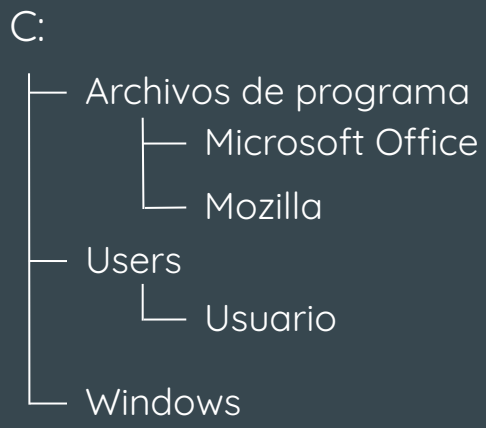
Árboles

...

Algoritmos y Estructuras de Datos
2024

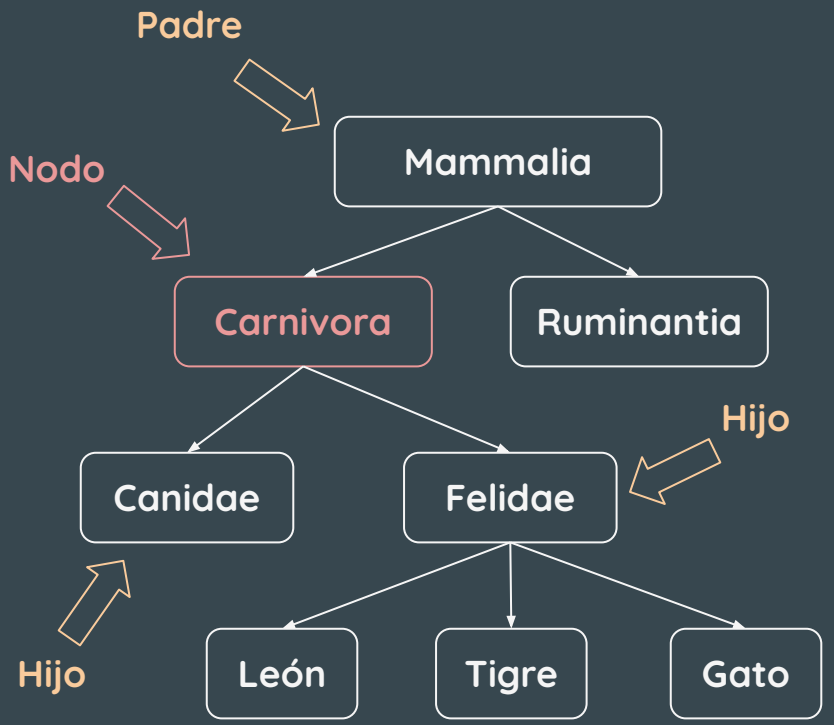
UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Cómo representaría la organización de los archivos y carpetas en un sistema operativo como Windows o Linux?



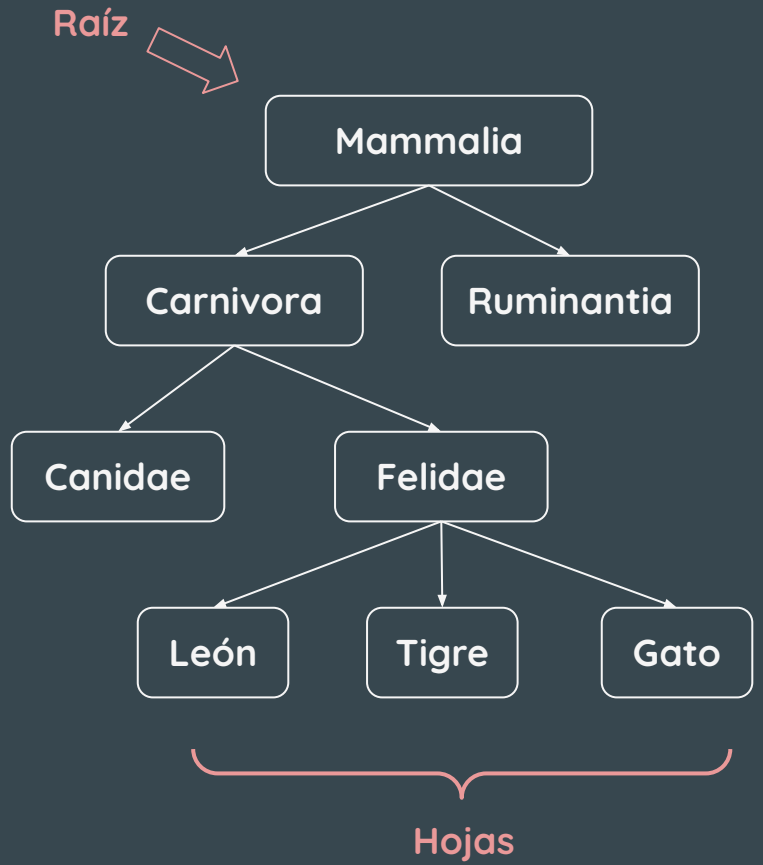
Definición

- Un árbol es una estructura de datos abstracta que ordena sus elementos de manera **jerárquica**.
- Están compuestos por nodos que poseen un **nodo padre** y 0 o más **nodos hijos**.



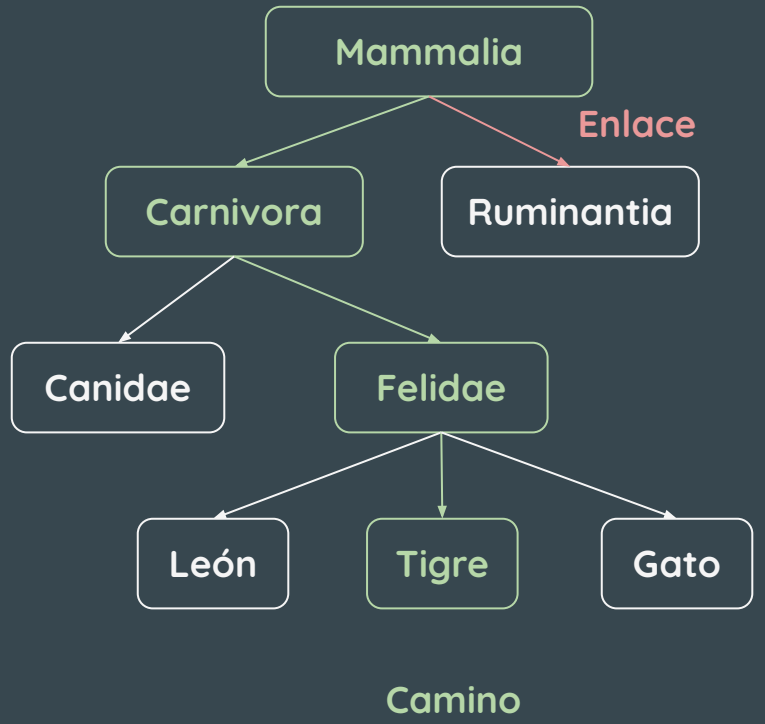
Definición

- El nodo **raíz** es el nodo principal de un árbol y es el único sin nodo padre.
- Los nodos al final del árbol, sin nodos hijos, se denominan **hojas**.

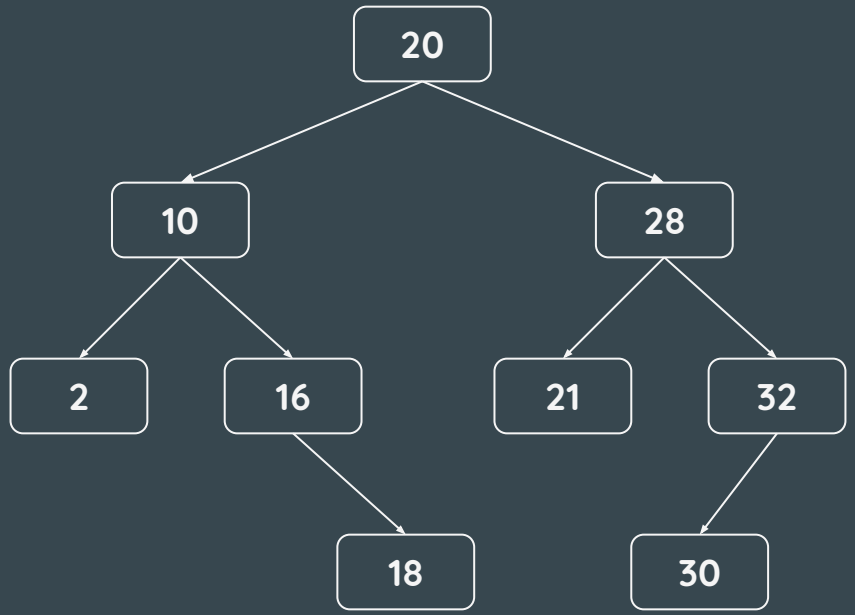


Más terminología

- La conexión entre dos nodos se denomina **enlace** o **unión** (***edge*** en inglés).
- La sucesión de nodos enlazados que conecta dos nodos específicos se denomina **camino** (***path***)



Árboles binarios



- Son una subclase de árboles donde cada nodo puede tener, a lo sumo, **dos nodos hijos**.
- Estos nodos son denominados normalmente **izquierda** y **derecha**.

Implementación: clase Nodo

- Para definir un nodo, es necesario primero definir su **elemento** o valor.
- Para los árboles binarios se requieren 2 punteros: **izquierda** y **derecha**.
- Aunque no es un requerimiento, sumar el puntero **padre** puede facilitar la implementación de algunas funciones

```
class Nodo:
  def __init__(self, e):
    self._elemento = e
    self._izquierda = None
    self._derecha = None
    self._padre = None
```

Implementación: clase ArbolBinario

- La clase árbol tiene que tener sí o sí definido cual es el **nodo raíz**.
- Al momento de iniciar, se pueden agregar más variables, como por el ejemplo el tamaño (cantidad de nodos)

```
class ArbolBinario:
  def __init__(self):
    self._raiz = None
    self._tamano = 0
```

Primeras funciones (I)

```
class ArbolBinario:
  def __len__(self):
    return self._tamano

  def vacio(self):
    if self._tamano == 0:
      return True

  def es_raiz(self, actual):
    if self._raiz == actual:
       return True

  def es_hoja(self, actual):
    if  actual._izquierda == None and
        actual._derecha == None:
      return True
```

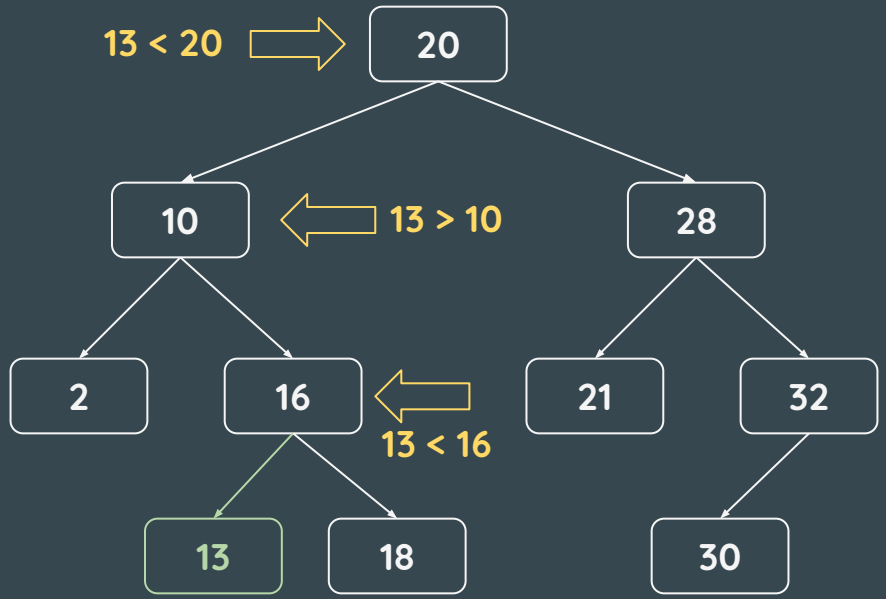
Primeras funciones (II)

```
class ArbolBinario:
  def raiz(self):
    return self._raiz

  def izquierda(self, actual):
    if actual._izquierda != None:
      return actual._izquierda

  def derecha(self, actual):
    if actual._derecha != None:
      return actual._derecha

  def padre(self, actual):
    if actual._padre != None:
    # Alternativa “not (es_raiz(n))”
      return actual._padre
```

Agregar nodos (I)

- En general, los datos en un árbol son ubicados en un árbol en relación con su valor.
- Si el valor a agregar es mayor que el de un nodo, se lo agrega en el **subárbol de la derecha**; sino en el **subárbol de la izquierda**.

Agregar nodos (II)



- Si se quisiera agregar el valor **13**, se comienza comparando desde la raíz.
- Se visita, recursivamente, un nodo hijo dependiendo de si el valor a ingresar es mayor o menor que el almacenado en el nodo.

Agregar nodos: pseudocódigo

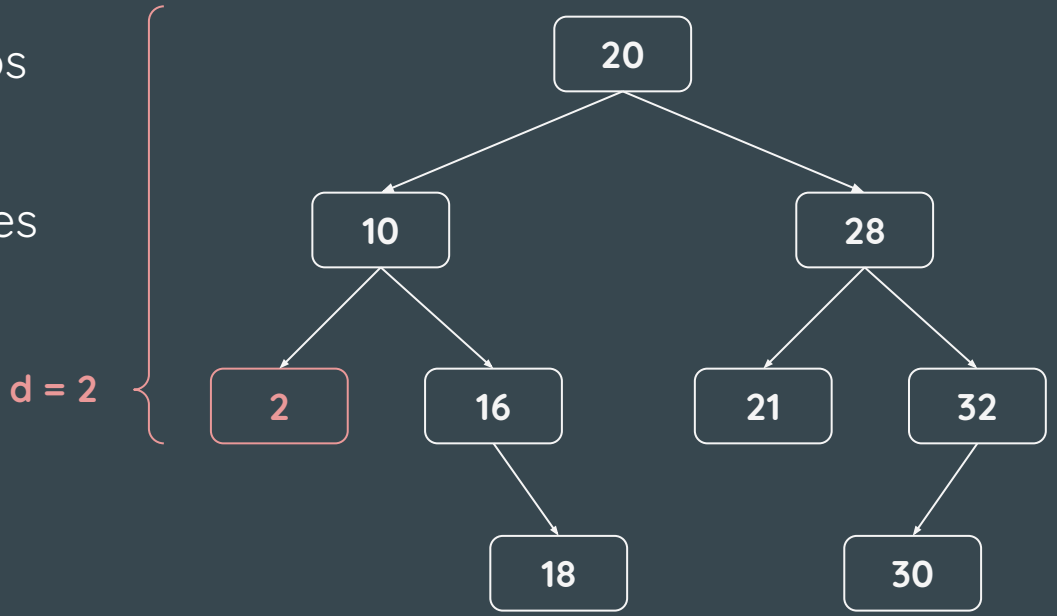
```
entradas: valor, nodo (por defecto, None)

si árbol vacio:
  crear nodo
  raiz = nodo
sino:
  si actual == None:
    actual = raiz

  si valor > actual.elemento y actual.derecha == None:
    crear nodo
    actual.derecha = nodo
  sino, si valor > actual.elemento y actual.derecha != None:
    agregar(valor, actual.derecha)
  sino, si valor < actual.elemento y actual.izquierda == None:
    crear nodo
    actual.izquierda = nodo
  sino, si valor < actual.elemento y actual.izquierda == None:
    agregar(valor, actual.izquierda)
```

Agregar nodos: código

```
def agregar(self, valor, actual=None):
  if self.vacio():
    nodo = Nodo(valor)
    self._raiz = nodo
    self._tamano += 1
  else:
    if actual == None:
      actual = self._raiz
    if valor > actual._elemento and actual._derecha == None:
      nodo = Nodo(valor)
      nodo._padre = actual
      actual._derecha = nodo
      self._tamano += 1
    elif valor > actual._elemento and actual._derecha != None:
      self.agregar(valor, actual._derecha)
    elif valor < actual._elemento and actual._izquierda == None:
      nodo = Nodo(valor)
      nodo._padre = actual
      actual._izquierda = nodo
      self._tamano += 1
    elif valor < actual._elemento and actual._izquierda != None:
      self.agregar(valor, actual._izquierda)
```

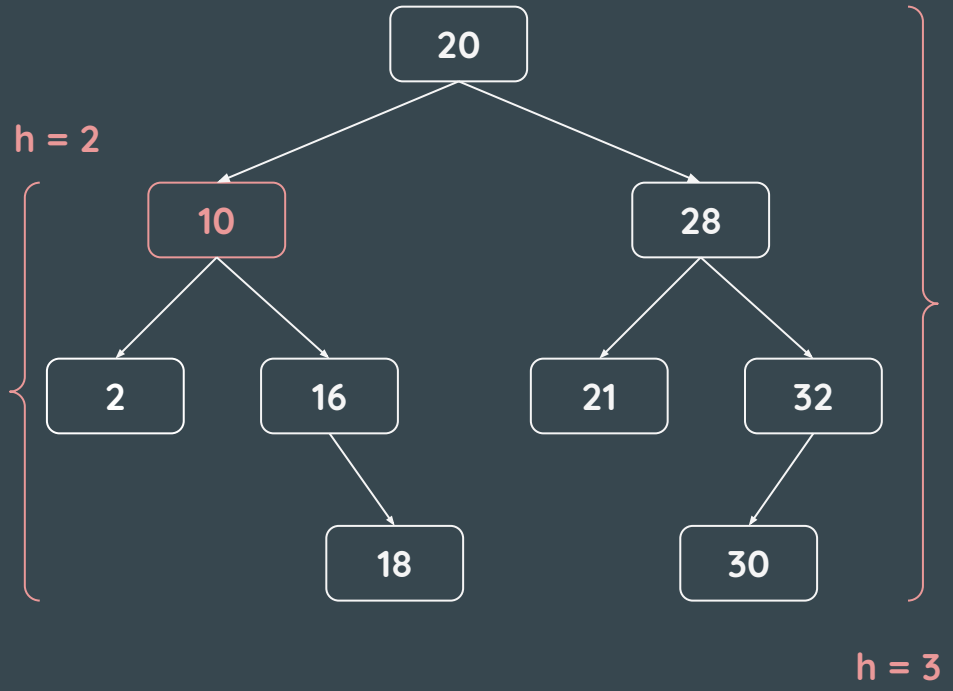
Profundidad y altura (I)

- Se denomina **profundidad** (***depth***) a la cantidad de nodos padres que tiene un nodo.
- La profundidad del nodo raíz es siempre 0.



Profundidad y altura (II)

- La **altura** de un nodo es la **máxima profundidad** alcanzada por sus hojas hasta llegar a dicho nodo.
- La altura de las hojas es siempre 0.
- La altura total de un árbol es la altura de su nodo raíz.



Profundidad y altura: pseudocódigo

```
funcion profundidad:
  si nodo es raiz:
    devolver 0
  sino:
    devolver 1 + profundidad(nodo.padre)

funcion altura:
  si nodo es hoja:
    devolver 0
  si nodo.izquierda == None # no hay subárbol izquierdo, ver el derecho
    devolver 1 + altura(nodo.derecha)
  si nodo.derecha == None # no hay subárbol derecho, ver el izquierdo
    devolver 1 + altura(nodo.derecha)
  si altura(nodo.izquierdo) > altura(nodo.derecha):
    devolver 1 + altura(nodo.izquierda)
  sino:
    devolver 1 + altura(nodo.derecha)
```

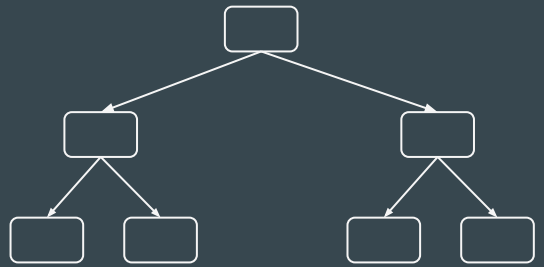
Profundidad y altura: código

```
class ArbolBinario:
  def profundidad(self, actual):
    if self.es_raiz(actual):
      return 0
    else:
      return 1 + profundidad(actual._padre)

  def altura(self, actual):
    if self.es_hoja(actual):
      return 0
    elif actual._izquierda == None:
      return 1 + self.altura(actual._derecha)
    elif actual._derecha == None:
      return 1 + self.altura(actual._izquierda)
    elif self.altura(actual._izquierda) > self.altura(actual._derecha):
      return 1 + self.altura(actual._izquierda)
    else:
      return 1 + self.altura(actual._derecha)
```

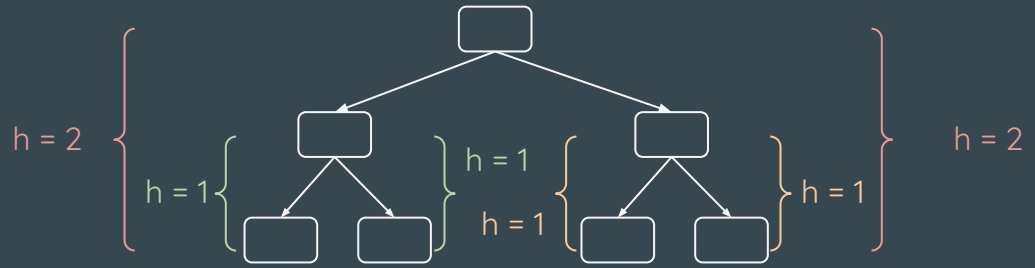
Balanceo de árboles

- A medida que se agregan nodos, un subárbol puede crecer más que el otro.
- Si un subárbol es mucho más grande que el otro, el recorrido de la estructura de datos pierde eficiencia.
- Es por ello que se busca que los árboles estén **balanceados**.
- Un árbol se considera balanceado si la altura del subárbol de la izquierda y el de la derecha no difieren en más de uno.

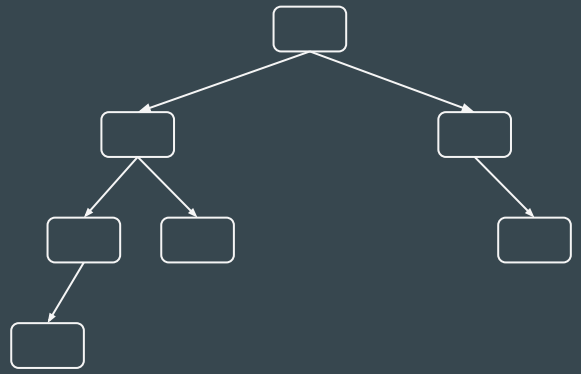
Balanceo de árboles



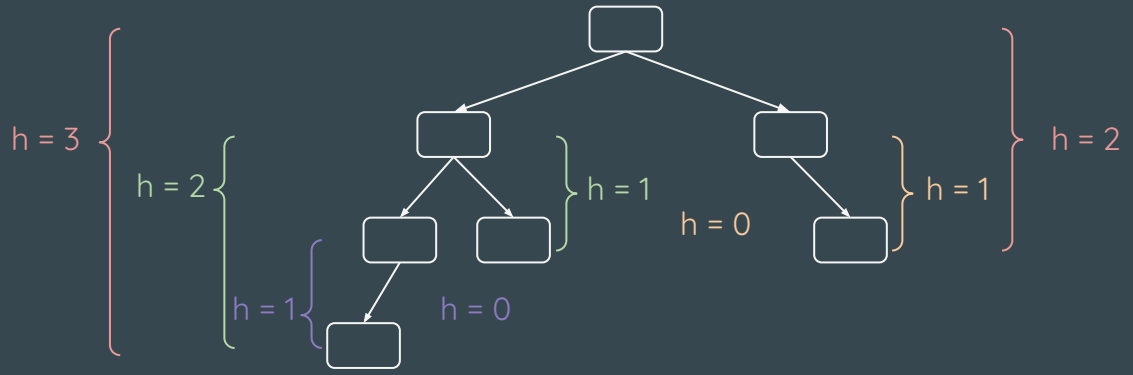
Balanceo de árboles



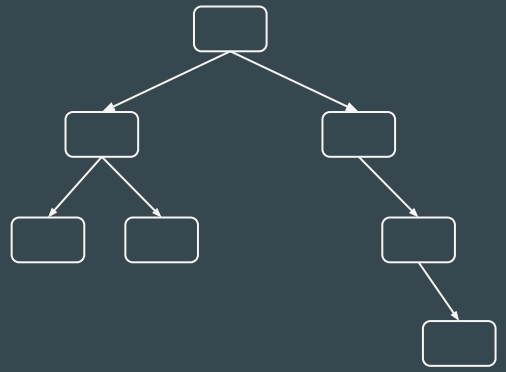
Balanceo de árboles



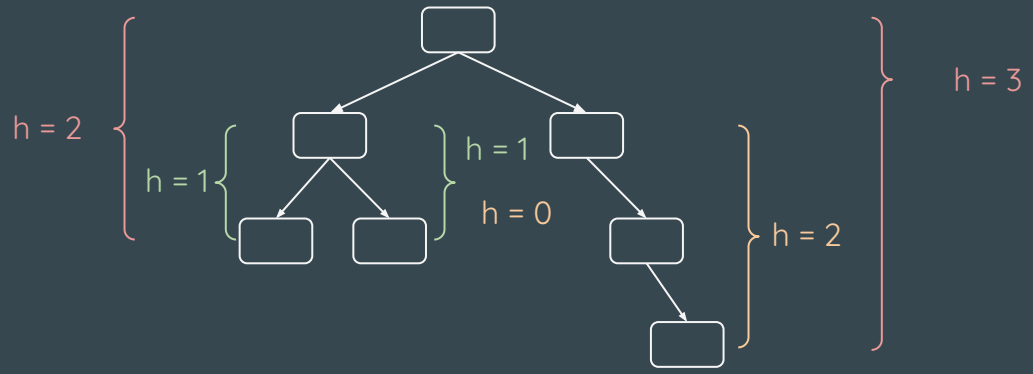
Balanceo de árboles



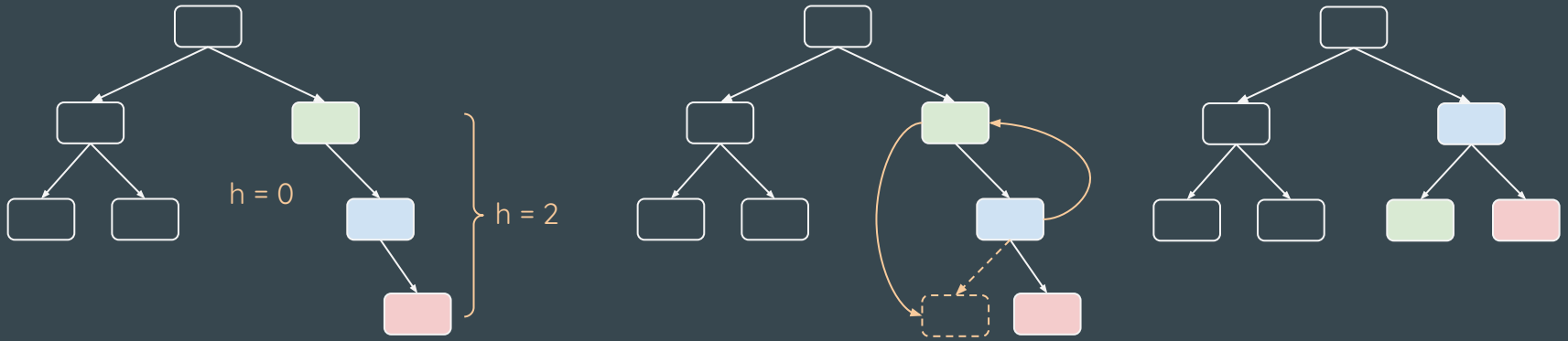
Balanceo de árboles



Balanceo de árboles



Balanceo de árboles



Balanceo de árboles: pseudocódigo

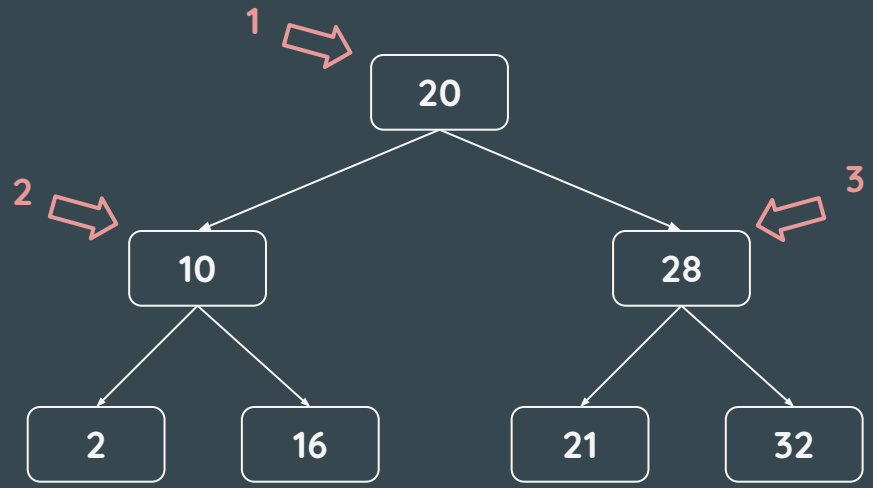
```
# Balanceado del subárbol de la derecha
si altura(actual.derecha) > altura(actual.izquierda) + 1:
  actual.derecha.padre = actual.padre
  actual.derecha.izquierda = actual
  actual.padre = actual.derecha
  actual.padre.derecha = actual.derecha

sino, si altura(actual.izquierda) > altura(actual.derecha) + 1:
  actual.izquierda.padre = actual.padre
  actual.izquierda.derecha = actual
  actual.padre = actual.izquierda
  actual.padre.izquierda = actual.izquierda
```

Recorrido de árboles

- Existen 2 formas básicas de recorrer todos los nodos de un árbol: **por anchura** y **por profundidad**.
- Dentro de los recorridos por profundidad, se puede definir 3 recorridos:
    - Preorder.
    - Inorder.
    - Postorder.
- La principal diferencia entre estos recorridos es cuándo se visita el nodo actual

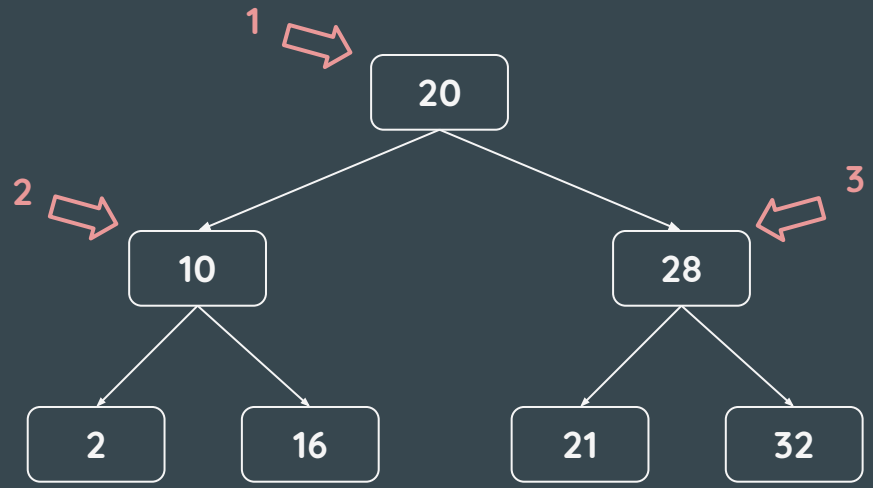
Recorrido preorder

1) Se visita el nodo.
2) Se recorre el subárbol de la izquierda.
3) Se recorre el subárbol de la derecha.



Recorrido preorder

1) Se visita el nodo.
2) Se recorre el subárbol de la izquierda.
3) Se recorre el subárbol de la derecha.



Resultado: 20; 10; 2; 16; 28; 21; 32

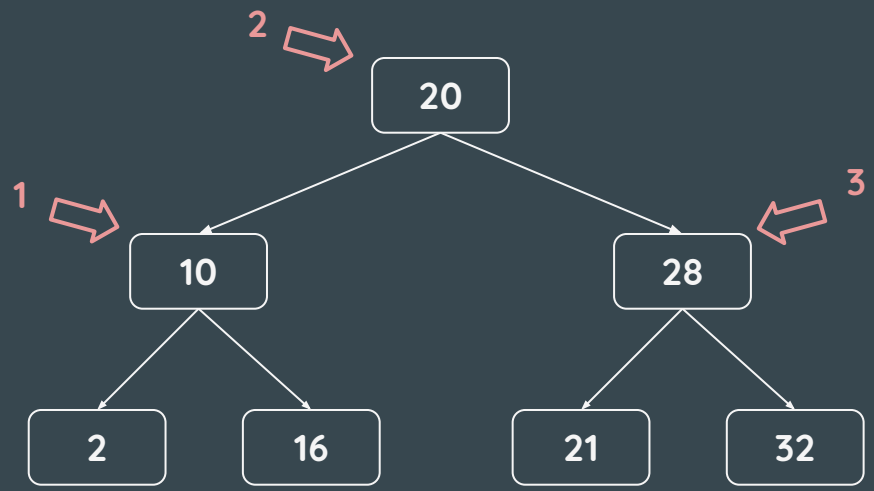
Recorrido preorder

1) Se visita el nodo.
2) Se recorre el subárbol de la izquierda.
3) Se recorre el subárbol de la derecha.

```
def preorder(self, actual = None):
  if actual == None:
    actual = self._raiz

  print(actual._elemento)
  if actual._izquierda != None:
    self.preorder(actual._izquierda)
  if actual._derecha != None:
     self.preorder(actual._derecha)
```

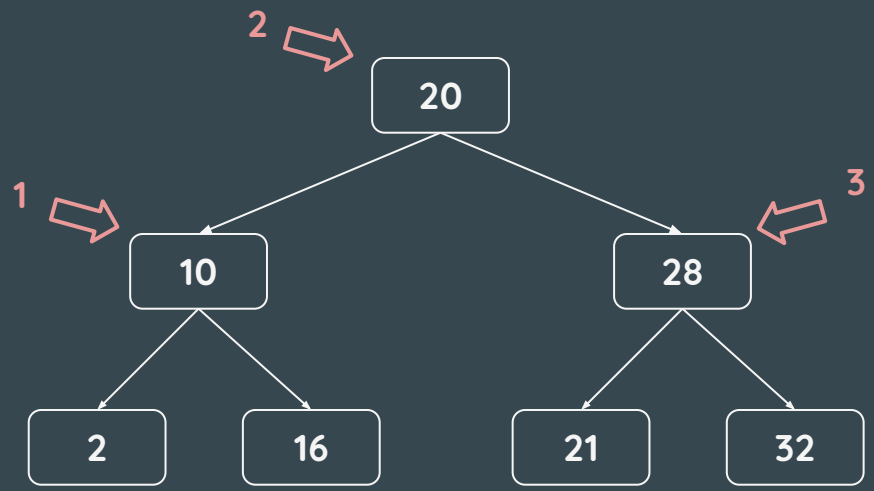
Recorrido inorder

1) Se recorre el subárbol de la izquierda.
2) Se visita el nodo.
3) Se recorre el subárbol de la derecha.



Recorrido inorder

1) Se recorre el subárbol de la izquierda.
2) Se visita el nodo.
3) Se recorre el subárbol de la derecha.



Resultado: 2; 10; 16; 20; 21; 28; 32

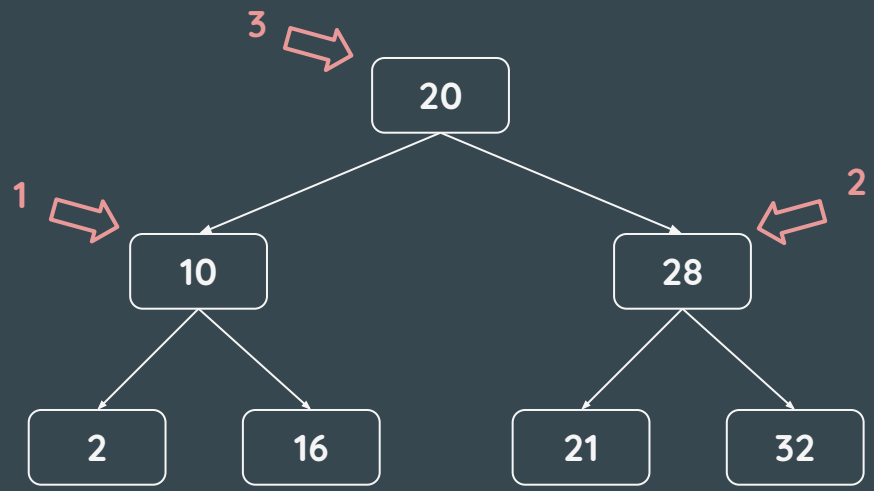
Recorrido inorder

1) Se recorre el subárbol de la izquierda.
2) Se visita el nodo.
3) Se recorre el subárbol de la derecha.

```
def inorder(self, actual = None):
  if actual == None:
    actual = self._raiz

  if actual._izquierda != None:
    self.inorder(actual._izquierda)
  print(actual._element)
  if actual._derecha != None:
     self.inorder(actual._derecha)
```

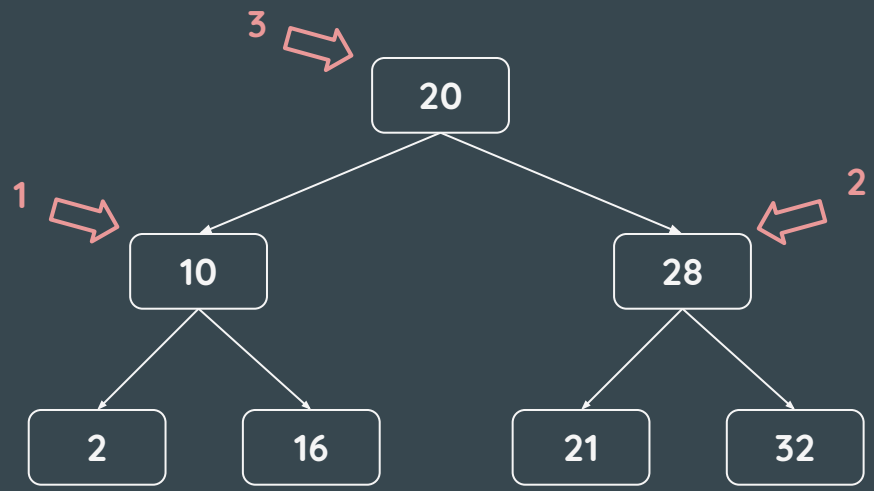
Recorrido postorder

1) Se recorre el subárbol de la izquierda.
2) Se recorre el subárbol de la derecha.
3) Se visita el nodo.



Recorrido postorder

1) Se recorre el subárbol de la izquierda.
2) Se recorre el subárbol de la derecha.
3) Se visita el nodo.



Resultado: 2; 16; 10; 21; 32; 28; 20

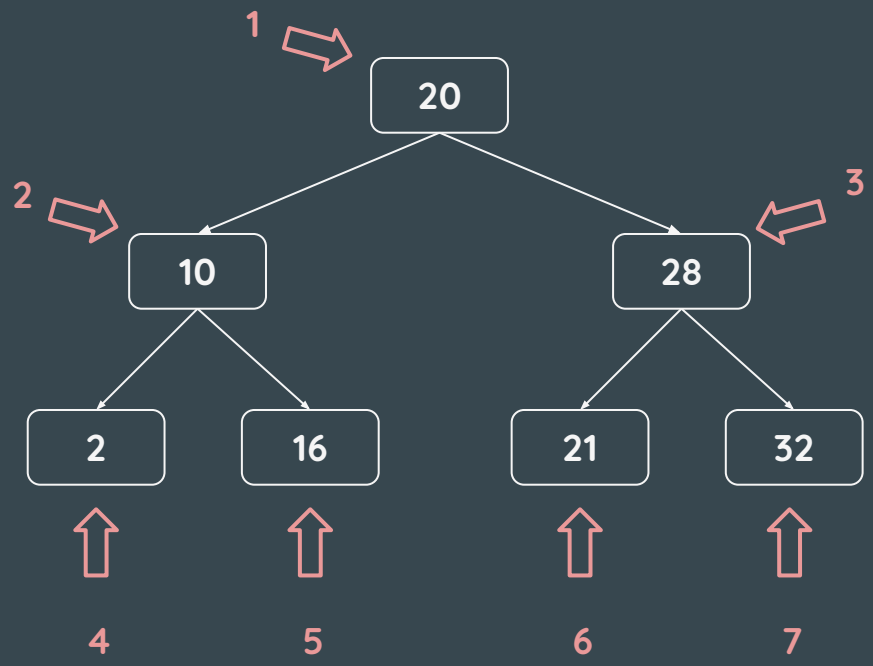
Recorrido postorder

1) Se recorre el subárbol de la izquierda.
2) Se recorre el subárbol de la derecha.
3) Se visita el nodo.

```
def postorder(self, actual = None):
  if actual == None:
    actual = self._raiz

  if actual._izquierda != None:
    self.postorder(actual._izquierda)
  if actual._derecha != None:
     self.postorder(actual._derecha)
  print(actual._elemento)
```

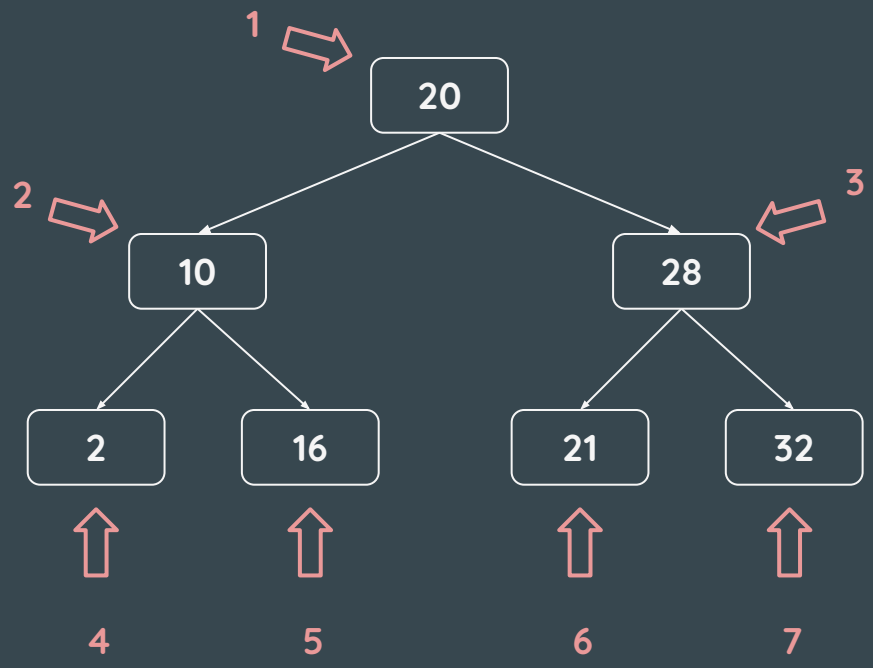
Recorrido por anchura

1) Se inicia una cola de visitas con el nodo raíz.
2) Se agregan todos los nodos hijos.
3) Se quita de la cola.
4) Se visita el siguiente nodo en la cola.



Recorrido por anchura

1) Se inicia una cola de visitas con el nodo raíz.
2) Se agregan todos los nodos hijos.
3) Se quita de la cola.
4) Se visita el siguiente nodo en la cola.



Resultado: 20; 10; 28; 2; 16; 21; 32

Recorrido por anchura

1) Se inicia una cola de visitas con el nodo raíz.
2) Se agregan todos los nodos hijos.
3) Se quita de la cola.
4) Se visita el siguiente nodo en la cola.

```
def por_anchura(self):
  cola = []
  cola.append(self._raiz)
  while len(cola) > 0:
    actual = cola.pop(0)
    print(actual._elemento)
    if actual._izquierda != None:
      cola.append(actual._izquierda)
    if actual._derecha != None:
      cola.append(actual._derecha)
```

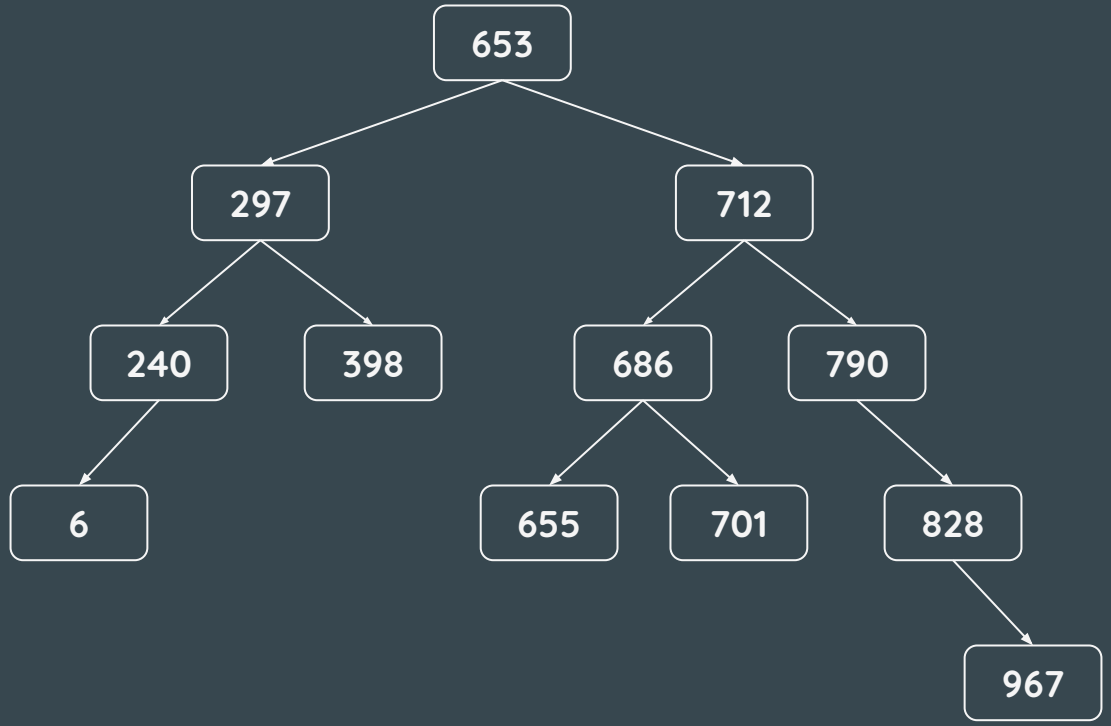
Ejemplo

Dada la siguiente lista de números:

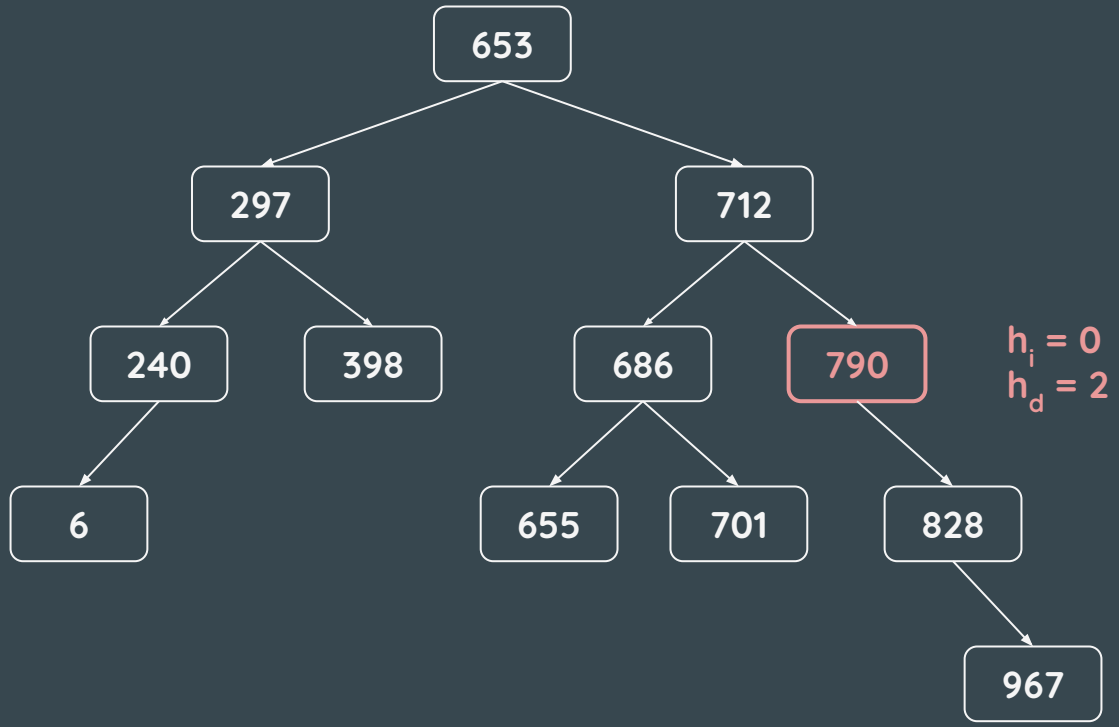
653, 712, 790, 297, 686, 398, 701, 240, 6, 655, 828, 967

1. Construya un árbol, respetando el orden en el que aparecen los números
2. Determine si está balanceado
3. Recorralo usando las estrategias preorder, inorder y por anchura

Ejemplo: construcción



Ejemplo: balanceado?



Ejemplo: recorridos

- Preorder:

  653; 297; 240; 6; 398; 712; 686; 655; 701; 790; 828; 967

- Inorder:

  6; 240; 297; 398; 653; 655; 686; 701; 712; 790; 828; 967

- Por anchura:

  653; 297; 712; 240; 398; 686; 790; 6; 655; 701; 828; 967

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 8.