

# Árboles binarios de búsqueda

Clase 06

IIC 2133 - Sección 3

Prof. Eduardo Bustos

# Sumario

**Introducción**

Árboles binarios de búsqueda

Operaciones

Cierre

# El Misterio de EDD

- Hasta aquí, somos capaces de ordenar una secuencia usando diferentes algoritmos
- En determinados casos, alguno de ellos puede ser más adecuado
  - Por características del input
  - Por requisitos de memoria y tiempo

¿Cuál era el problema que motivó esta primera parte?

# El Misterio de EDD

Dada una secuencia desordenada, nos interesa **buscar** un elemento

¿ Zallen Misterio ∈

Apellido	Nombre
Alen	Misterio
Misterio	Misterio
Zalen	Berenice
Gonzalópez	D
Turing	Alan
Misterio	Yadran
Zeta	Hache
Ararán	Jota
Alenn	Cristina
...	...

pág. 1/376

?

# El Misterio de EDD

Escogemos algún **algoritmo de ordenación**

QuickSort ( $A, i, f$ ):

```
1  if  $i \leq f$  :  
2       $p \leftarrow \text{Partition}(A, i, f)$   
3      Quicksort( $A, i, p - 1$ )  
4      Quicksort( $A, p + 1, f$ )
```

# El Misterio de EDD

Obtenemos la secuencia ordenada

¿ Zallen Misterio €

Apellido	Nombre
Abarca	Yadran
Abusleme	Nicole
Arenas	Camila
Arenas	D
Bañados	Richard
Beterraga	Brócoli
Blanco	Ximena
Brahms	Johannes
Castillo	Raquel
...	...

pág. 1/376

?

# El Misterio de EDD

Usamos algún **algoritmo de búsqueda** para encontrar el elemento

BinarySearch ( $A, x, i, f$ ):

```
1  if  $f < i$  : return -1
2   $m \leftarrow \left\lfloor \frac{i+f}{2} \right\rfloor$ 
3  if  $A[m] = x$  : return  $m$ 
4  if  $A[m] > x$  :
5      return BinarySearch ( $A, x, i, m-1$ )
6  return BinarySearch ( $A, x, m+1, f$ )
```

¿Habr  otra forma de combinar **ordenaci n y b squeda**?

# Objetivos de la clase

- ☐ Comprender la noción de diccionario y qué operaciones soporta
- ☐ Conocer los árboles binarios de búsqueda
- ☐ Comprender las propiedades básicas de un ABB
- ☐ Identificar la utilidad de los ABB
- ☐ Comprender los algoritmos que implementan sus operaciones básicas



# Sumario

Introducción

**Árboles binarios de búsqueda**

Operaciones

Cierre

# Una nueva estructura

Construiremos una estructura con nuevas características

- Dada una **llave o clave**, queremos asociarle un **valor**
- Si la llave no está en la EDD, lo sabemos de forma eficiente
- Si la llave está en la EDD, también lo sabemos de forma eficiente
- Podemos agregar, modificar y eliminar **pares llave-valor** de forma eficiente

# Diccionarios

## Definición

Un **diccionario** es una estructura de datos con las siguientes operaciones

- **Asociar** un valor a una llave
- **Actualizar** el valor asociado a una llave
- **Obtener** el valor asociado a una llave
- En ciertos casos, **eliminar** de la estructura una asociación llave-valor

## Ejemplos

- RUT como llave y nombre como valor
- RUT como llave y (nombre, apellido, edad,...) como valor

# Diccionarios

Uno de los objetivos centrales de un diccionario es facilitar la **búsqueda**

- Primero buscamos la llave (si está o no está)
- Buscar = *buscar eficientemente*

¿Cómo almacenamos las llaves para lograr búsqueda eficiente?

Hasta ahora tenemos dos opciones: **arreglos** y **listas**... ¿cumplen nuestro objetivo?

# Limitaciones de arreglos y listas

En una **listas ligada** de llaves

- No tenemos **acceso por índice** de forma eficiente
- La búsqueda, incluso en el caso ordenado, es  $\mathcal{O}(n)$

En un **arreglo** de llaves

- Hay **acceso por índice** en  $\mathcal{O}(1)$
- La búsqueda en general es  $\mathcal{O}(n)$
- Para el caso ordenado, podemos lograrla en  $\mathcal{O}(\log(n))$

¿Qué punto débil tienen los arreglos comparados con las listas?

# Limitaciones de arreglos y listas

En una **listas ligada** de llaves

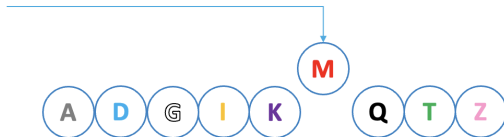
- Para insertar solo necesitamos reasignar (pocos) punteros
- La inserción de un nuevo elemento es  $\mathcal{O}(1)$

En un **arreglo** de llaves

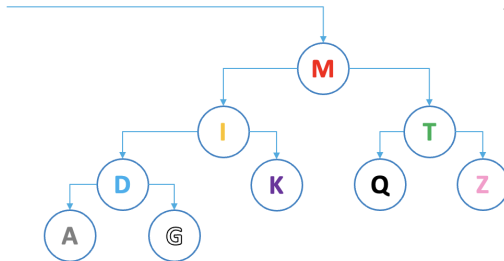
- Insertar un elemento puede gatillar un desplazamiento de datos
- En promedio, la inserción es  $\mathcal{O}(n)$

¿Podemos construir una EDD con buen desempeño en ambas operaciones?

# Modifiquemos las listas



Podemos tener un puntero a un elemento más o menos en el centro de la lista



... y ese elemento puede tener punteros a elementos más o menos en el centro de cada una de las dos sublistas, a su izquierda y a su derecha; ... y así recursivamente

# Árboles binarios de búsqueda

## Definición

Un **árbol binario de búsqueda (ABB)** es una estructura de datos que almacena **pares (llave, valor)** asociándolos mediante punteros según una estrategia recursiva

1. Un ABB tiene un **nodo** que contiene una tupla (llave, valor)
2. El nodo puede tener hasta dos ABB's asociados mediante punteros
  - Hijo izquierdo
  - Hijo derecho

y que además, satisface la **propiedad ABB**: las llaves menores que la llave del nodo están en el sub-árbol izquierdo, y las llaves mayores, en el sub-árbol derecho.

La estrategia **dividir para conquistar** aplicada a una EDD



# Árboles binarios

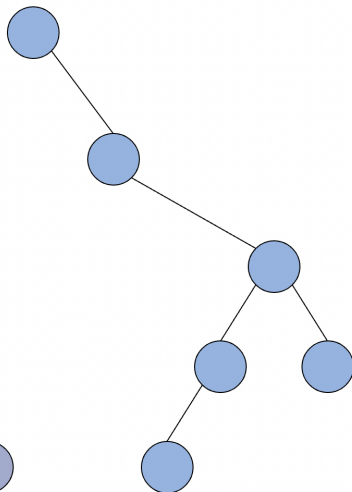
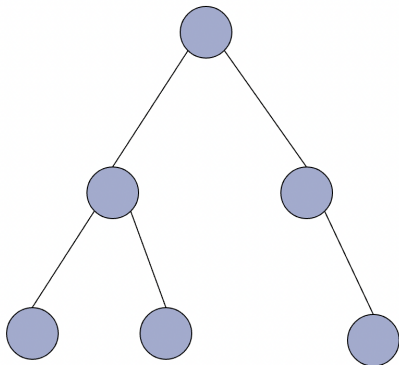
Un árbol binario (de búsqueda o no) cumple que

- Cada nodo  $x$  tiene a lo más un **padre**  $x.p$
- El nodo sin padre se conoce como **raíz**
- Cada nodo  $x$  tiene hasta dos punteros que (*apuntan*) a sub-árboles
  - $x.left$  es un puntero al hijo izquierdo
  - $x.right$  es un puntero al hijo derecho
  - $x.p$  es puntero al padre (si tiene)
- Un nodo sin punteros descendentes, i.e. sin hijos, se conoce como **hoja**

¿Necesariamente un árbol binario tiene nodos con la misma cantidad de hijos?

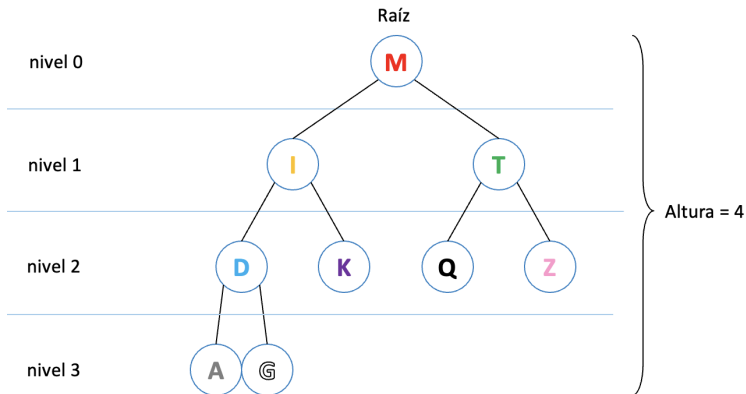
# Árboles binarios

Dos ejemplos de árboles binarios con 6 nodos



# Anatomía de un árbol binario

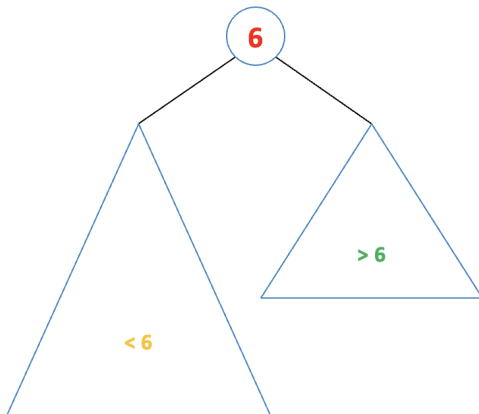
Por simplicidad, representaremos solo las llaves de los árboles



Notemos que ir de una hoja a la raíz toma tiempo  $\mathcal{O}(\text{altura})$

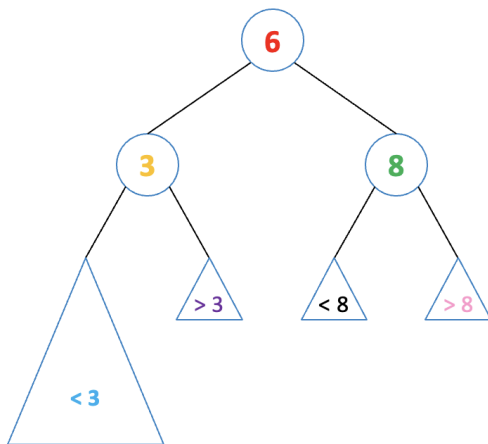
# Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



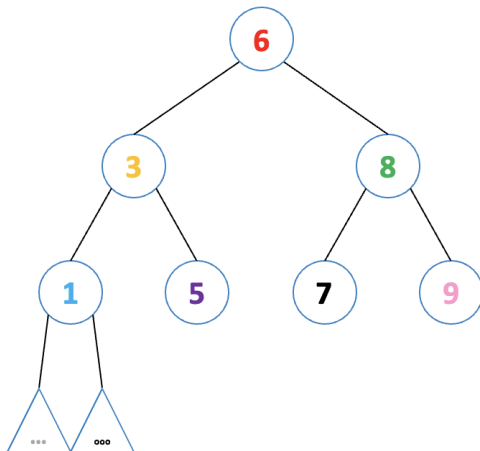
# Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



# Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



# Sumario

Introducción

Árboles binarios de búsqueda

**Operaciones**

Cierre

# Operaciones de un ABB

Recordemos nuestro objetivo al definir esta nueva estructura

- Queremos búsqueda rápida
- Para esto buscamos lograr un diccionario
- Queremos garantizar operaciones eficientes para búsqueda, inserción, modificación y eliminación
- A través de la definición concreta de estas operaciones para un ABB mostraremos que un ABB nos sirve como **diccionario**



# Operaciones de un ABB

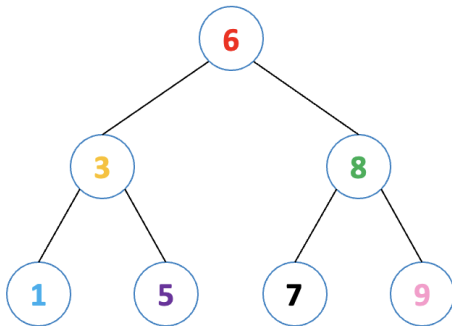
Completamos las definiciones de atributos de un **nodo**  $x$  en un ABB

- $x.key$  es la llave del nodo
- $x.value$  es su valor
- $x.left$  el puntero a su hijo izquierdo
- $x.right$  el puntero a su hijo derecho
- $x.p$  el puntero al padre

En general no incluiremos  $x.value$  en los algoritmos.  
Solo será un espacio de almacenamiento

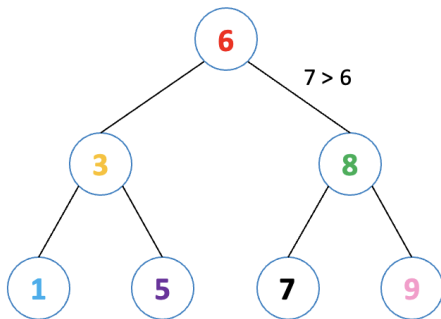
# Ejemplo de búsqueda

Nos interesa encontrar el nodo con llave 7. Solo conocemos el nodo raíz



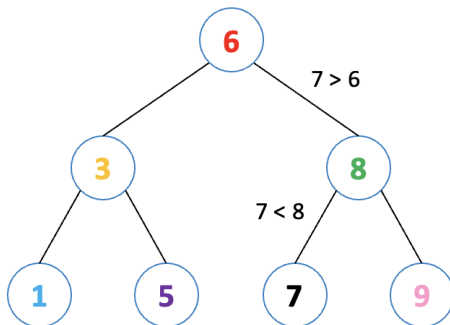
## Ejemplo de búsqueda

Comparamos con la llave raíz y sabemos que, si está, debe estarlo en el sub-árbol derecho



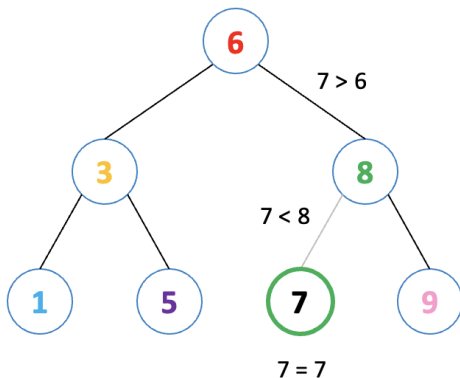
## Ejemplo de búsqueda

Recursivamente, repetimos para la raíz del sub-árbol detectado y determinamos que hay que revisar el sub-árbol izquierdo



## Ejemplo de búsqueda

Al revisar la raíz de este nuevo sub-árbol, encontramos la llave buscada



# Operación de búsqueda

Proponemos el siguiente algoritmo de búsqueda en ABB's

**input** : Árbol binario de búsqueda  $A$ , llave buscada  $k$

**output**: Árbol binario de búsqueda, o  $\emptyset$  si no se encuentra

Search ( $A, k$ ):

```
1  if  $A = \emptyset \vee A.key = k$  :  
2      return  $A$   
3  if  $k < A.key$  :  
4      return Search( $A.left, k$ )  
5  return Search( $A.right, k$ )
```

El llamado inicial es Search( $root, k$ ) para la raíz  $root$  del árbol

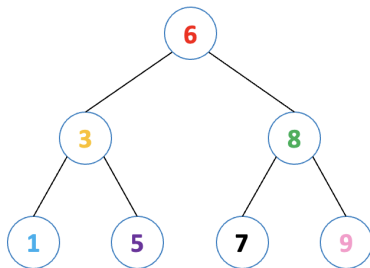
# Operaciones para modificar un ABB

Pensemos ahora en modificar el árbol

- **Insertar** un nodo con una nueva llave produce un cambio en la estructura del árbol
- De igual forma, **eliminar** un nodo también lo hace
- Ambas operaciones pueden afectar la **propiedad ABB**
- Nuestra propuesta de algoritmos para estas operaciones debe **restaurar la propiedad ABB** si se incumple

# Ejemplo de inserción

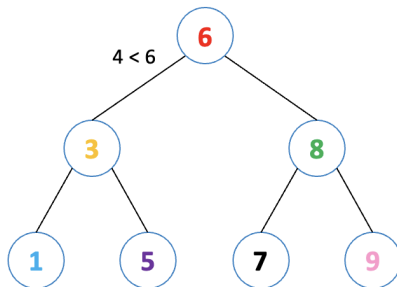
Insertemos un nodo con llave 4





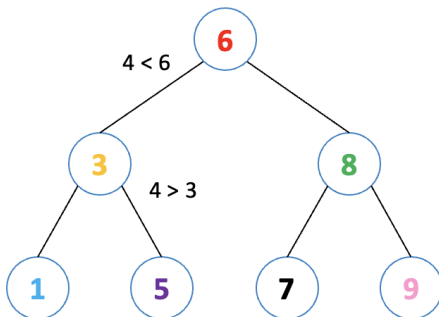
## Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado

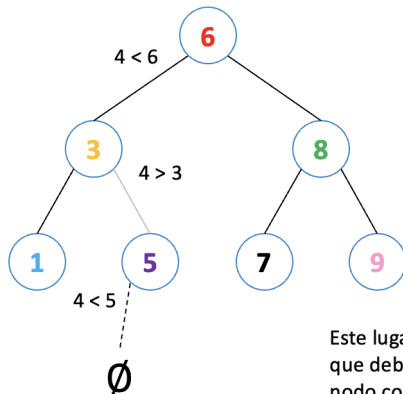


## Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado



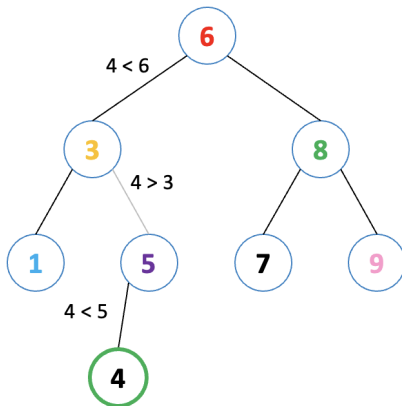
## Ejemplo de inserción



Este lugar vacío es el lugar en que debería haber estado el nodo con clave 4 si hubiera estado en el árbol

## Ejemplo de inserción

Dado que, para  $x.key = 5$  se tiene  $x.left = \emptyset$ , lo reemplazamos con la llave indicada



# Un cambio a la búsqueda

Modificaremos la búsqueda para saber quién es el padre del nodo encontrado

**input** : ABB  $A$ , ABB padre  $p$ , llave buscada  $k$

**output**: Tupla con ABB encontrado y su padre

Search( $A, p, k$ ):

```
1  if  $A = \emptyset \vee A.key = k$  :  
2      return ( $A, p$ )  
3  if  $k < A.key$  :  
4      return Search( $A.left, A, k$ )  
5  return Search( $A.right, A, k$ )
```

Si retorna ( $A, \emptyset$ ), sabemos que  $A$  es la raíz

# Operación de inserción

Proponemos el siguiente algoritmo de inserción de valores según llave ABB's

**input** : Árbol binario de búsqueda  $A$ , llave  $k$ , valor  $v$

Insert ( $A, k, v$ ):

```
1  ( $B, p$ )  $\leftarrow$  Search( $A, \emptyset, k$ )  ▷ versión que indica el padre
2  if  $B = \emptyset$  :
3       $B \leftarrow$  nodo vacío
4       $B.key \leftarrow k$ 
5      Conectar  $B$  al padre  $p$  en la posición adecuada
6   $B.value \leftarrow v$ 
```

Este algoritmo mantiene la propiedad ABB al insertar

# Operación de eliminación

La eliminación es un poco más compleja

Si el nodo a eliminar es hoja o tiene solo un hijo

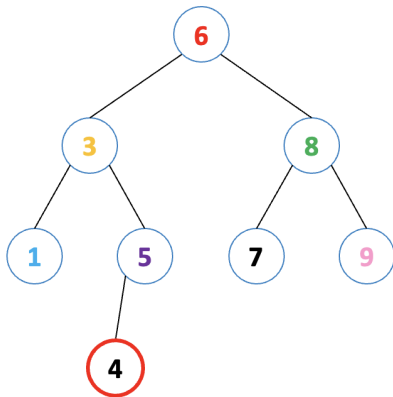
- Lo borramos
- Si tenía un hijo, el hijo lo reemplaza
- Es claro que se mantiene la propiedad ABB

En caso contrario...

¿Se puede reemplazar por otro árbol?

# Ejemplo de eliminación

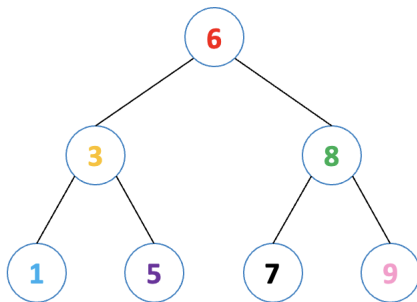
Si queremos eliminar el nodo con llave 4





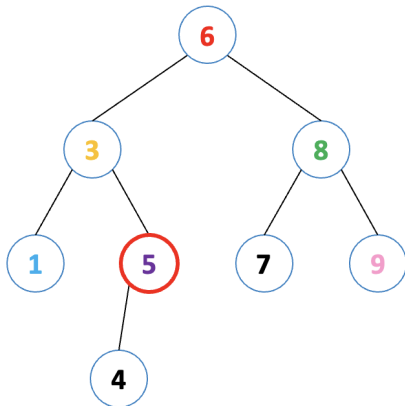
# Ejemplo de eliminación

Simplemente se elimina y se preserva la propiedad ABB



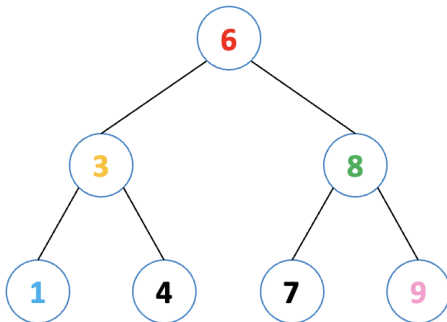
# Ejemplo de eliminación

Si queremos eliminar el nodo con llave 5



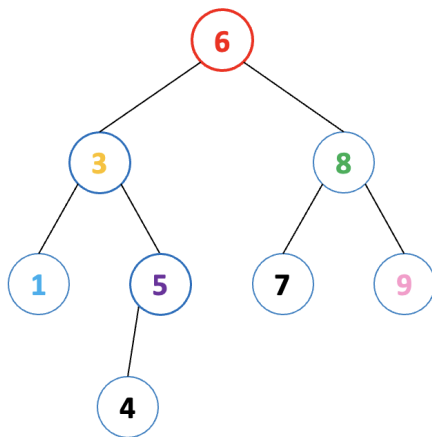
## Ejemplo de eliminación

Se reemplaza por su único hijo y se preserva la propiedad ABB



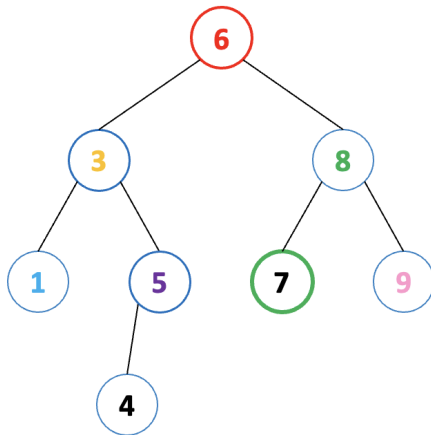
# Ejemplo de eliminación

Si queremos eliminar el nodo con llave 6, estamos en problemas



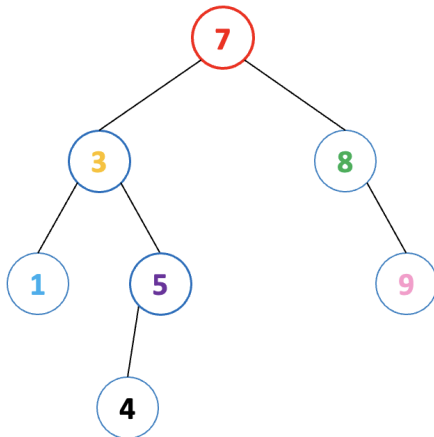
# Ejemplo de eliminación

Podemos reemplazarlo por el nodo con llave 7 (**su sucesor**)



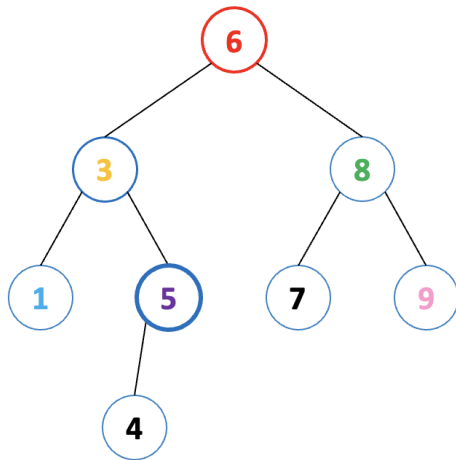
## Ejemplo de eliminación

Y dado que no tenía hijos, no hay que hacer más modificaciones



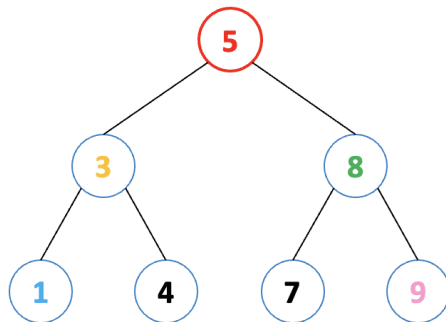
## Ejemplo de eliminación

De forma alternativa, podemos reemplazarlo por el nodo con llave 5 (su **antecesor**)



# Ejemplo de eliminación

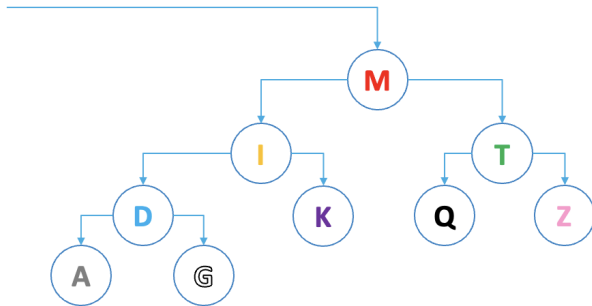
Y reubicamos su hijo con llave 4





# Operación de eliminación

Nos interesa encontrar el sucesor/antecesor del nodo extraído



Min (A):

```
1  if A.left =  $\emptyset$  :  
2      return A  
3  return Min(A.left)
```

Max (A):

```
1  if A.right =  $\emptyset$  :  
2      return A  
3  return Max(A.right)
```

# Operación de eliminación

Proponemos el siguiente algoritmo que preserva la propiedad ABB

Delete ( $A, k$ ):

```
1  ( $D, p$ )  $\leftarrow$  Search( $A, \emptyset, k$ )  ▷ Permite saber el padre de  $D$ 
2  if  $D \neq \emptyset$  :
3      if  $D$  es hoja :  $D \leftarrow \emptyset$  y se elimina la referencia en  $p$ 
4      elif  $D$  tiene un solo hijo  $H$  :  $D \leftarrow H$  y se actualiza  $p$ 
5      else:
6           $R \leftarrow \text{Min}(D.\text{right})$ 
7           $t \leftarrow R.\text{right}$ 
8           $D.\text{key} \leftarrow R.\text{key}$ 
9           $D.\text{value} \leftarrow R.\text{value}$ 
10          $R \leftarrow t$ 
```

Notemos que al borrar un nodo, se debe eliminar la referencia desde su padre

# Antecesor y sucesor en general

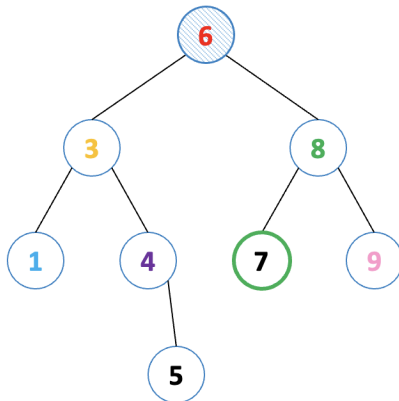
¿Qué tan fácil es determinar el sucesor y antecesor de un nodo?

Ya tenemos algoritmos recursivos para esto

¿Y si los tuviéramos en una lista ordenados?

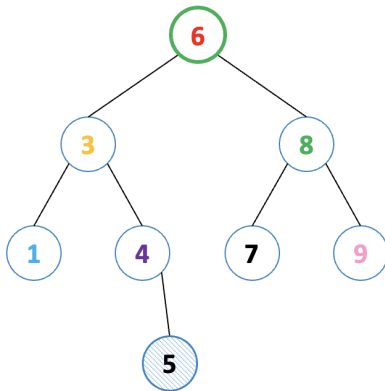
# Antecesor y sucesor en general

Ya sabemos que es *fácil* encontrarlos preguntando por la raíz



# Antecesor y sucesor en general

Pero ya no tenemos acceso a Min y Max si preguntamos por un nodo no raíz



# Sumario

Introducción

Árboles binarios de búsqueda

Operaciones

**Cierre**

# Objetivos de la clase

- ☐ Comprender la noción de diccionario y qué operaciones soporta
- ☐ Conocer los árboles binarios de búsqueda
- ☐ Comprender las propiedades básicas de un ABB
- ☐ Identificar la utilidad de los ABB
- ☐ Comprender los algoritmos que implementan sus operaciones básicas