Diccionario: estructura de datos con las siguientes operaciones

Asociar un **valor** (p.ej., un archivo con la solución de la tarea 1) a una **clave** (p.ej., un rut o número de alumno)

… o **actualizar** el valor asociado a la clave (p.ej., cambiar el archivo)

Obtener el **valor** asociado a una **clave**

(… y para ciertos casos de uso)

Eliminar del diccionario una **clave** y su **valor** asociado

Así, la idea de un diccionario es:

… si me dan el rut (la clave), entonces yo quiero encontrar el archivo

…. si me dan el rut y me doy cuenta de que ese rut no está en mis registros (el diccionario), entonces ingresar el rut a mis registros

… si me dan el rut y me doy cuenta de que no hay un archivo asociado, entonces asociar un archivo al rut

… si me dan el rut y me doy cuenta de que tiene un archivo asociado, entonces cambiar el archivo por uno más actual

La búsqueda es lo primero

O sea, a partir del rut, lo primero es buscarlo en el diccionario (y encontrarlo o estar seguros de que no está)

… y por “buscarlo” queremos decir buscarlo rápidamente, eficientemente

¿Cómo logramos esto? es decir **¿qué estructura de datos nos conviene usar para lograrlo?**

(en los ejemplos, vamos a mostrar sólo las claves, no los valores, y las claves van a ser números enteros no muy grandes o las letras del abecedario o similar **)**

Recordemos lo que sabemos

Recurramos primero a nuestras opciones fundamentales de organización de información en memoria principal

¿Recuerdan cuáles son?

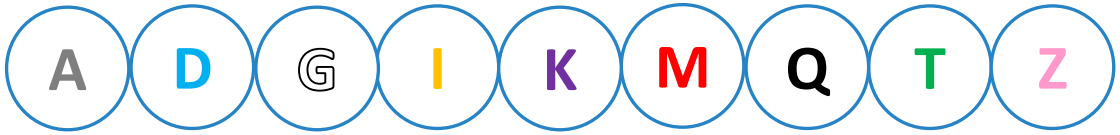
¿Cuáles son las ventajas y limitaciones de c/u?

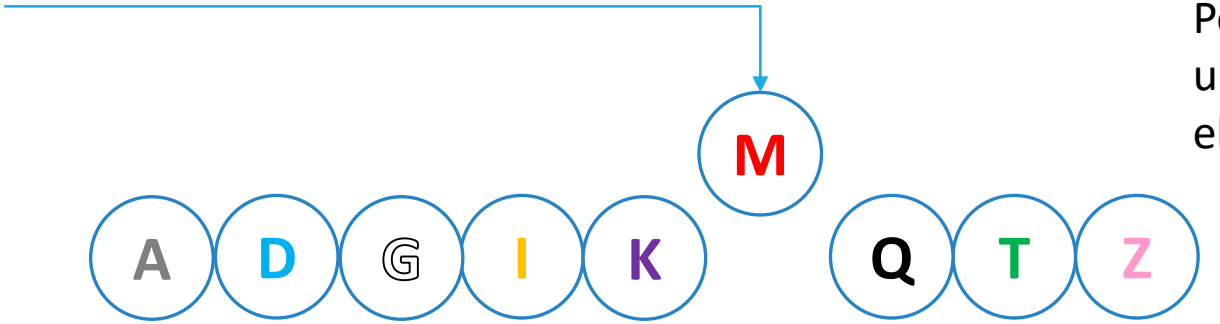
La lista ligada frente al arreglo

La limitación para buscar eficientemente una clave en una lista ligada de claves ordenadas —comparando con un arreglo— es que no tenemos cómo “ir” en un paso a la mitad de la lista (algo que sí podemos hacer en un arreglo)

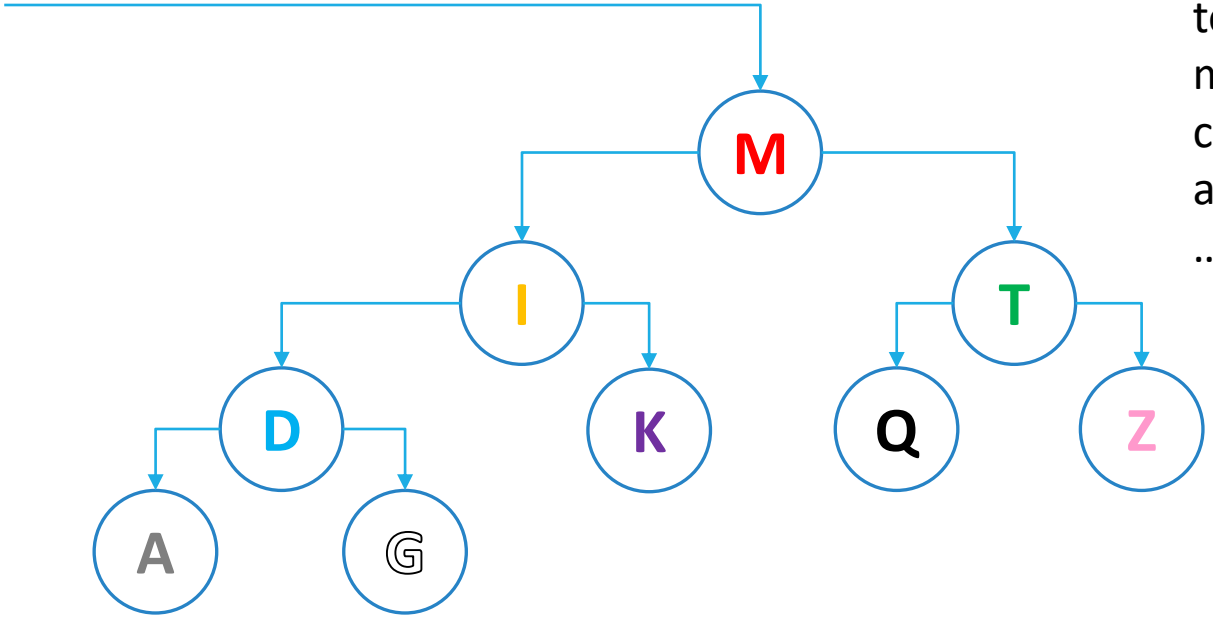
Por su parte, la limitación del arreglo —comparando con una lista ligada— no está en la búsqueda, sino en la inserción de una nueva clave ordenadamente con respecto a las claves que ya están:

- exige desplazar, en promedio, la mitad de los elementos del arreglo (en una lista, no es necesario desplazar nada)





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

El árbol binario de búsqueda (ABB)

Es una estructura de datos que guarda tuplas —pares *(key, value)*— organizadas en nodos de forma recursiva:

- en las figuras, mostramos sólo las *keys*

La raíz del árbol almacena una tupla y el resto se organiza recursivamente en uno o dos ABBs como hijos (izquierdo y/o derecho) de la raíz:

- la estrategia dividir para reinar aplicada a la estructura de datos

Propiedad ABB: Los *keys* menores que la raíz cuelgan del hijo izquierdo, y los *keys* mayores, del hijo derecho … **recursivamente**

En un **árbol binario** (ya sea de búsqueda o no), cada nodo *x* es apuntado por un solo nodo, su *padre* (*x.p*)

… excepto el nodo *raíz*, que no es apuntado por ningún otro nodo

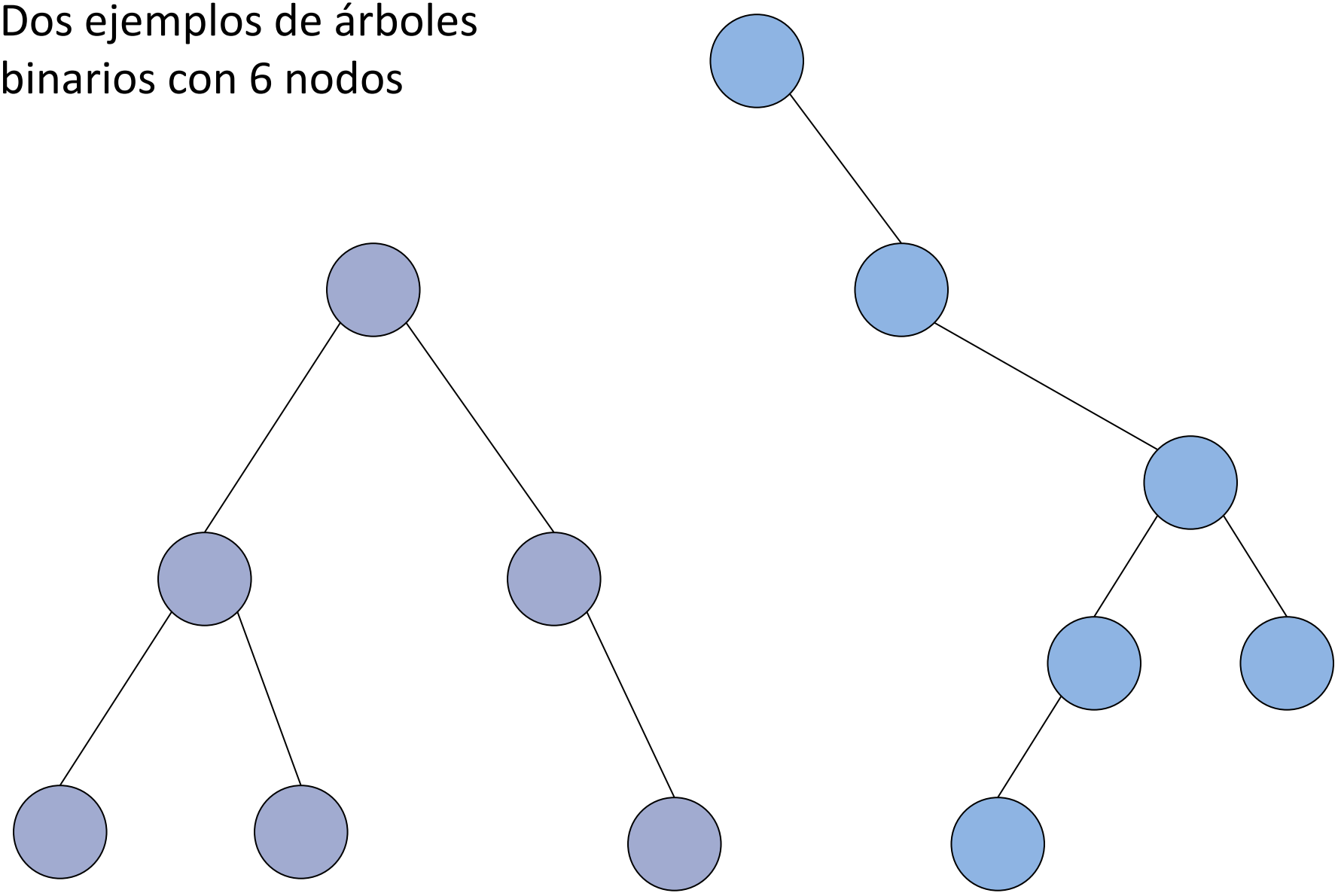
Cada nodo *x* tiene dos links, uno izquierdo (*x.left*) y otro derecho (*x.right*), que apuntan respectivamente a los nodos llamados

… el *hijo izquierdo de x* … el *hijo derecho de x*

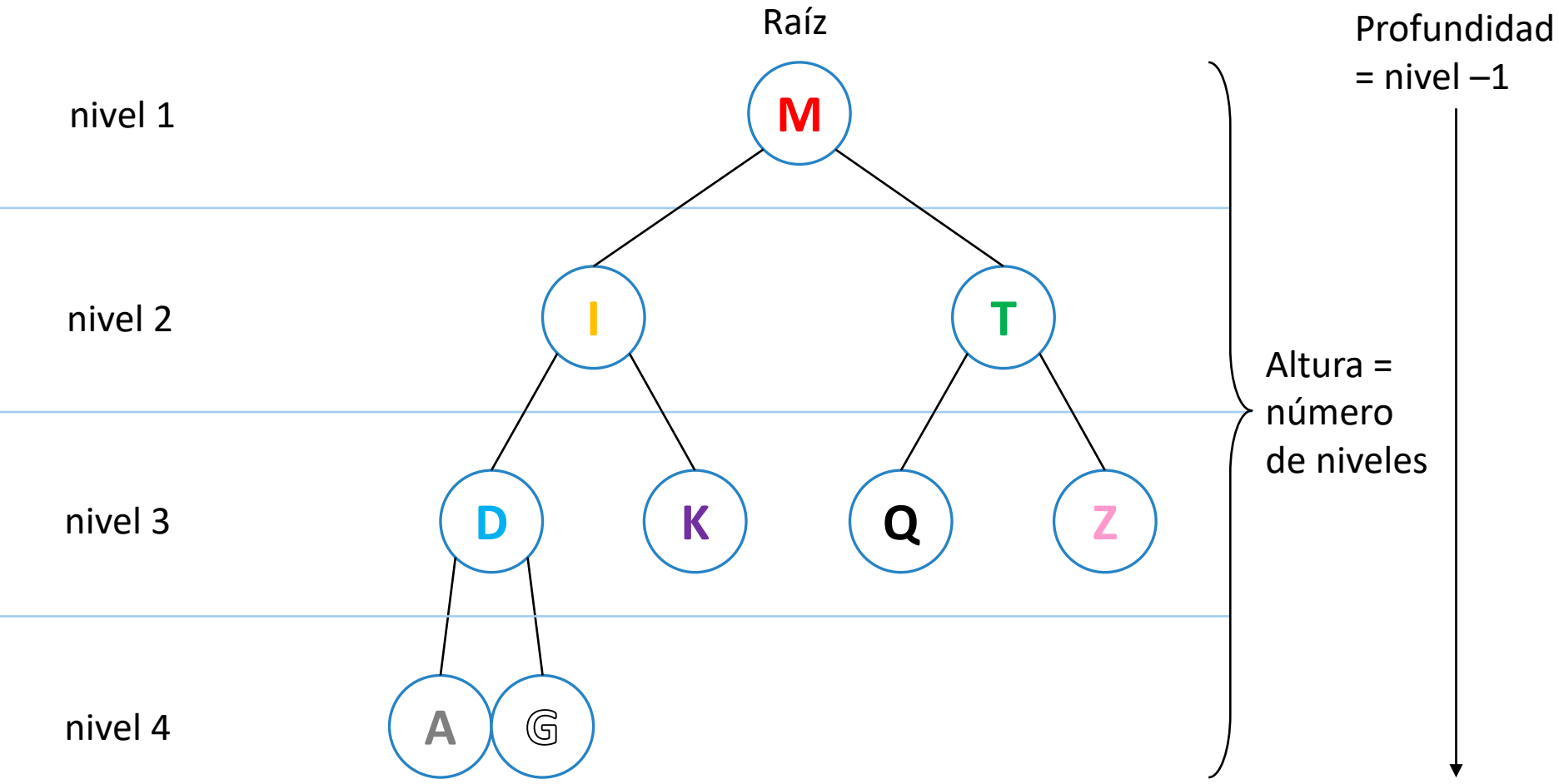
Un **árbol binario** es, además, una *estructura recursiva*:

aún cuando los links apuntan a nodos, podemos ver cada link como apuntando a un árbol binario → el árbol cuya raíz es el nodo apuntado

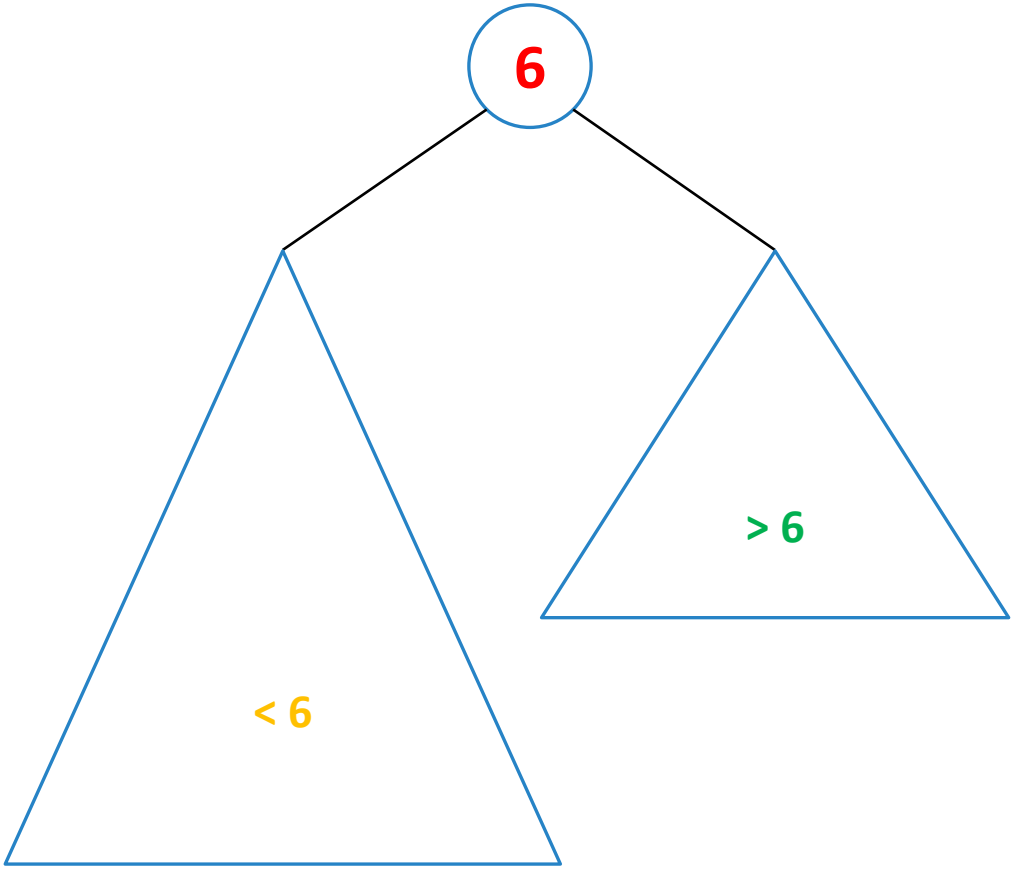
Dos ejemplos de árboles binarios con 6 nodos



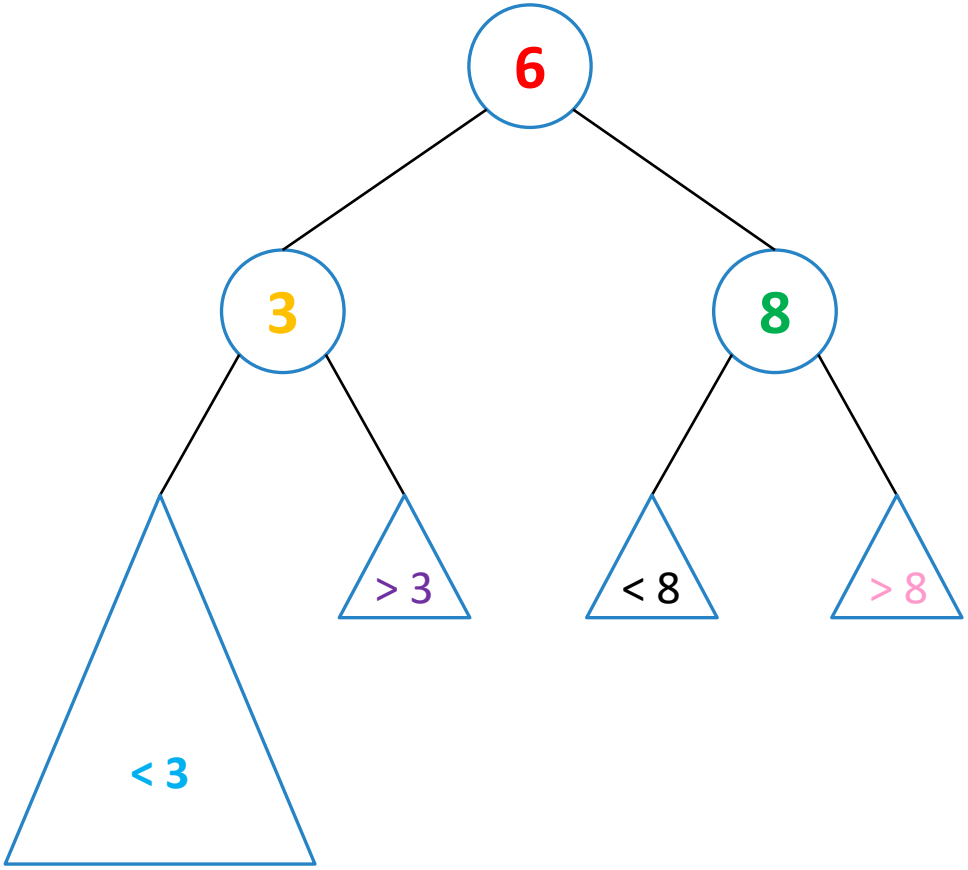
Anatomía de un árbol binario (mostramos solo las *keys*)



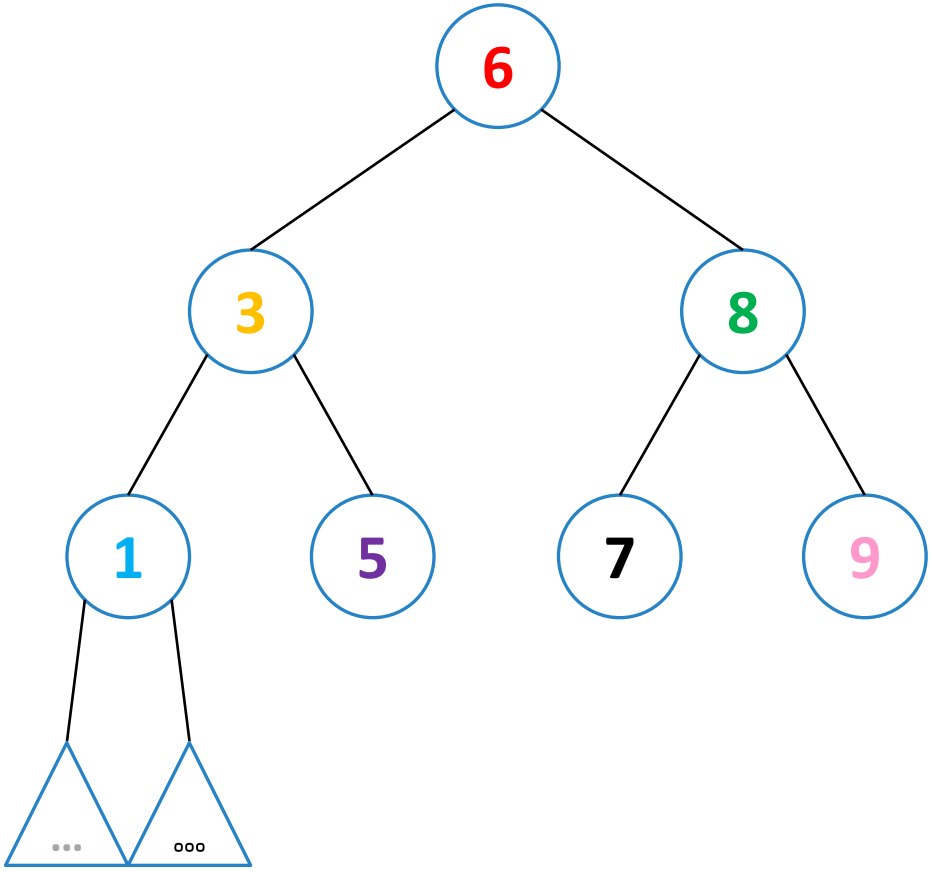
El árbol binario de búsqueda ...



… está compuesto por (sub) árboles binarios de búsqueda



… y así hasta las hojas

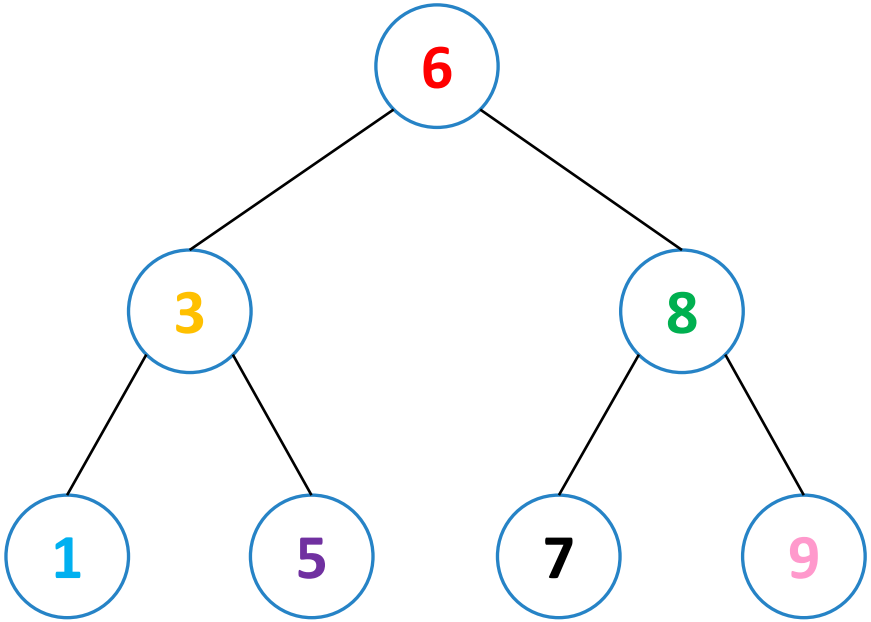


Operaciones del ABB

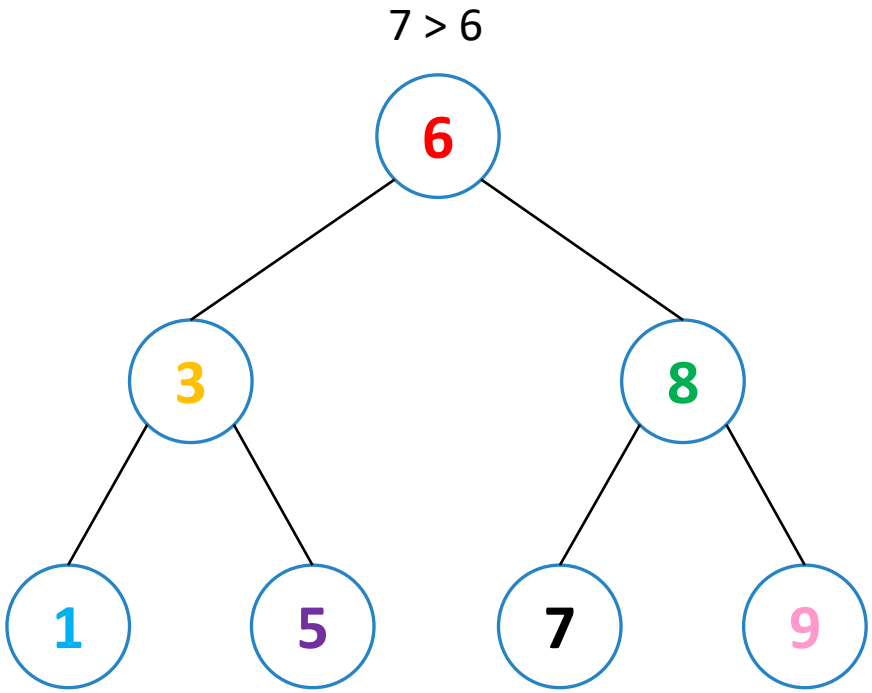
¿Cómo se busca un elemento en el árbol?

Tratemos de aprovechar que la estructura es recursiva

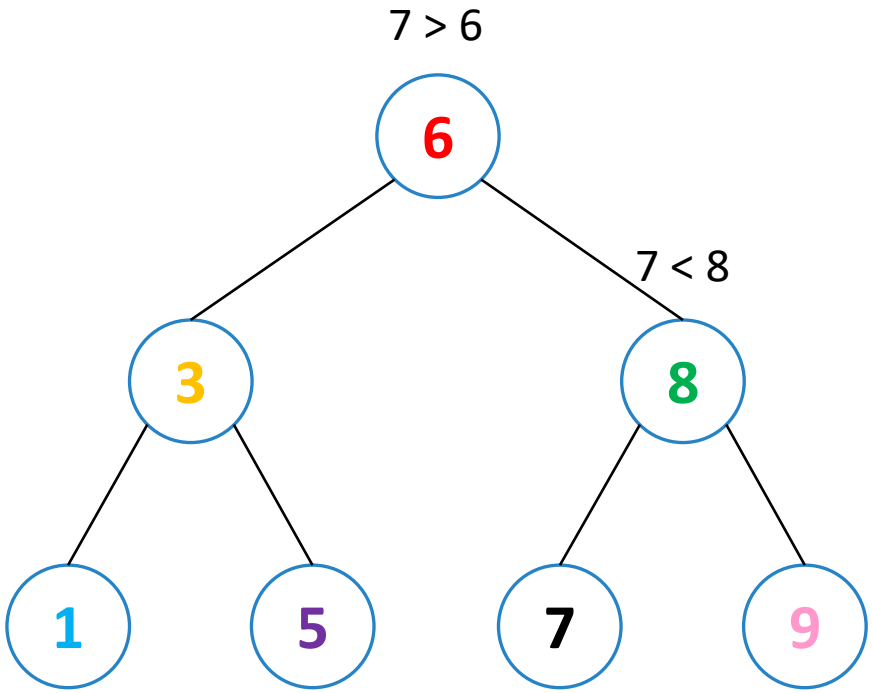
Busquemos el 7



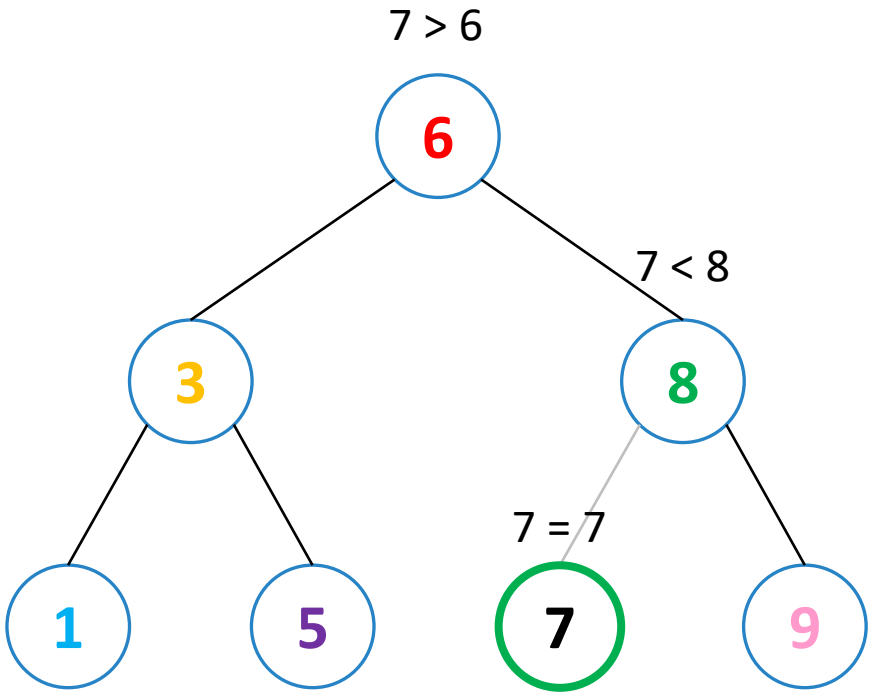
Busquemos el 7



Busquemos el 7



Busquemos el 7



```
search(A, k):
    if A = ∅  o  A.key = k:
        return A
    else if k < A.key:
        search(A.left , k)
    else:
        search(A.right, k)
```

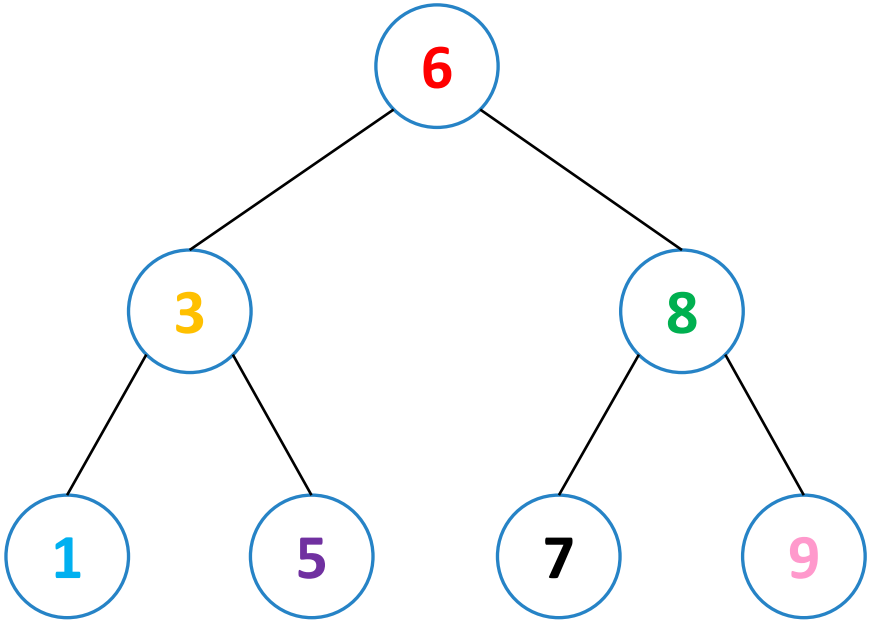
Operaciones que modifican el árbol

Insertar una nueva *key* (y su *value* asociado) produce un cambio en la estructura del árbol

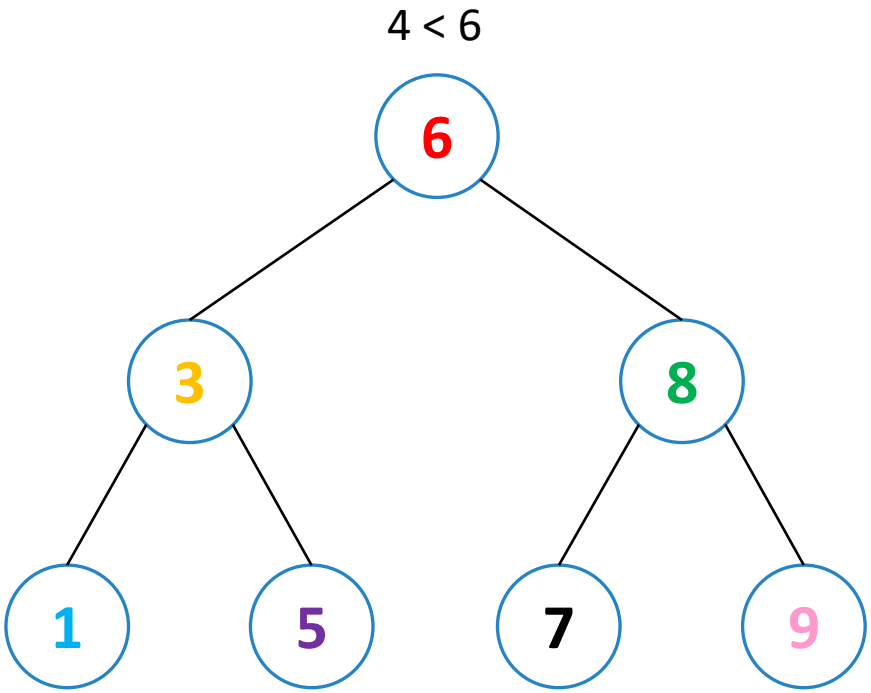
Eliminar una *key* (y su *value* asociado) produce un cambio en la estructura del árbol

Ambas operaciones hay que realizarlas de modo de que, una vez terminadas, el árbol sea efectivamente un ABB
→ si es necesario, hay que restaurar la propiedad de ABB

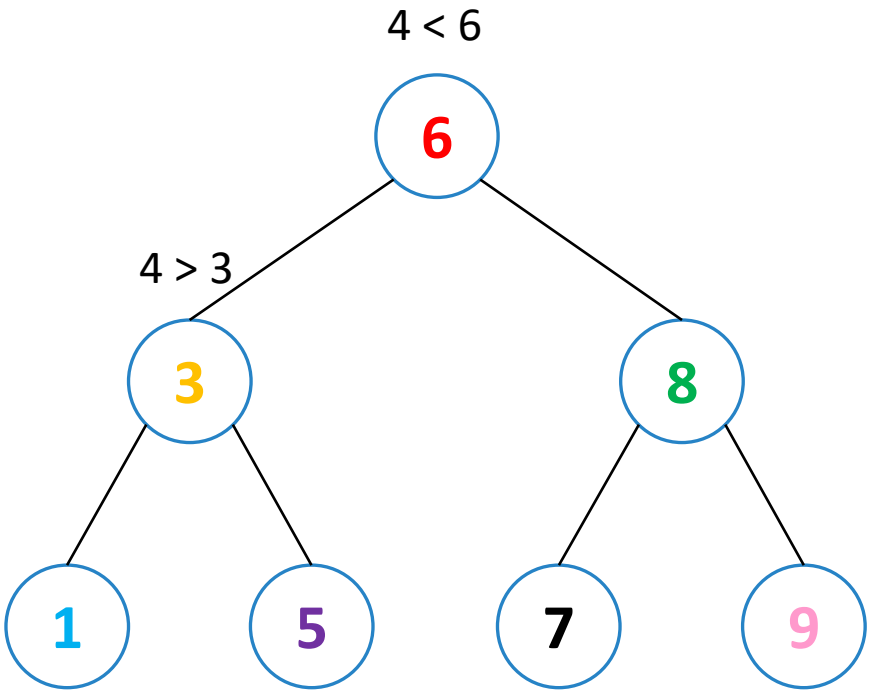
Insertemos el 4



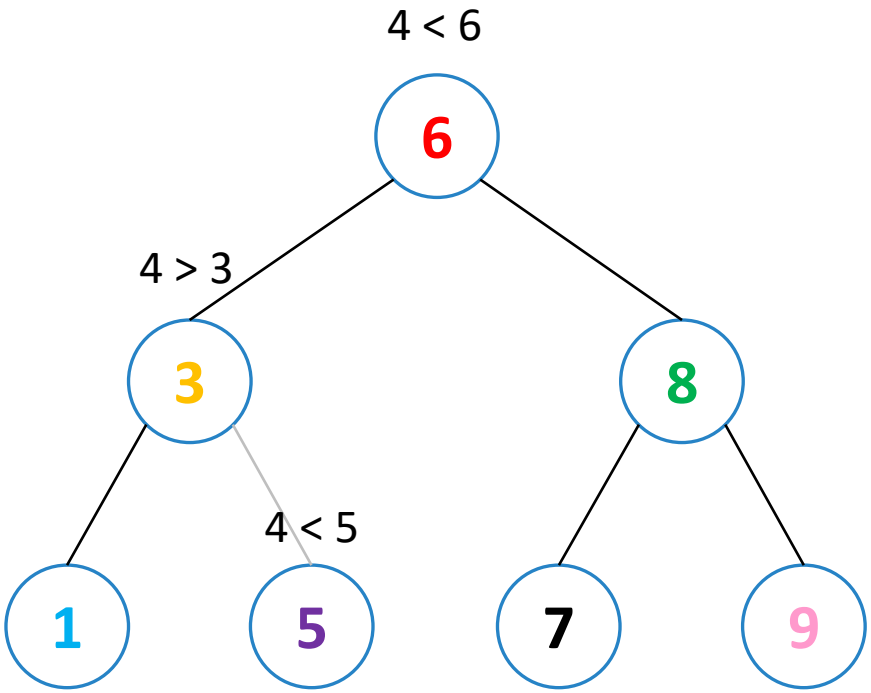
Insertemos el 4: primero, hay que buscarlo



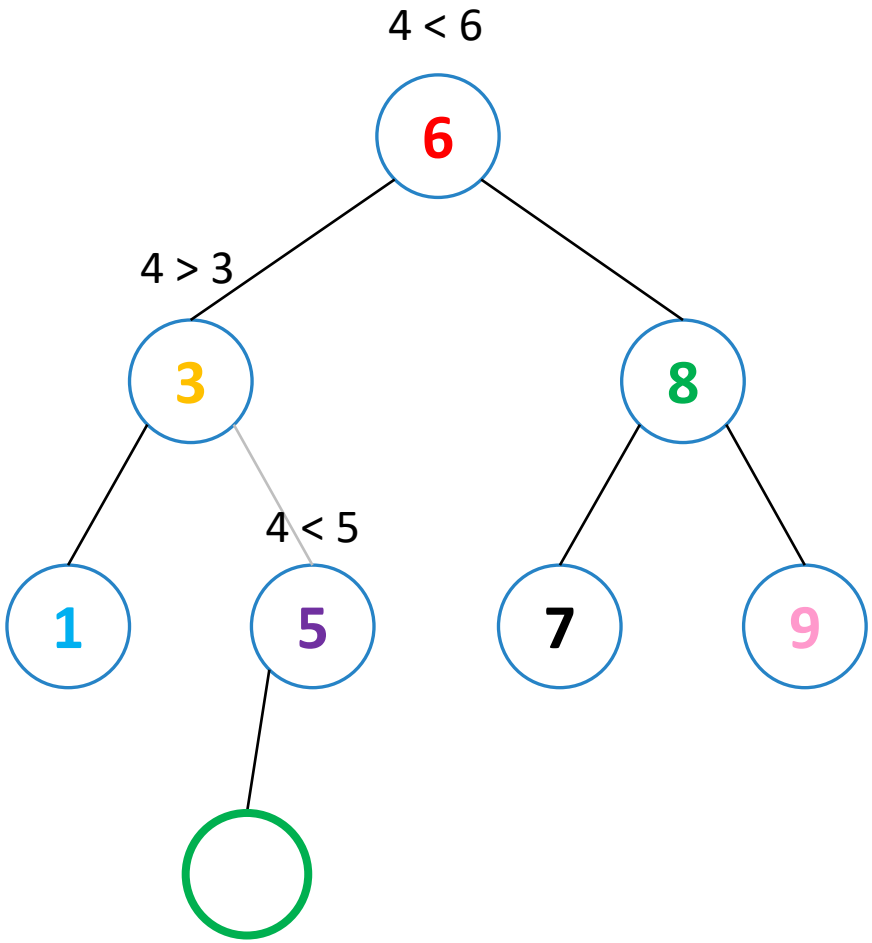
… seguimos buscando el 4



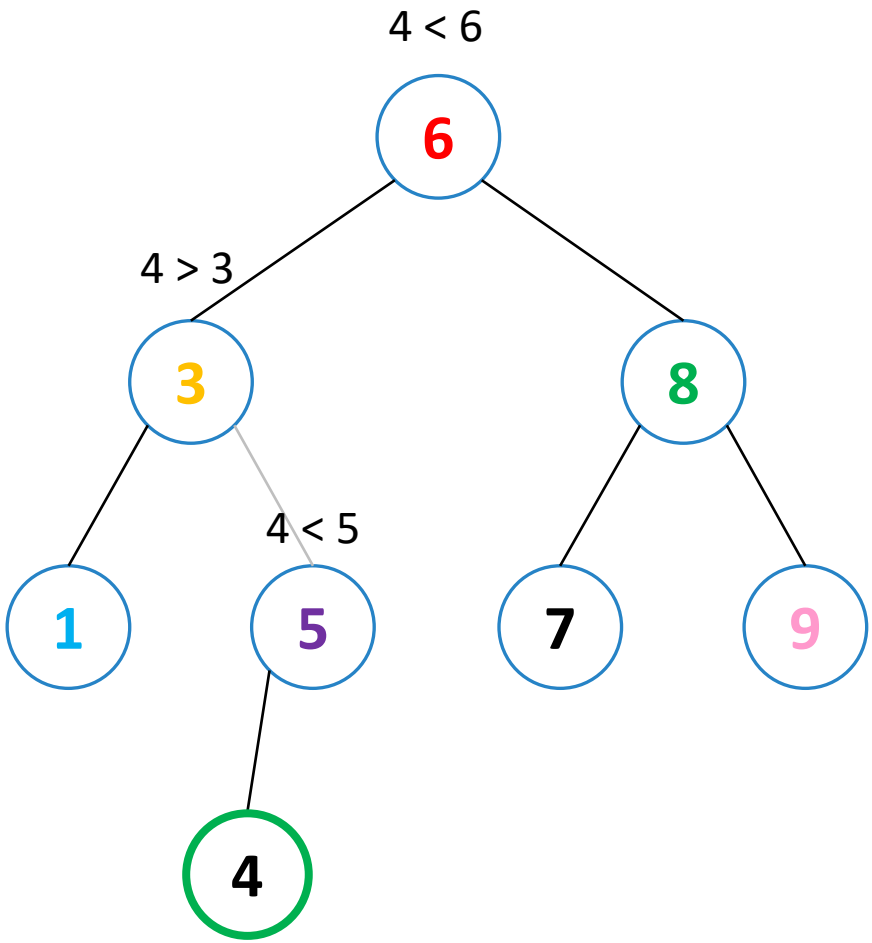
… seguimos buscando el 4



… hasta que llegamos al lugar en que debería estar y no está …



… y ahí insertamos el 4



$\boldsymbol{insert(A, k)}$:

$\boldsymbol{B \leftarrow search(A, k)}$

—crear un nodo B

—conectar B al árbol

$\boldsymbol{B.key \leftarrow k}$

Eliminación

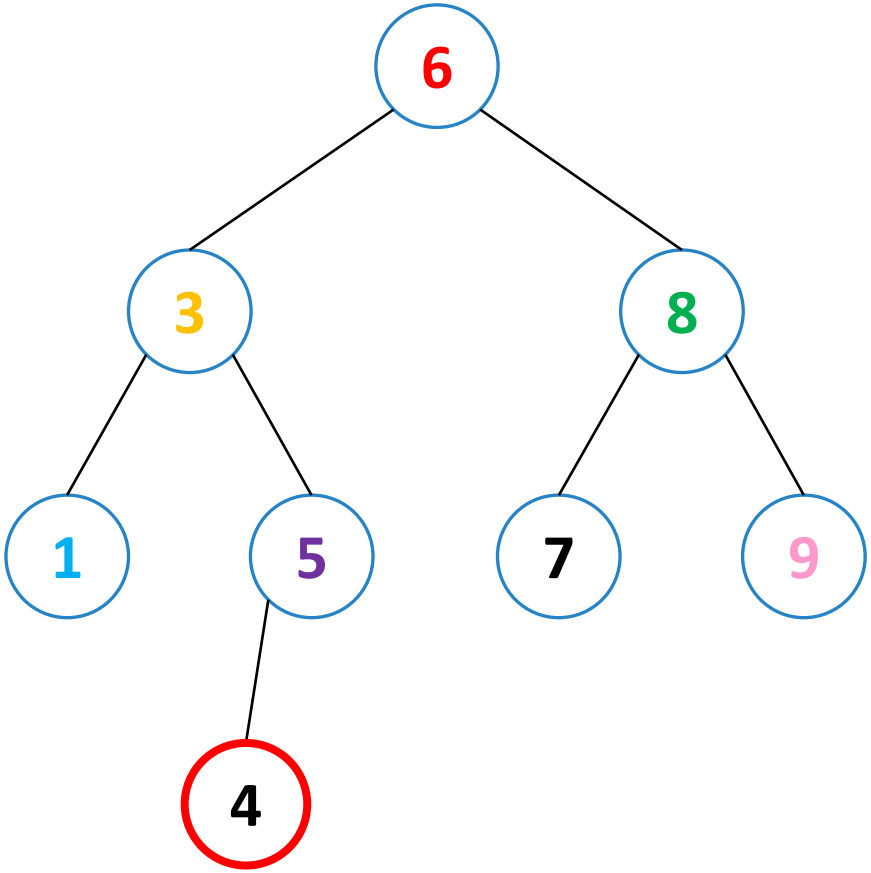
Queremos eliminar un dato (una *key* y su *value*) del árbol

Si el dato está en una hoja, o tiene un solo hijo, eliminarlo es trivial

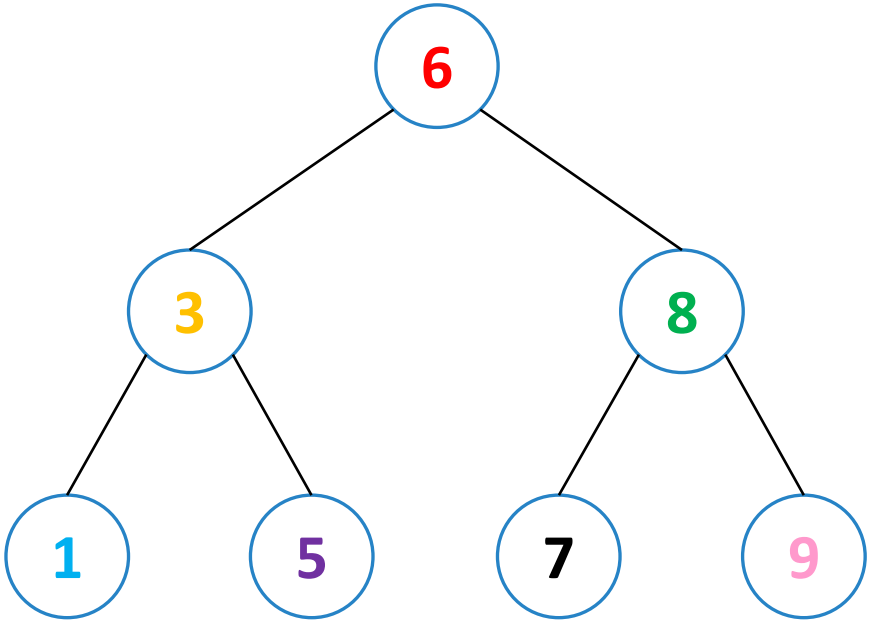
Si no, ¿cómo podemos eliminarlo sin romper la estructura?

¿Podremos reemplazarlo por otro nodo del árbol? ¿Cuál?

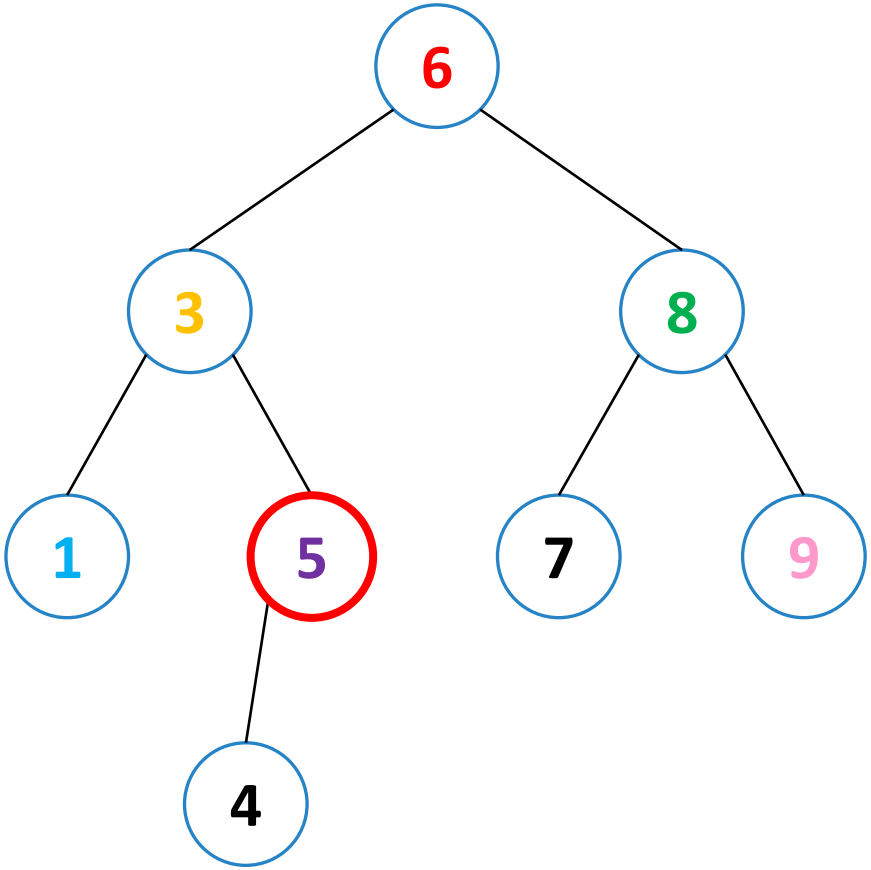
Eliminemos el 4



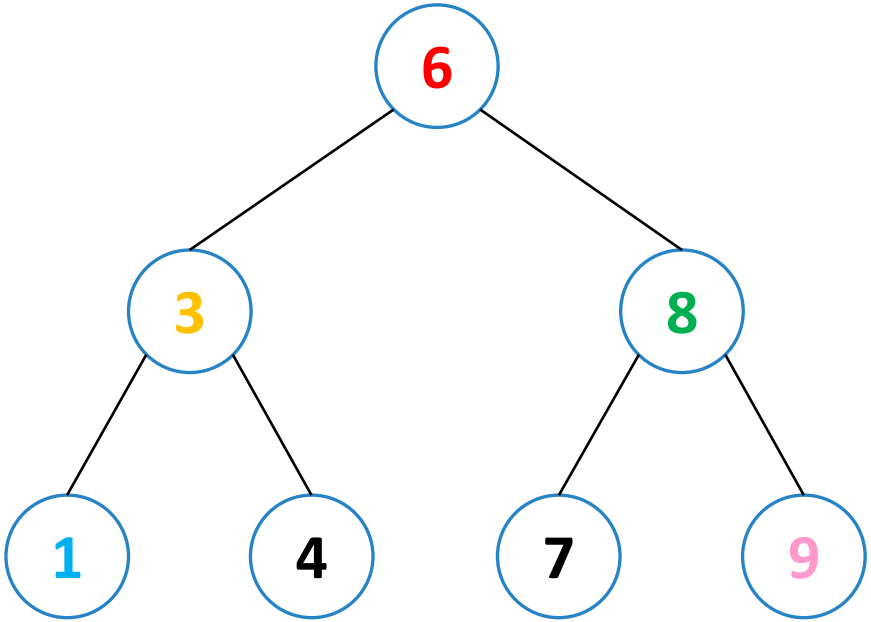
Eliminemos el 4



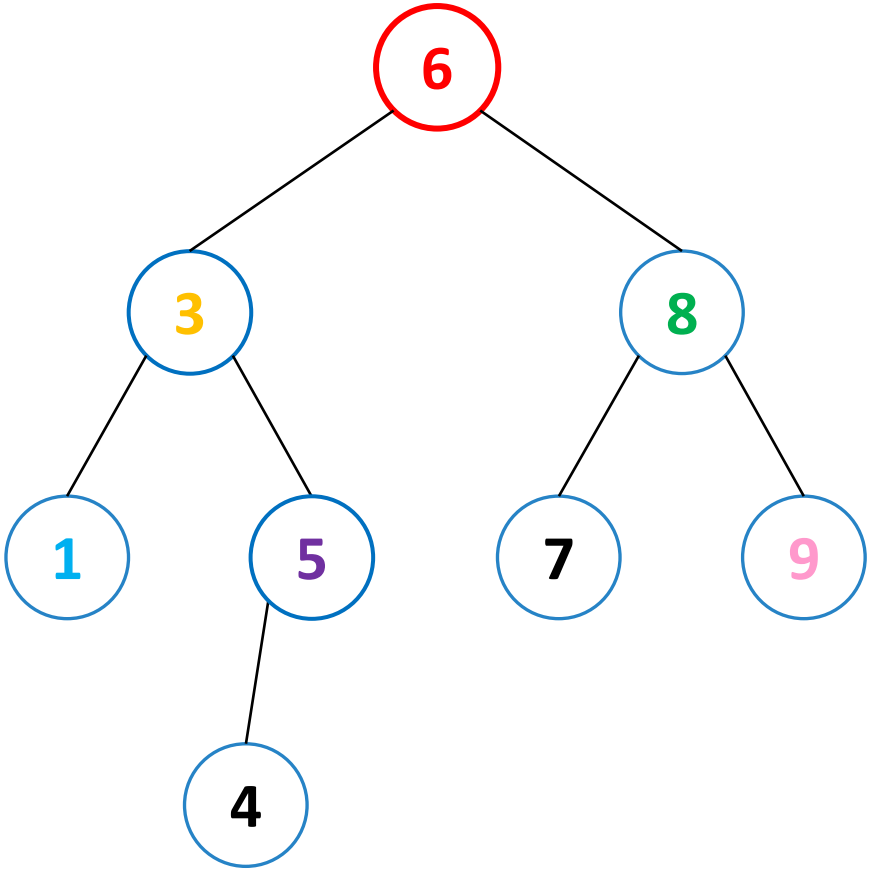
Eliminemos el 5



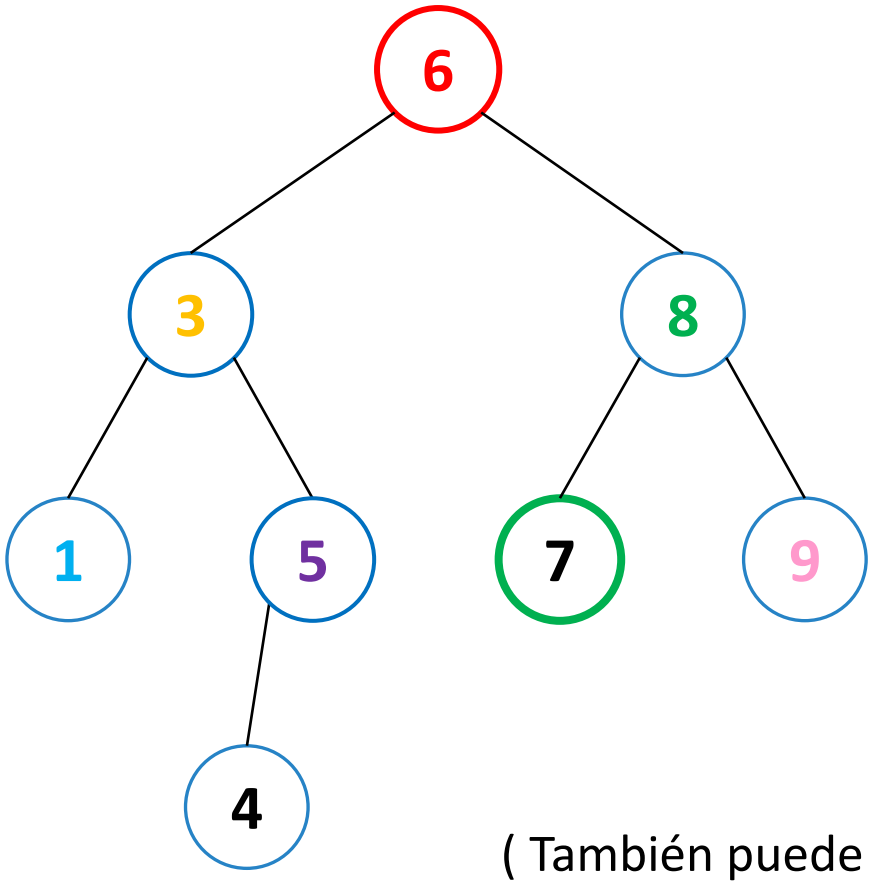
Eliminemos el 5



Ahora el 6… no es tan fácil

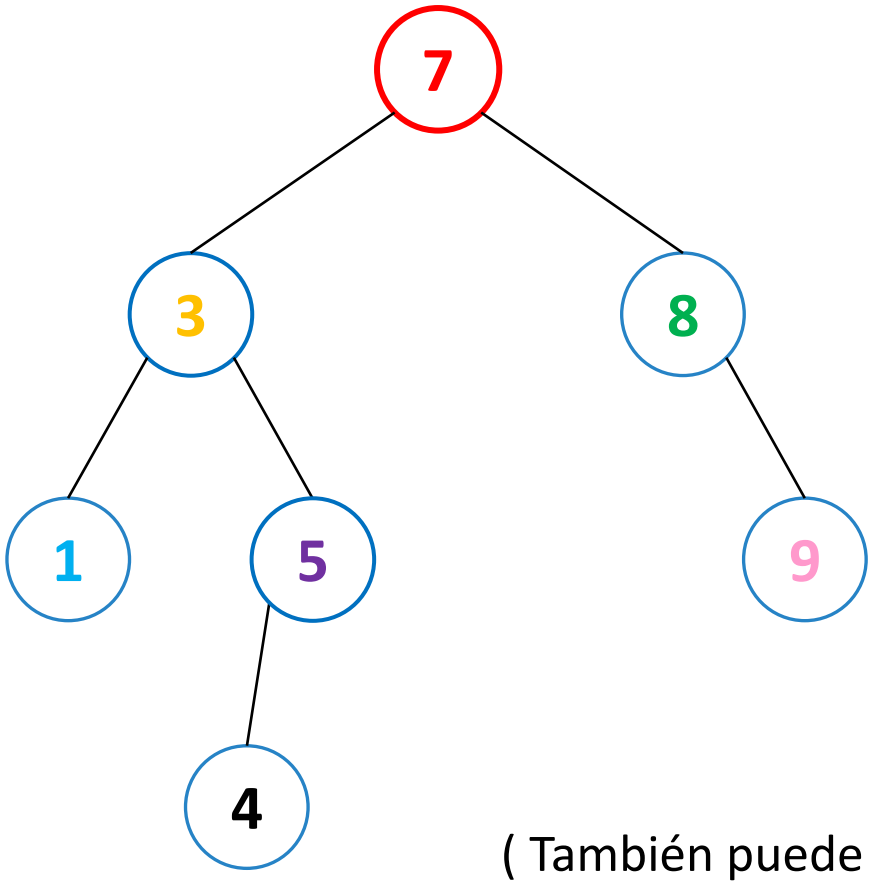


Se remplaza por el sucesor



(También puede ser el antecesor)

Se remplaza por el sucesor



(También puede ser el antecesor)

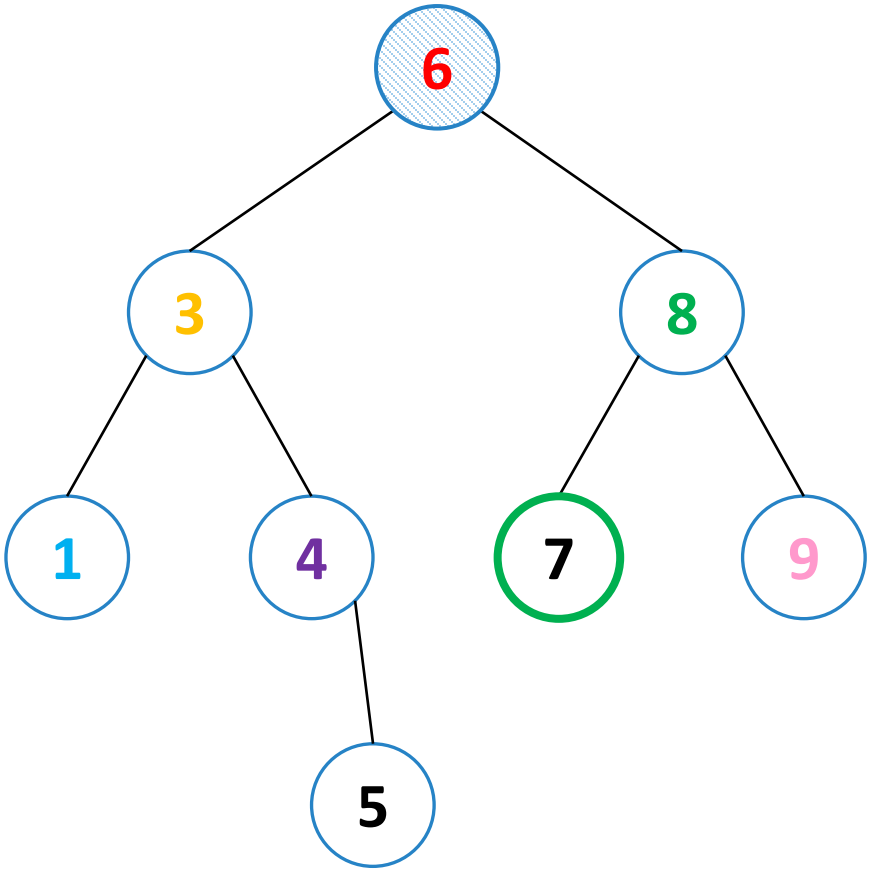
Antecesor y sucesor

Si los nodos estuvieran ordenados en una lista según su key:

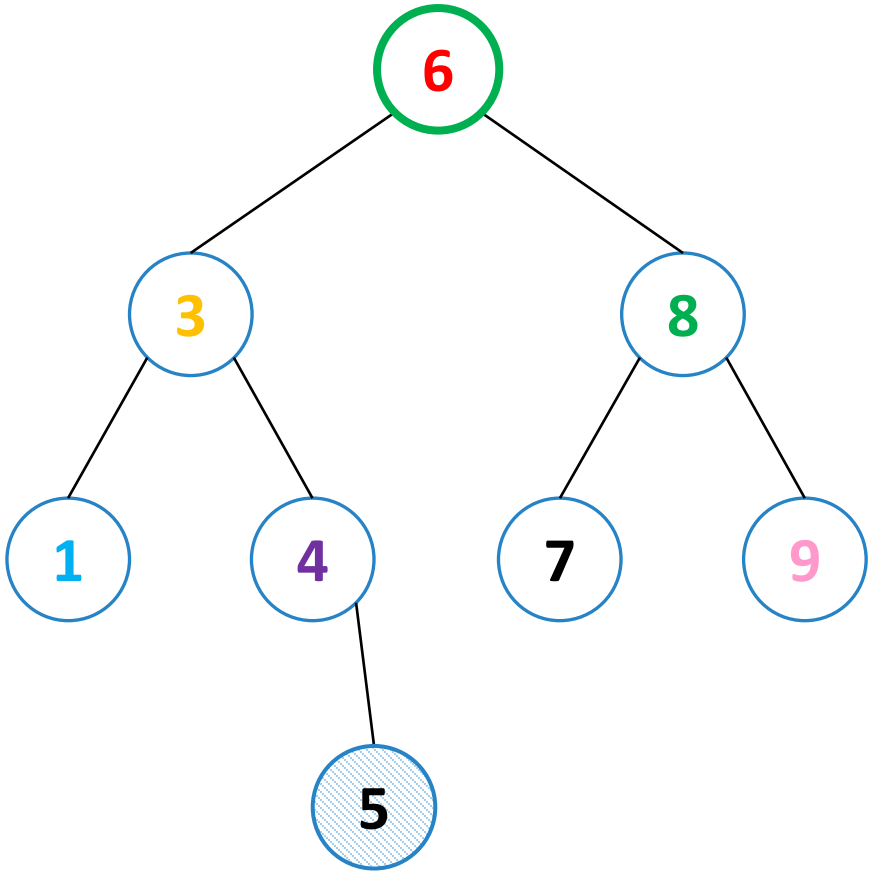
- El **sucesor** de un nodo es el siguiente en la lista
- El **antecesor** de un nodo es el anterior en la lista

¿Cómo podemos encontrar estos elementos dentro del árbol?

Busquemos el sucesor del 6



Busquemos el sucesor del 5



Ya no es tan sencillo… pero no lo necesitamos para eliminar

```
min(A):
    if A.left = ∅:
        return A
    else:
        return min(A.left)
```

Este pseudo código es válido sólo en el caso en que buscamos el sucesor de un nodo *A* que tiene dos hijos

```
succesor(A):
    if A.right ≠ ∅:
        return min(A.right)
    return ∅
```

$delete(A, k)$:

$D = search(A, k)$

if D *es hoja*:

$D = \emptyset$

else if D *tiene un solo hijo* H:

$D = H$

else:

$S = sucessor(D)$

$D.k = S.k$

$delete(S, S.k)$

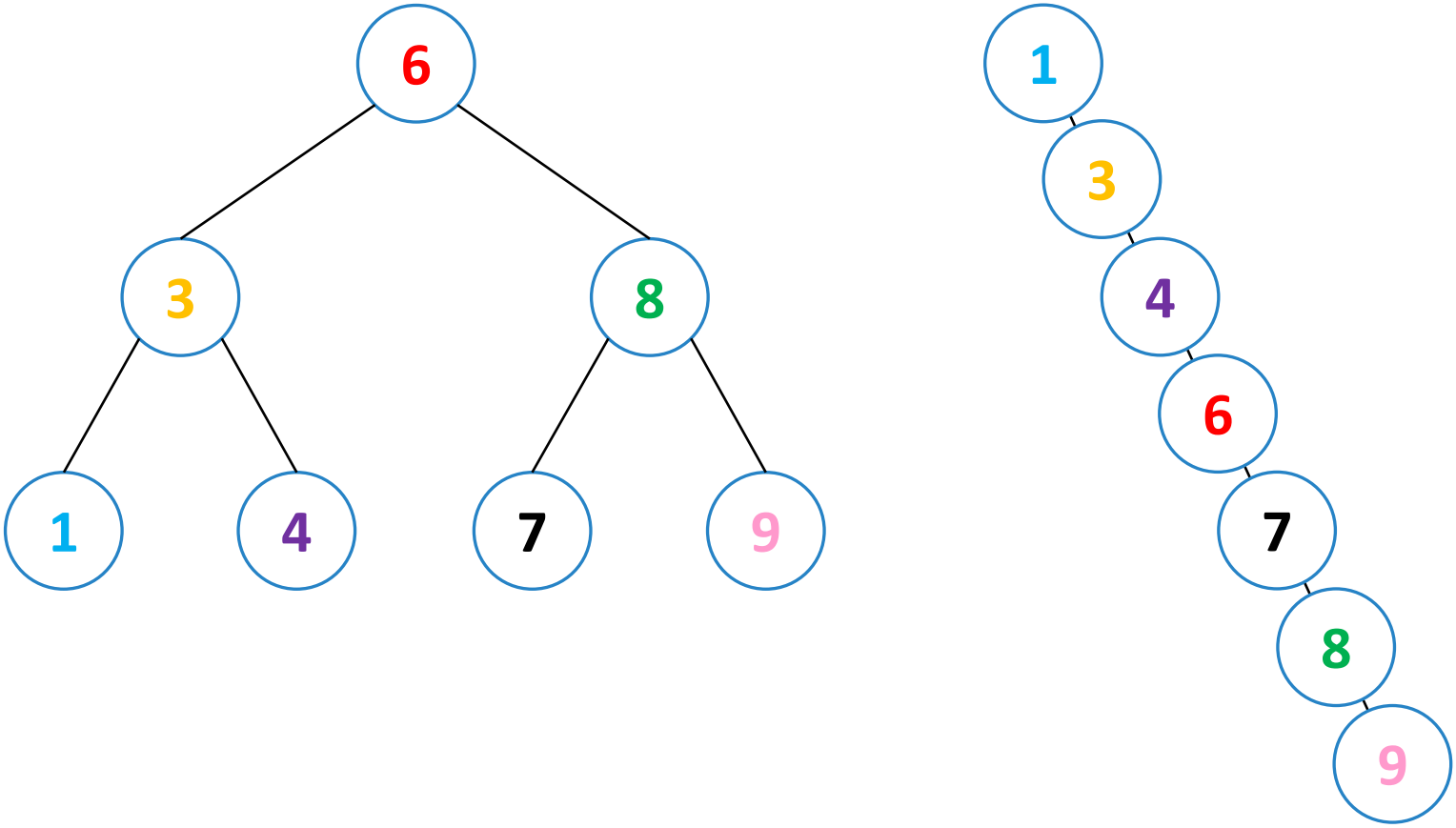
El bueno, el malo y el feo

¿Hay algunas raíces más convenientes que otras?

¿Qué pasa con el árbol si no queda un dato conveniente como raíz?

¿Cómo varía la complejidad de las operaciones?

Mismos datos, distinto árbol



Debemos intentar que el árbol sea lo más balanceado posible