Licenciatura en Ciencia de Datos

Algoritmos II

¿Por qué árboles?

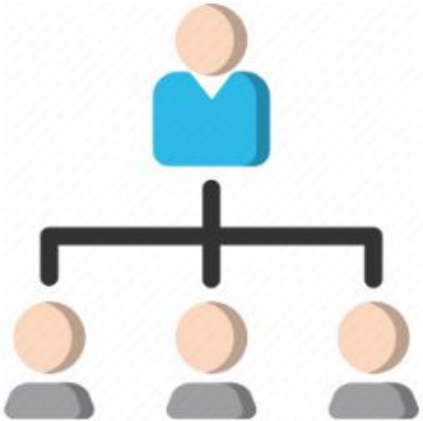
-Las listas solo nos proveen la capacidad de representar **relaciones** restringidas a **una sola dimensión.**

- A lo sumo, podemos usar estructuras de **pila (LIFO)** o **cola (FIFO)** para dar **prioridad,** pero seguimos en una única dimensión.

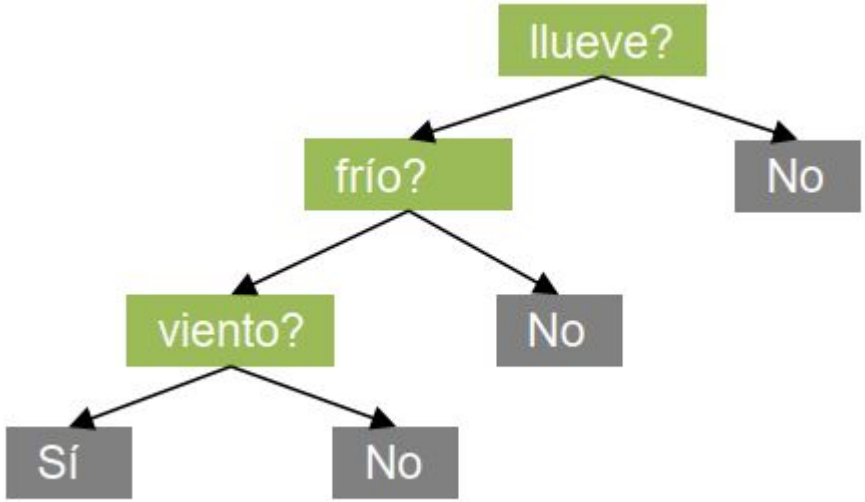
CUANDO NECESITAMOS REPRESENTAR UNA **RELACIÓN JERÁRQUICA** USAMOS **ÁRBOLES!**

Ejemplos

Organigrama

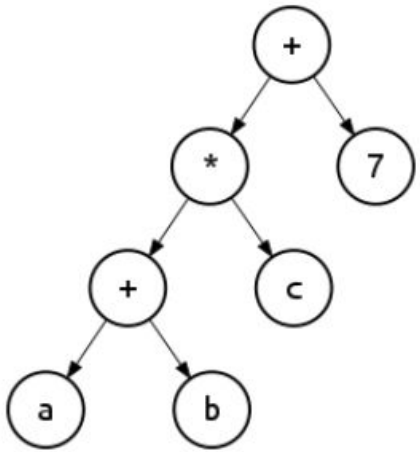


Árbol de decisión

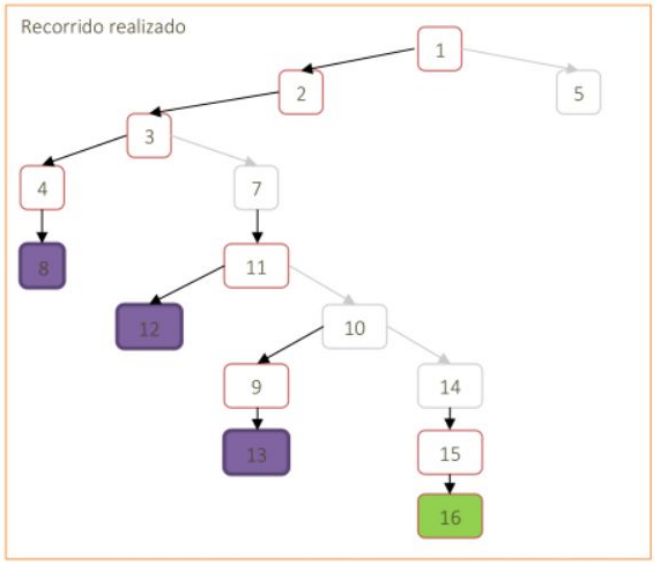


Ejemplos

Árbol de expresión aritmética



Árbol de soluciones



Definiciones

"*El árbol es una **estructura** que permite representar un **orden jerárquico** de una **colección de elementos**"*

Formalmente:

Un **árbol T** es un conjunto **finito** de **cero o más nodos (v1, v2, …, vn)** donde:

• existe un **nodo** especial llamado nodo **raíz**.

• los **nodos restantes** se **particionan** en **m ≥ 0 conjuntos disjuntos T1, T2, …, Tm**, donde **cada Ti** es un **árbol** con el **nodo raíz** como **padre.**

Definiciones

- Un **bosque** es un **conjunto de árboles T1, T2, …, Tn.** (conjunto de dos o más árboles)

-Un árbol se **compone** de:

- **nodos o vértices**
- **arcos o aristas**

Definiciones

-Los árboles tienen **aristas** con dirección **descendente.** Representa la relación **“es padre de”.**

- **Nodo Raíz:** es tanto **padre** como **ancestro** de **todos** los nodos del árbol.

-**Nodo ancestro: precede** a un nodo dado en el árbol y comparte un camino común de **conexión con la raíz.**

-**Nodo padre:** tiene **uno o más nodos hijos** conectados a él mediante aristas dirigidas = tiene otros nodos descendientes que se originan desde él.

-**Nodo hijo:** está **directamente** conectado a un **nodo padre** a través de una arista dirigida descendente.

Definiciones

TODO NODO TIENE UN ÚNICO PADRE (EXCEPTO LA RAÍZ) AUNQUE UN NODO PADRE PUEDE TENER MÚLTIPLES NODOS HIJOS!

Definiciones

-**Nodos hoja: no** tienen **hijos**.

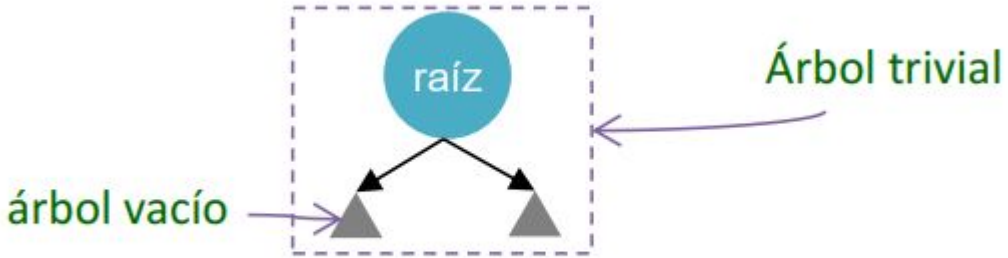
-Nodos hermanos: aquellos que **comparten el mismo padre**.

-**Nodos primos: comparten el mismo nivel** en la jerarquía del árbol, **pero no** tienen el **mismo padre**.

Definiciones

-**Árbol vacío: sin nodos**.

-**Árbol trivial:** sólo tiene **un nodo**, el raíz. **No suele tener aplicación práctica**; sirve para tratar situaciones particulares al momento de **construir un árbol o modificarlo**.



-Aridad: cantidad **máxima de hijos** que puede tener un árbol.

Ej: **binario**: máximo dos hijos por nodo; **ternario**: máximo tres hijos por nodo; **n-ario**: sin límite.

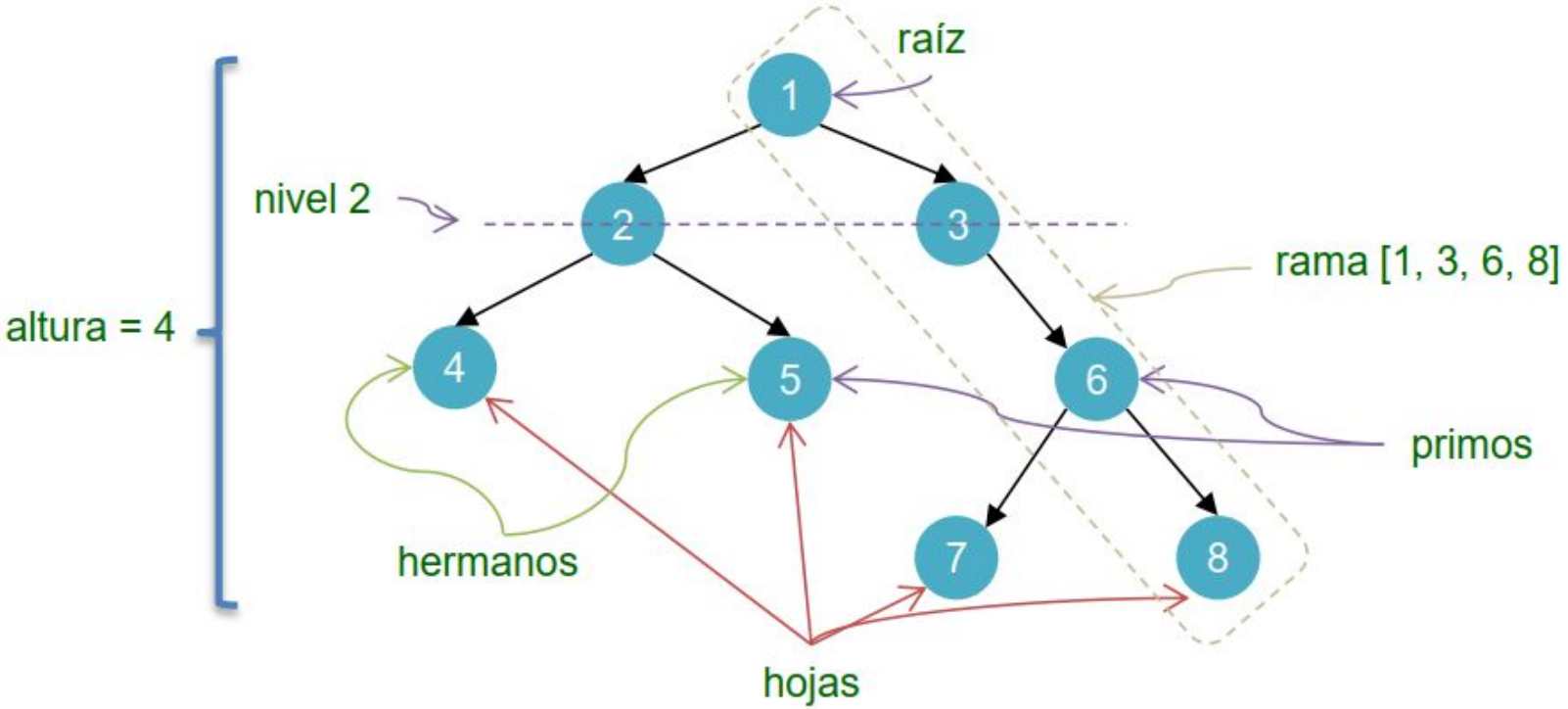
Definiciones

-**Niveles: distancia** de un **nodo** desde la **raíz** (la raíz está en el **nivel 1**).

-**Rama: secuencia de nodos** que **conecta** un **nodo cualquiera** del árbol con la **raíz.**

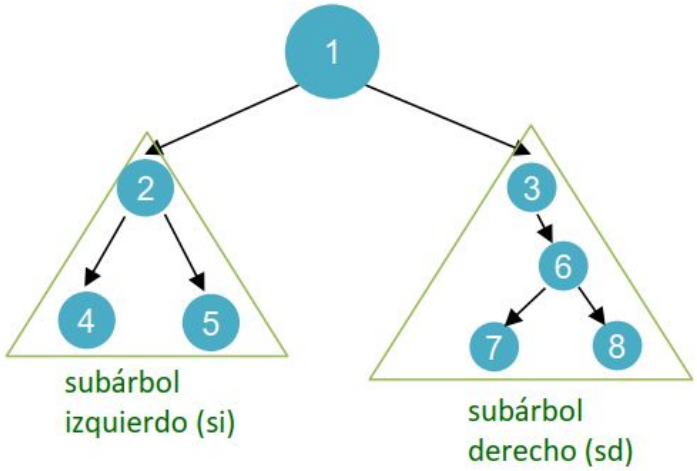
-**Altura:** se define a partir de la **cantidad de niveles** del árbol, siendo la **longitud de la rama más larga** del árbol.

Gráficamente



Definiciones

-**Subárbol:** una **parte** del árbol que consta de un **nodo y todos sus descendientes, incluido ese nodo**. Es un **árbol completo,** ya que tiene su **propia raíz** y su **propia estructura jerárquica**, pero está **contenido dentro del árbol más grande**.

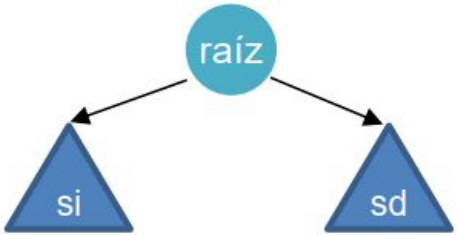


Definiciones

LOS ÁRBOLES SON ESTRUCTURAS RECURSIVAS, DONDE UN NODO PUEDE TENER VARIOS NODOS HIJOS QUE A SU VEZ SON ÁRBOLES

Implementaciones simples: Árbol Binario

ESTRUCTURA CON RECURSIÓN DIRECTA:



La **ETIQUETA** (dato) de un nodo debería poder representar a cada nodo de forma **simple y única.**

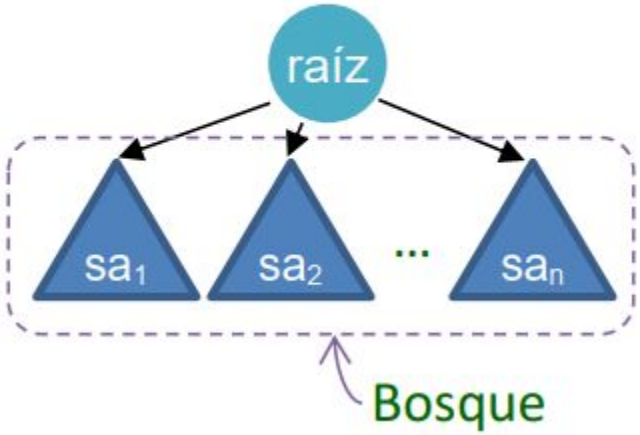
Implementaciones árbol binario

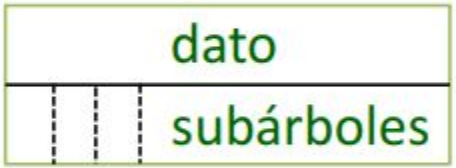
```
from typing import Generic, Optional, TypeVar

T = TypeVar('T')

class ArbolBin(Generic[T]):
    def __init__(self, dato: T):
        self.dato: T = dato
        self.si: Optional[ArbolBin[T]] = None  # subárbol izquierdo
        self.sd: Optional[ArbolBin[T]] = None  # subárbol derercho
```

Si bien es **suficiente** para definir un árbol binario, es **complicada** para representar la abstracción de un **árbol vacío**.

Implementaciones Árbol N-Ario

Estructura con recursión mutua





Implementaciones Árbol N-Ario. Consideraciones

-A partir de un **cierto nodo** existen **ninguno o más subárboles**, sin restricción en la cantidad.

-Surge el concepto de **bosque** para representar el **conjunto de árboles**.

-Un **bosque** se puede implementar como una **Lista(Arbol(a))**

-En esta implementación **no existe el concepto de árbol vacío.**

Implementación Árbol N-Ario

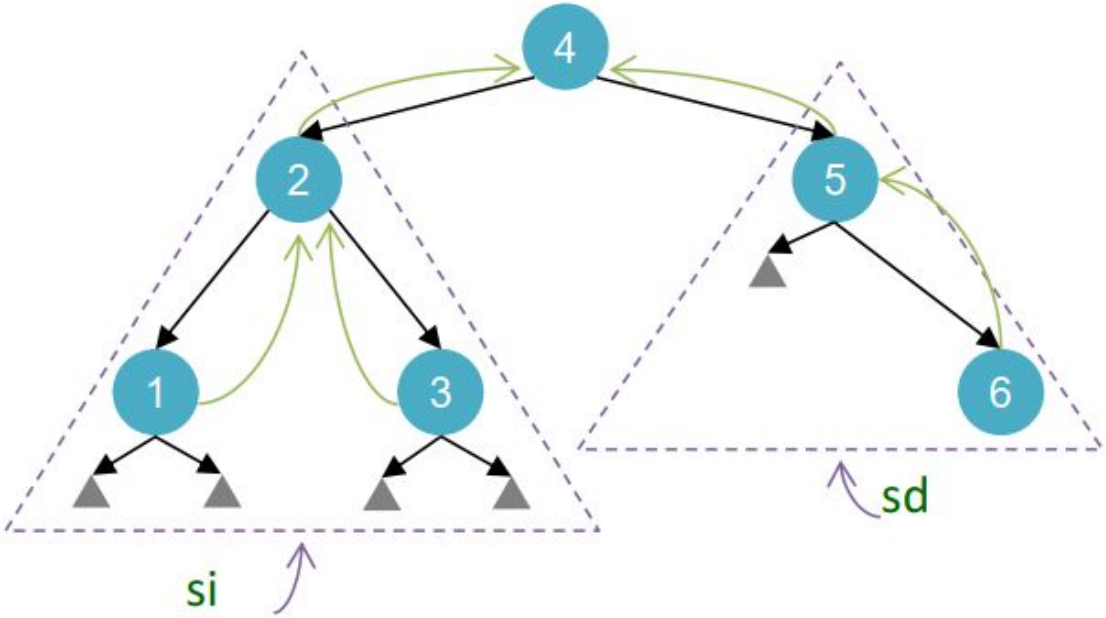
```
from typing import Generic, TypeVar

T = TypeVar('T')

class Arbol(Generic[T]):
    def __init__(self, dato: T):
        self.dato: T = dato
        self.subarboles: Bosque[T] = Bosque()

class Bosque(Generic[T]):
    def __init__(self):
        self.arboles: list[Arbol[T]] = []
```

Estructuras alternativas para árbol binario

-Podemos encontrar algunas **implementaciones particulares** para los árboles.

-Quien implemente seleccionará la **más adecuada para resolver su problema**, ya que cada variación tendrá sus ventajas y desventajas.

Árbol binario con acceso al nodo padre



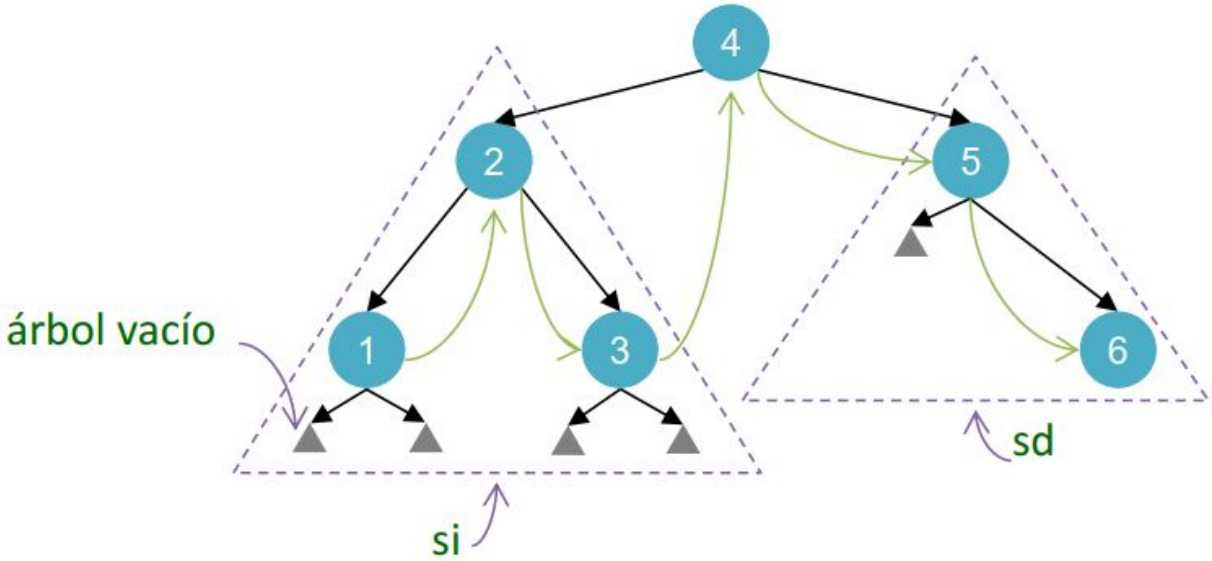
Implementación

```
class ArbolBin(Generic[T]):
    def __init__(self, dato: T, padre: Optional['ArbolBin'] = None):
        self.dato: T = dato
        self.si: Optional[ArbolBin] = None  # subárbol izquierdo
        self.sd: Optional[ArbolBin] = None  # subárbol derecho
        self.padre: Optional[ArbolBin] = padre  # nodo padre (None para la raíz)

# Ejemplo de uso:
raiz = ArbolBin('A')
b = ArbolBin('B', padre=raiz)
c = ArbolBin('C', padre=raiz)
d = ArbolBin('D', padre=b)
```

Esta versión **NO INCLUYE** el concepto de **árbol vacío**!

Incorporación de un campo que apunte al nodo siguiente

-Se realiza de acuerdo a la **estrategia de recorrido del árbol**, en este caso, inorder (lo vamos a ver más adelante).



Implementación

```
from typing import Optional, TypeVar

T = TypeVar('T')

class ArbolBin(Generic[T]):
    def __init__(self, dato: T):
        self.dato: T = dato
        self.si: Optional[ArbolBin] = None  # subárbol izquierdo
        self.sd: Optional[ArbolBin] = None  # subárbol derecho
        self.sig: Optional[ArbolBin] = None  # siguiente nodo en el recorrido

# Creamos algunos nodos de ejemplo
raiz = ArbolBin('A')
b = ArbolBin('B')
c = ArbolBin('C')
d = ArbolBin('D')
# Establecemos las relaciones de los nodos
raiz.si = b
raiz.sd = c
b.sig = c
```

TAD Árbol Binario: Nodo: abrir repo!

ESTRUCTURA CON RECURSIÓN MUTUA: nos permite representar el concepto de **árbol vacío**

```
class NodoAB(Generic[T]):
    def __init__(self, dato: T, si: "Optional[ArbolBinario[T]]" = None, sd: "Optional[ArbolBinario[T]]" = None):
        # Constructor que inicializa un nodo con un dato y opcionalmente sus subárboles izquierdo y derecho.
        self.dato = dato  # Almacena el dato del nodo.
        self.si: ArbolBinario[T] = ArbolBinario() if si is None else si
self.sd: ArbolBinario[T] = ArbolBinario() if sd is None else sd
    def __str__(self):
        return self.dato  # Retorna el dato del nodo como una cadena.
```

TAD Árbol Binario

```
class ArbolBinario(Generic[T]):  # Define la clase ArbolBinario que es genérica respecto a T.
    def __init__(self):
        self.raiz: Optional[NodoAB[T]] = None  # Inicializa el árbol con la raíz como None (árbol vacío).Notar que no se le pasa ningún parámetro a init!

    class _Decoradores:  # Define una clase interna para contener decoradores.
        @classmethod
        def valida_es_vacio(cls, f: Callable[..., Any]) -> Callable[..., Any]:
            # Decorador de método que verifica si el árbol está vacío antes de ejecutar la función.
            @wraps(f)
            def wrapper(self, *args: Any, **kwargs: Any) -> Any:
                if self.es_vacio():  # Verifica si el árbol está vacío.
                    raise TypeError('Arbol Vacio')  # Lanza una excepción si el árbol está vacío.
                return f(self, *args, **kwargs)  # Llama a la función decorada si el árbol no está vacío.
            return wrapper
```

TAD Árbol Binario

```
    @staticmethod          # Método estático para crear un nuevo árbol con un nodo raíz dado.
    def crear_nodo(dato: T, si: "Optional[ArbolBinario[T]]" = None, sd: "Optional[ArbolBinario[T]]" = None)
-> "ArbolBinario[T]":
        t = ArbolBinario()  # Crea un nuevo árbol binario.
        t.raiz = NodoAB(dato, si, sd)  # Asigna un nuevo nodo como la raíz del árbol.
        return t  # Retorna el árbol creado.

    def es_vacio(self) -> bool:
        return self.raiz is None

    def es_hoja(self) -> bool:
        return not self.es_vacio() and self.si().es_vacio() and self.sd().es_vacio()

    @_Decoradores.valida_es_vacio
    def si(self) -> "ArbolBinario[T]":# Retorna el subárbol izquierdo, verificando que el árbol no esté
vacío.
        assert self.raiz is not None  # Asegura que la raíz no sea None.
        return self.raiz.si  # Retorna el subárbol izquierdo.
```

TAD Árbol binario

```
    @_Decoradores.valida_es_vacio
    def sd(self) -> "ArbolBinario[T]":
        assert self.raiz is not None  # Asegura que la raíz no sea None.
        return self.raiz.sd  # Retorna el subárbol derecho.

    @_Decoradores.valida_es_vacio
    def dato(self) -> T:  # Retorna el valor del nodo raíz.
        return self.raiz.valor

    @_Decoradores.valida_es_vacio
    def insertar_si(self, si: "ArbolBinario[T]"):  # Inserta un subárbol izquierdo.
        self.raiz.si = si  # Establece el subárbol izquierdo del nodo raíz.

    @_Decoradores.valida_es_vacio
    def insertar_sd(self, sd: "ArbolBinario[T]"):  # Inserta un subárbol derecho.
        self.raiz.sd = sd  # Establece el subárbol derecho del nodo raíz.

     def set_raiz(self, nodo: NodoAB[T]):
        self.raiz = nodo
```

TAD Árbol Binario: Altura

```
    def altura(self) -> int:  # Calcula la altura del árbol.
        if self.es_vacio():  # Si el árbol está vacío, su altura es 0.
            return 0
        else:
            return 1 + max(self.si().altura(), self.sd().altura())  #
Altura es 1 más la altura máxima entre los subárboles.
```

TAD Árbol binario: longitud

```
    def __len__(self) -> int:  # Calcula el número total de nodos en el árbol.
        if self.es_vacio():  # Si el árbol está vacío, su tamaño es 0.
            return 0
        else:
            return 1 + len(self.si()) + len(self.sd())  # Suma 1 (nodo actual) y los tamaños de los subárboles
```

TAD Árbol binario: str

```
  def __str__(self):  # Convierte el árbol en una representación en cadena para impresión.
        def recorrer(t: ArbolBinario[T], nivel: int) -> str:  # Función interna para recorrer el árbol y formatear la salida.
            tab = '.' * 4 * nivel  # Crea una indentación basada en el nivel del nodo.
            if t.es_vacio():  # Si el árbol está vacío, indica con 'AV' (Árbol Vacío).
                return tab + 'AV\n'
            else:
                tab += str(t.dato()) + '\n'  # Agrega el dato del nodo actual.
                tab += recorrer(t.si(), nivel + 1)  # Agrega la representación del subárbol izquierdo.
                tab += recorrer(t.sd(), nivel + 1)  # Agrega la representación del subárbol derecho.
                return tab

        return recorrer(self,0)
```

Ejercicio 1

Crear dentro del TAD ArbolBinario una función recursiva nivel, que dado un valor, retorne en qué nivel se encuentra en el nodo del árbol. Si el valor no se encontrara en el árbol, retornar un valor superior a la altura del árbol.

Ejercicio 2

Desarrollar dentro del TAD ArbolBinario una función recursiva copy, que devuelva la deep copy de un árbol (el prototipo se encuentra en el template)