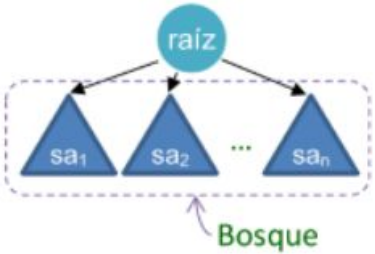
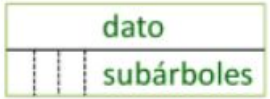
Licenciatura en Ciencia de Datos

Algoritmos II

Definición





-Tipo de árbol que permite que **un nodo** tenga una **cantidad variable e indefinida de subárboles**.

-Podemos tener **ninguno o varios subárboles** descendiendo de un nodo.

-**Bosque**:conjunto de **subárboles**.Forma simple de implementarlo: **lista**.

-Una **hoja** tiene la **lista de subárboles vacía**.

- **NO REPRESENTAMOS EL ÁRBOL VACÍO!!!**

Implementación

```
from typing import Any, Generic, TypeVar

T = TypeVar('T')

class ArbolN(Generic[T]):
    def __init__(self, dato: T):
        self._dato: T = dato
        self._subarboles: list[ArbolN[T]] = []
```

Dado que no necesitamos representar un árbol vacío, podemos definir una estructura con **recursión directa múltiple** (tipo de recursión en la que una función/estructura **se llama a sí misma más de una vez dentro de su propio cuerpo**)

Implementación

```
from typing import Any, Generic, TypeVar

T = TypeVar('T')

class ArbolN(Generic[T]):
    def __init__(self, dato: T):
        self._dato: T = dato
        self._subarboles: list[ArbolN[T]] = []
```

Dado que no necesitamos representar un árbol vacío, podemos definir una estructura con **recursión directa múltiple** (tipo de recursión en la que una función/estructura **se llama a sí misma más de una vez dentro de su propio cuerpo**)

Setter y getters

```python
@property
def dato(self) -> T:
    return self._dato

@dato.setter
def dato(self, valor: T):
    self._dato = valor

@property
def subarboles(self) -> list[ArbolN[T]]:
    return self._subarboles

@subarboles.setter
def subarboles(self, subarboles: list[ArbolN[T]]):
    self._subarboles = subarboles
```

Operaciones básicas

```
def insertar_subarbol(self, subarbol: ArbolN[T]):
    self.subarboles.append(subarbol)

def es_hoja(self) -> bool:
    return self.subarboles == []
```

Altura: con recursión múltiple a través de list comprehensions

```
def altura(self) -> int:
    if self.es_hoja():
        return 1
    else:
        return 1 + max([subarbol.altura() for subarbol in self.subarboles])
```

Altura: con un bucle

```
def altura(self) -> int:
        altura_actual = 0
        for subarbol in self.subarboles:
            altura_actual = max(altura_actual, subarbol.altura())
        return altura_actual + 1
```

Se visualiza más fácil

Altura con recursión múltiple

```
def altura(self) -> int:
    def altura_n(bosque: list[ArbolN[T]]) -> int:
        if not bosque:
            return 0
        else:
            return max(bosque[0].altura(), altura_n(bosque[1:]))

    return 1 + altura_n(self.subarboles)
```

Forma de **resolver una operación la** cual **invoque a otra operación** que **reciba la lista** de sus **subárboles**. **Esta última** operación invocará luego nuevamente **la original para cada uno** de los **subárboles** de la lista recibida.

Estrategias de recorrido: DFS: Preorder. Versión funcional

Machete: **Primero** visitamos el **nodo raíz** y luego visitamos con el **mismo orden** los **subárboles de izquierda a derecha** (o viceversa).

```
def preorder(self) -> list[T]:
    return reduce(
        lambda recorrido, subarbol: recorrido + subarbol.preorder(),
        self.subarboles,
        [self.dato]
    )
```

Estrategias de recorrido: DFS: Preorder. Imperativa

```
def preorder(self) -> list[T]:
    recorrido = [self.dato]
    for subarbol in self.subarboles:
        recorrido += subarbol.preorder()
    return recorrido
```

Estrategias de recorrido: DFS: Preorder. Recursión mutua

```
def preorder(self) -> list[T]:
    def preorder_n(bosque: list[ArbolN[T]]) -> list[T]:
        if not bosque:
            return []
        else:
            return bosque[0].preorder() + preorder_n(bosque[1:])

    return [self.dato] + preorder_n(self.subarboles)
```

Ejercicio

Desarrollar las funciones de postorder con las mismas tres estrategias vistas, que devuelva una lista con el orden de los nodos visitados.

Estrategias de recorrido: BFS v1

```
    def bfs(self)->list[T]:
        def recorrer():
            if q:
                actual = q.pop()                    # desencolar árbol visitado
                lista.append(actual.dato)
                for subarbol in actual.subarboles:  # para cada subárbol
                    q.insert(0, subarbol)           # encolar subárbol
                recorrer()

        q: list[ArbolN[T]] = [self]                 # encolar raíz
        lista:list[T] = []
        recorrer()
        return lista
```

Ejercicio

Implementar la función **nivel**, que dado un valor contenido en el árbol, devuelva el nivel del mismo. Debe contemplar el caso en el que el valor no se encuentre en el árbol. (en TAD)

Ejercicio

Desarrollar la función **copy**, que devuelve una copia profunda del árbol actual (en TAD)

Ejercicio

Desarrollar la función **sin_hojas**, que devuelva un nuevo árbol sin las hojas del árbol actual.(En TAD)

Ejercicio

Desarrollar la función recursiva **ramas**, que devuelve una **lista de listas** con todas las **ramas** del árbol n-ario.

def ramas(self) -> list[list[T]]

Ejercicio

Desarrollar una función que devuelva una **lista** con los **antecesores** del **dato** buscado en el árbol, contemplando la situación de que el dato pueda no existir (devolver en este caso una lista vacia)

def antecesores(self,valor:T)->list[T]