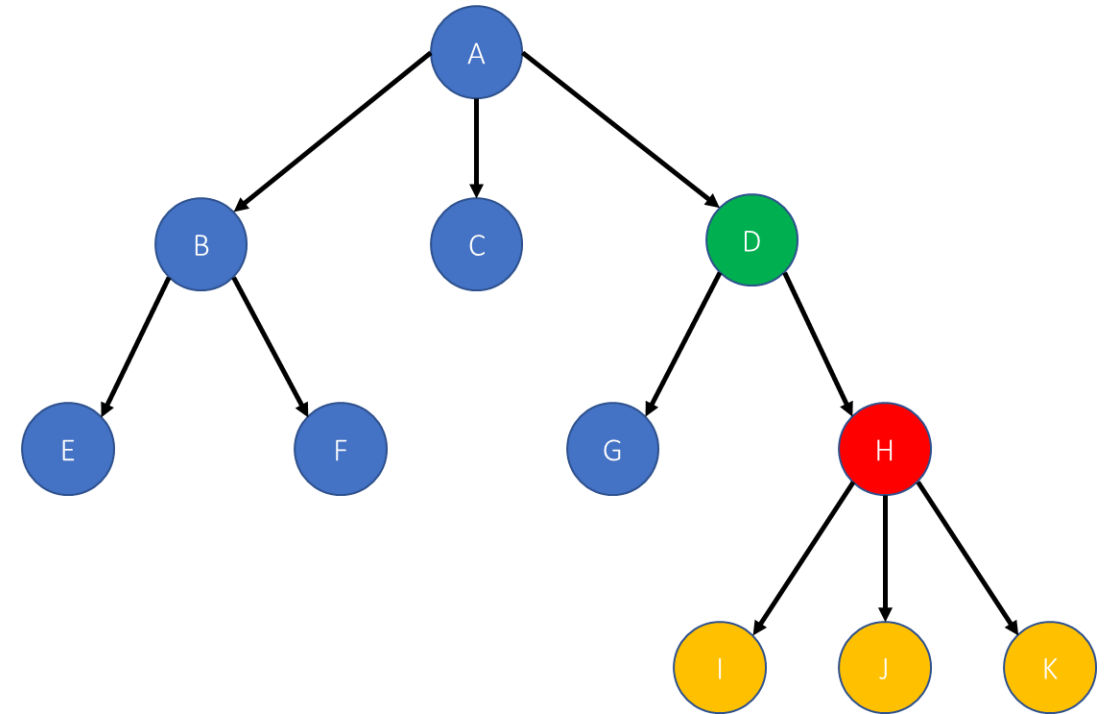


Árboles

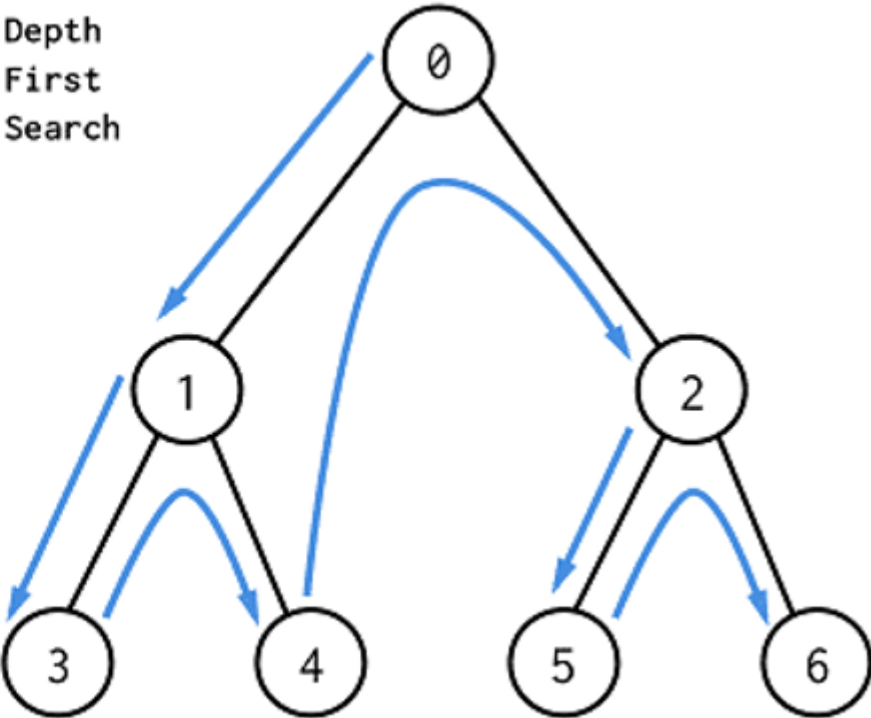
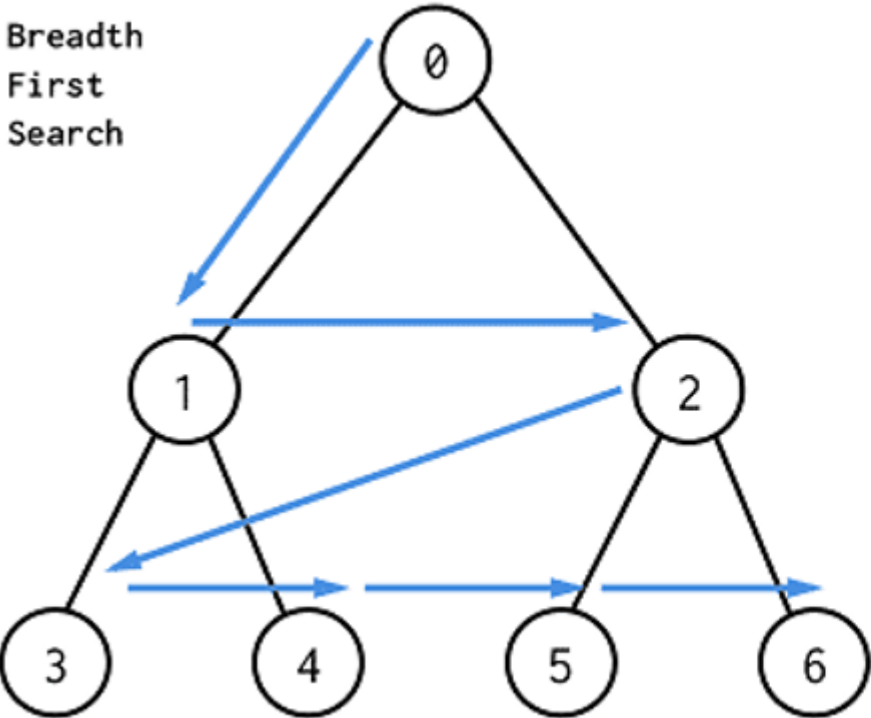


¿Cuándo son útiles los árboles como estructura de datos?

- Cuando los datos tienen una estructura jerárquica: rutas en una red de transporte o un organigrama.
- Para buscar cosas rápido: autocompletar de búsqueda en Google
- Para encontrar el camino más corto en una red.
- Para tomar decisiones paso a paso: en IA, por ejemplo, cuando se analizan varias opciones antes de elegir la mejor.
- Mucho de esto se logra mediante búsqueda eficiente en árboles.

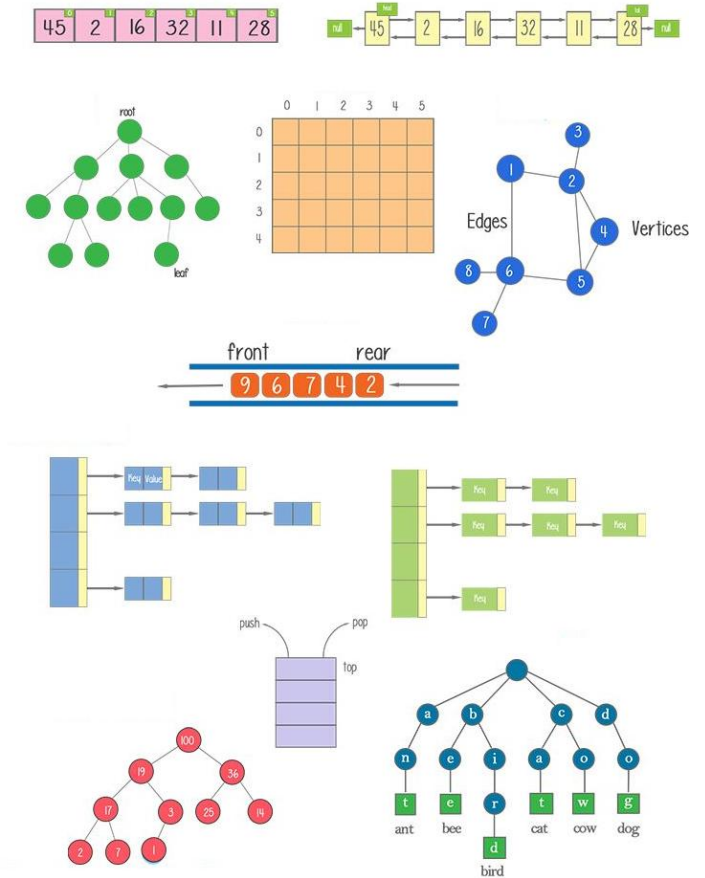


Búsqueda en amplitud y en profundidad son las más utilizadas



Un breve y somero resumen

- Las estructuras de datos (incluyendo las clases) corresponden a tipos de dato especializados, **diseñados para organizar, almacenar y/o acceder la información de manera más eficiente** que un tipo de dato básico.
- La elección adecuada de la estructura de datos, o la manera de organizar las clases, es fundamental para el desarrollo de un buen programa y muchas veces es la única posibilidad para solucionar un problema de forma realista.
- Pero siempre es conveniente pensar primero en una solución básica a los problemas, y luego incorporar las estructuras donde corresponda.



Cómo sigue la sesión de hoy

- Revisión breve de los ejercicios
- Trabajo personal o grupal
- Entrega final avance y Ticket de salida (17:10 a 17:30)

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos: POO y EDD

Profesor: Hans Löbel