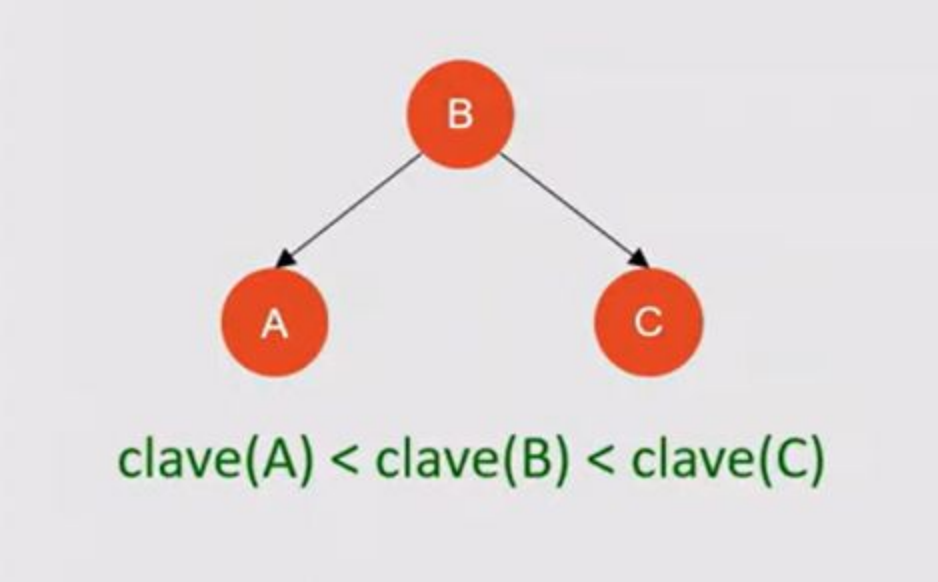
Universidad Nacional de San Martín

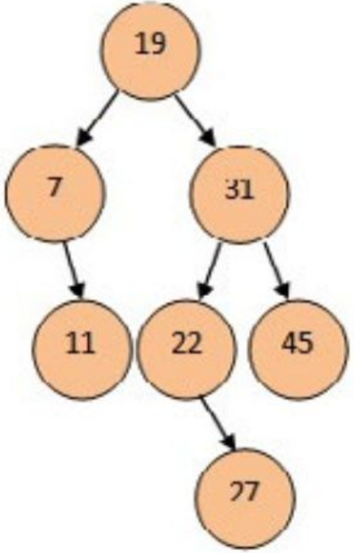
Licenciatura en Ciencia de Datos

Algoritmos II

Árboles Binarios de Búsqueda (ABB)

<u>Def:</u> son árboles binarios en los que los nodos del **subárbol izquierdo** son **menores** que el **padre** y todos los del **subárbol derecho mayores**





Árboles binarios de búsqueda (cont)

- Existe el concepto de **árbol vacío**.

- Los nodos deben estar necesariamente **etiquetados** o contener **una clave.** La condición particular de esta clave es que debe **identificar unívocamente** a los nodos y **respetar una relación de orden**.

- Su recorrido **DFS inorder** devuelve sus **elementos ordenados.**

Características

- **Búsqueda eficiente:** Debido a que los elementos están organizados de manera ordenada y cada nodo tiene referencias a nodos hijo izquierdo y derecho, se puede realizar una **búsqueda binaria** eficiente.

- **Inserción y eliminación eficientes:** especialmente útil en aplicaciones donde se requiere una estructura de datos que admita operaciones de inserción y eliminación rápidas mientras mantiene un cierto orden.

- **Estructuras de datos ordenadas:** mantienen automáticamente sus elementos ordenados. Esto es útil en situaciones donde los datos deben estar organizados de manera ordenada para realizar operaciones como **recorridos ordenados, búsqueda de elementos mínimo y máximo,** etc.

Operaciones

Serán **similares** a las del árbol binario, a diferencia de las que **alteran la estructura**:

- **Inserción**
- **Búsqueda**
- **Eliminación**

En todos estos casos se deberá **verificar que las reglas de orden se mantienen**.

Analizando el código del TAD

```
from typing import TypeVar, Optional, Protocol # Importamos `TypeVar` para definir tipos genéricos, `Optional` para indicar que
una variable puede ser None, y `Protocol` para definir una interfaz que otros tipos pueden implementar.

from arbol_binario import ArbolBinario, NodoAB # Importamos las clases `ArbolBinario` y `NodoAB` que ya implementamos :)

class Comparable(Protocol):
    # Definimos un protocolo `Comparable` que especifica una interfaz para cualquier tipo que pueda ser comparado.
    def __lt__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "menor que" (<). Debe ser implementado por cualquier clase que siga el protocolo.

    def __le__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "menor o igual que" (<=).

    def __gt__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "mayor que" (>).

    def __ge__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "mayor o igual que" (>=).

    def __eq__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "igual que" (==).

    def __ne__(self: 'T', otro: 'T') -> bool: ...
    # Método para la comparación "diferente de" (!=).
```

Analizando el TAD

```
T = TypeVar('T', bound=Comparable)
# Definimos un tipo genérico `T` que está limitado a cualquier tipo que implemente el protocolo `Comparable`.
# Esto garantiza que los valores de tipo `T` se puedan comparar usando operadores como <, >, ==, etc.
#bound limita el tipo genérico a tipos que heredan o implementan una clase o protocolo específico, en este caso Comparable
```

Analizando el TAD

```
class NodoABO(NodoAB[T]):    # Definimos la clase `NodoABO`, que hereda de `NodoAB[T]`. Esta clase representa un nodo en
un árbol binario ordenado.

    def __init__(self, dato: T):
        # Constructor de la clase `NodoABO`.
        super().__init__(dato, ArbolBinarioOrdenado(), ArbolBinarioOrdenado())
        # Llama al constructor de la clase padre `NodoAB`, pasando el dato y dos árboles binarios ordenados vacíos

    def __lt__(self, otro: "NodoABO[T]") -> bool:
        # Definimos el comportamiento de comparación "menor que" (<) entre dos nodos.
        return isinstance(otro, NodoABO) and self.dato < otro.dato

    def __gt__(self, otro: "NodoABO[T]") -> bool:
        # Definimos el comportamiento de comparación "mayor que" (>) entre dos nodos.
        return isinstance(otro, NodoABO) and self.dato > otro.dato


    def __eq__(self, otro: "NodoABO[T]") -> bool
        return isinstance(otro, NodoABO) and self.dato == otro.dato
```

Ejercicio

Implementar los dunder methods restantes en la clase NodoABO

Analizando el TAD

```python
class ArbolBinarioOrdenado(ArbolBinario[T]):
    @staticmethod
    def crear_nodo(dato: T) -> "ArbolBinarioOrdenado[T]":
        nuevo = ArbolBinarioOrdenado() # Crea un nuevo árbol binario ordenado vacío.
        nuevo.set_raiz(dato)  # Establece la raíz del árbol con un nuevo nodo `NodoABO` que contiene el dato proporcionado.
        return nuevo
```

Analizando el TAD

```
  def es_ordenado(self) -> bool:
        def es_ordenado_interna( arbol: "ArbolBinarioOrdenado[T]",minimo: Optional[T] = None,
maximo: Optional[T] = None) -> bool:
        # Función interna recursiva que verifica si el subárbol es ordenado.
        # Recibe como parámetros un árbol, un valor mínimo y un valor máximo. Estos valores se
usan para
        # asegurarse de que todos los nodos estén dentro de los rangos correctos (menor que
`maximo` y mayor que `minimo`).
            if arbol.es_vacio():
                return True
            if (minimo is not None and arbol.dato() <= minimo) or (maximo is not None and
arbol.dato() >= maximo):
                return False # Si el valor del nodo actual no está dentro de los límites
permitidos (menor que `maximo` y mayor que `minimo`)
            return es_ordenado_interna(arbol.si(), minimo, arbol.dato()) and
es_ordenado_interna(arbol.sd(), arbol.dato(), maximo) # Realizamos una llamada recursiva para el
subárbol izquierdo y el derecho, ajustando los límites mínimo y máximo basados en el valor actual
del nodo.
        return es_ordenado_interna(self)
```

Analizando el TAD

```python
    def insertar_si(self, arbol: "ArbolBinarioOrdenado[T]"):
        # Método que inserta un árbol binario en el subárbol izquierdo, preservando el orden.
        si = self.si()
        # Guardamos el subárbol izquierdo actual antes de realizar la inserción.

        super().insertar_si(arbol)
        # Llamamos al método de la clase padre para insertar el nuevo árbol en el subárbol izquierdo.

        if not self.es_ordenado():
            super().insertar_si(si)
            # Si después de la inserción el árbol no es ordenado, revertimos la inserción y restauramos el subárbol original.
            raise ValueError("El árbol a insertar no es ordenado o viola la propiedad de orden del árbol actual")
            # Lanzamos un error si el árbol que intentamos insertar no preserva el orden.
```

Analizando el TAD

```
 def insertar_sd(self, arbol: "ArbolBinarioOrdenado[T]"):
        # Método que inserta un árbol binario en el subárbol derecho, preservando el orden.
        sd = self.sd()
        # Guardamos el subárbol derecho actual antes de realizar la inserción.

        super().insertar_sd(arbol)
        # Llamamos al método de la clase padre para insertar el nuevo árbol en el subárbol
derecho.

        if not self.es_ordenado():
            super().insertar_sd(sd)
            # Si después de la inserción el árbol no es ordenado, revertimos la inserción y
restauramos el subárbol original.
            raise ValueError("El árbol a insertar no es ordenado o viola la propiedad de orden
del árbol actual")
            # Lanzamos un error si el árbol que intentamos insertar no preserva el orden.
```

Analizando el TAD

```
  def insertar(self, valor: T):
        # Método para insertar un valor en el árbol, manteniendo el orden binario de búsqueda.
        if self.es_vacio():
            self.set_raiz(NodoABO(valor))
            # Si el árbol está vacío, insertamos el valor como la raíz del árbol.
        elif valor < self.dato():
            self.si().insertar(valor)
            # Si el valor es menor que el valor en la raíz, lo insertamos recursivamente en el
subárbol izquierdo.
        else:
            self.sd().insertar(valor)
            # Si el valor es mayor o igual al valor en la raíz, lo insertamos recursivamente en
el subárbol derecho.
```

¿Cuál es el problema con este código?

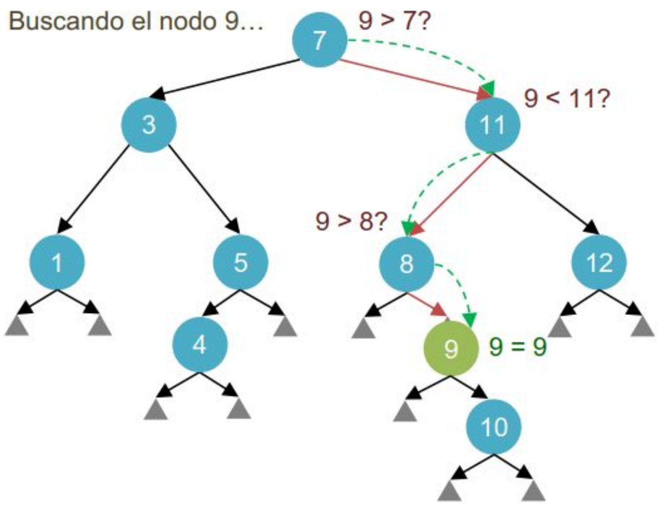
Ejercicio

Mejorar el algoritmo de inserción para que no admita elementos repetidos.

Ejercicio

Desarrollar la función _pertenece_ que se encuentra en el TAD respetando su firma

Búsqueda



La **estrategia** de búsqueda en un árbol binario ordenado **es la misma** que se utiliza en la inserción. Ante cada nodo visitado nos preguntamos **si es el nodo buscado**, de lo contrario realizamos la **misma búsqueda recursiva**, pero en el **subárbol izquierdo o derecho**, dependiendo si la clave buscada es menor o mayor a la actual.

Ejercicio

Desarrollar el algoritmo de búsqueda de un árbol binario de búsqueda, que devuelva True si el valor proporcionado fue encontrado o, caso contrario, False.

Ejercicio

Desarrollar la función recursiva <u>minimo</u> que devuelva el valor mínimo del Árbol Binario Ordenado, en caso de existir.

Ejercicio

Desarrollar una función recursiva valores_menor_a que, dado un valor, devuelva una lista con todos los valores menores a ese valor proporcionado

Ejercicio

Desarrollar una función recursiva recorrer_mayor_menor que devuelva una lista con los valores contenidos en el árbol binario ordenado de mayor a menor

Ejercicio

Desarrollar la función recursiva convertir_ordenado del TAD que convierte un árbol binario (sin que tengamos la garantía de que esté ordenado ordenado) en un árbol binario de búsqueda ordenado.

Ejercicio

Desarrollar la función recursiva con la siguiente firma: _encontrar_max(self, arbol: "ArbolBinarioOrdenado[T]") -> "ArbolBinarioOrdenado[T]", que encuentra y devuelve el subárbol que contiene el valor máximo en el subárbol pasado como parámetro.

Ejercicio

Desarrollar la función recursiva con la siguiente firma: _encontrar_min(self, arbol: "ArbolBinarioOrdenado[T]") -> "ArbolBinarioOrdenado[T]", que encuentra y devuelve el subárbol que tiene el valor mínimo en su raíz.

Eliminación

En la eliminación, existen **diversas alternativas** para **conectar** los **fragmentos** que quedan al eliminar un nodo, ya que debe mantenerse un **árbol binario ordenado DESPUÉS** de la eliminación.

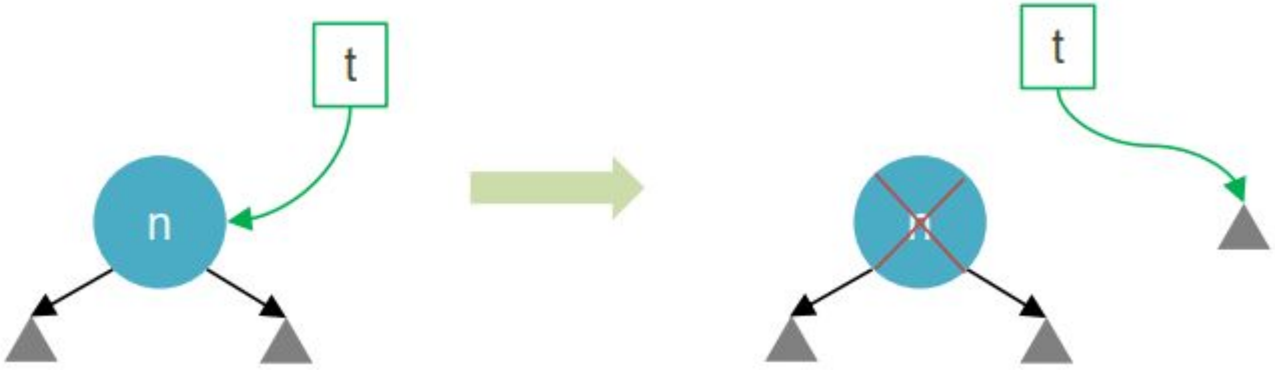
Existen **cuatro situaciones**:

1) **Caso trivial:** aquellos donde **el nodo a eliminar tiene como máximo un descendiente**
   a) **Eliminar hoja → convertirlo en árbol vacío**
   b) **Eliminar nodo con un descendiente izquierdo o uno derecho** (uno de los dos): el **si** o el **sd** pasa ser el árbol.

2) Eliminar nodo con **dos descendientes**
   a) **Fusión**
   b) **Copia**

Caso trivial: Eliminar hoja

Simplemente se deberá **liberar el contenido** de memoria de **n** y nos quedará un **árbol vacío**.



Caso trivial: Eliminar hoja

Caso trivial: Eliminar nodo con descendiente

Cuando debemos eliminar un **nodo** que **sólo tiene un subárbol derecho**, simplemente **reemplazamos** a ese nodo por su **subárbol derecho**

Caso trivial: Eliminar nodo con descendiente

Cuando debemos eliminar un **nodo** que sólo tiene un **subárbol izquierdo**, simplemente **reemplazamos** a ese nodo por su **subárbol izquierdo**

Eliminación por Fusión

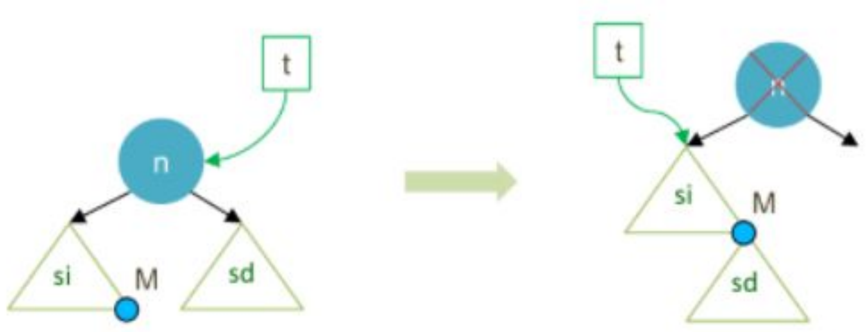
En la eliminación por fusión, el nodo a eliminar **tiene los dos subárboles no vacíos,** por lo cual **fusionamos los subárboles izquierdo y derecho en uno solo**, que luego **reemplaza al nodo eliminado**.

Eliminación por fusión

- **n** es el nodo a eliminar

-**M** es el nodo **máximo del subárbol izquierdo** de n. Tiene un **árbol vacío** como **descendiente derecho**

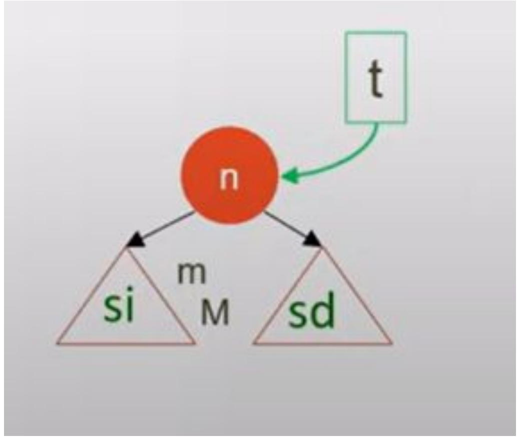
-**Idea:** conectar la raíz del subárbol derecho del nodo n como subárbol derecho del nodo M. Se desprende **sd** y se lo pone como nodo derecho de ese nodo máximo. Esto se puede hacer porque **M,** al ser **máximo**, tiene un **ÁRBOL VACÍO A LA DERECHA**



Eliminación por copia

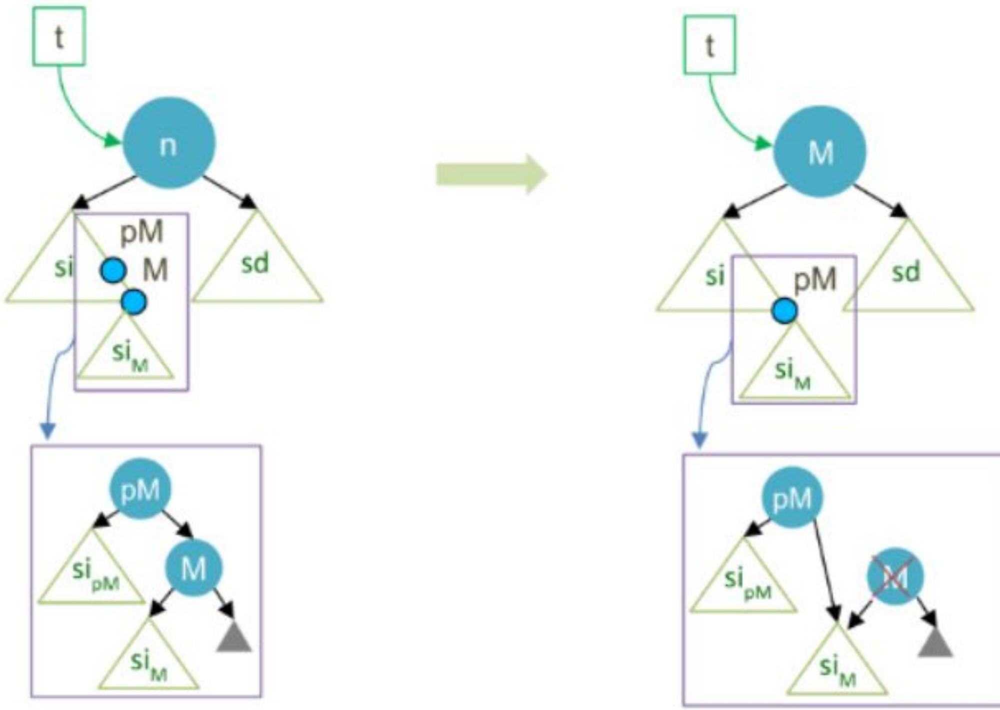
El nodo a eliminar **también tiene los dos subárboles no vacíos**, pero en este caso **copiamos el nodo máximo del lado izquierdo** al lugar del **nodo a eliminar**, para luego **reacomodar** los **subárboles necesarios** respetando la política de orden.

-Tomamos **M** (el nodo mayor del **si**) y **TAMBIÉN m** (el padre del mayor **M**)



Eliminación por copia

-**m** tiene un **si** y a **M,** que también tiene un **si** y un **“sd”** vacío.
-Lo que vamos a hacer es **copiar EL DATO DE M** al lugar donde estaba **n** (que es el que queremos eliminar)
-La **raíz del árbol** pasa a ser **M**, pero tenemos que mantener su **si** para no perderlo. => el **si pasa a ser sd del padre de M (m)**



Eliminación por copia

M representa al **máximo del subárbol izquierdo de n**

m es su **nodo predecesor** .

Copiamos el contenido (**dato**) del nodo **M** al **dato** del nodo **n**.

Liberaremos la memoria del nodo M

Antes de eliminar, **resguardar el subárbol izquierdo de M**: **siM**.

Siendo **M** el **mayor**, **siempre** tendrá un **árbol vacío como subárbol derecho**, por lo cual sólo nos preocupa **guardar el izquierdo.** Se asignará como **subárbol derecho de quien era su predecesor pM**.

Eliminación por fusión vs por copia

La **eliminación por fusión** es **más sencilla de implementar**, pero el **árbol** va quedando **desbalanceado**

Ejercicio

Desarrollar ambos algoritmos recursivos de eliminación (por fusión y copia) de un árbol binario de búsqueda. Incluir los casos triviales dentro de ellos. En caso de que el valor a eliminar no se encuentre en el árbol, no se debe realizar ninguna acción.