Árboles de búsqueda

Yadran Eterovic

Los **árboles de búsqueda** permiten realizar las operaciones de diccionario —*insertar*, *buscar* y *eliminar*— en tiempo O(log *n*) en promedio

… y, además, *mantienen la información ordenada*

Para ello, combinan dos ventajas de estructuras de datos más simples:

- la flexibilidad de la inserción en una lista ligada
- la eficiencia de la búsqueda en un arreglo ordenado

Desde el punto de vista de implementación, son estructuras hechas de *nodos* que contienen *links* que son referencias a otros nodos

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

… excepto en el caso del nodo *raíz* del árbol, al que no apunta ningún otro nodo

… y cada nodo *x* tiene exactamente 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

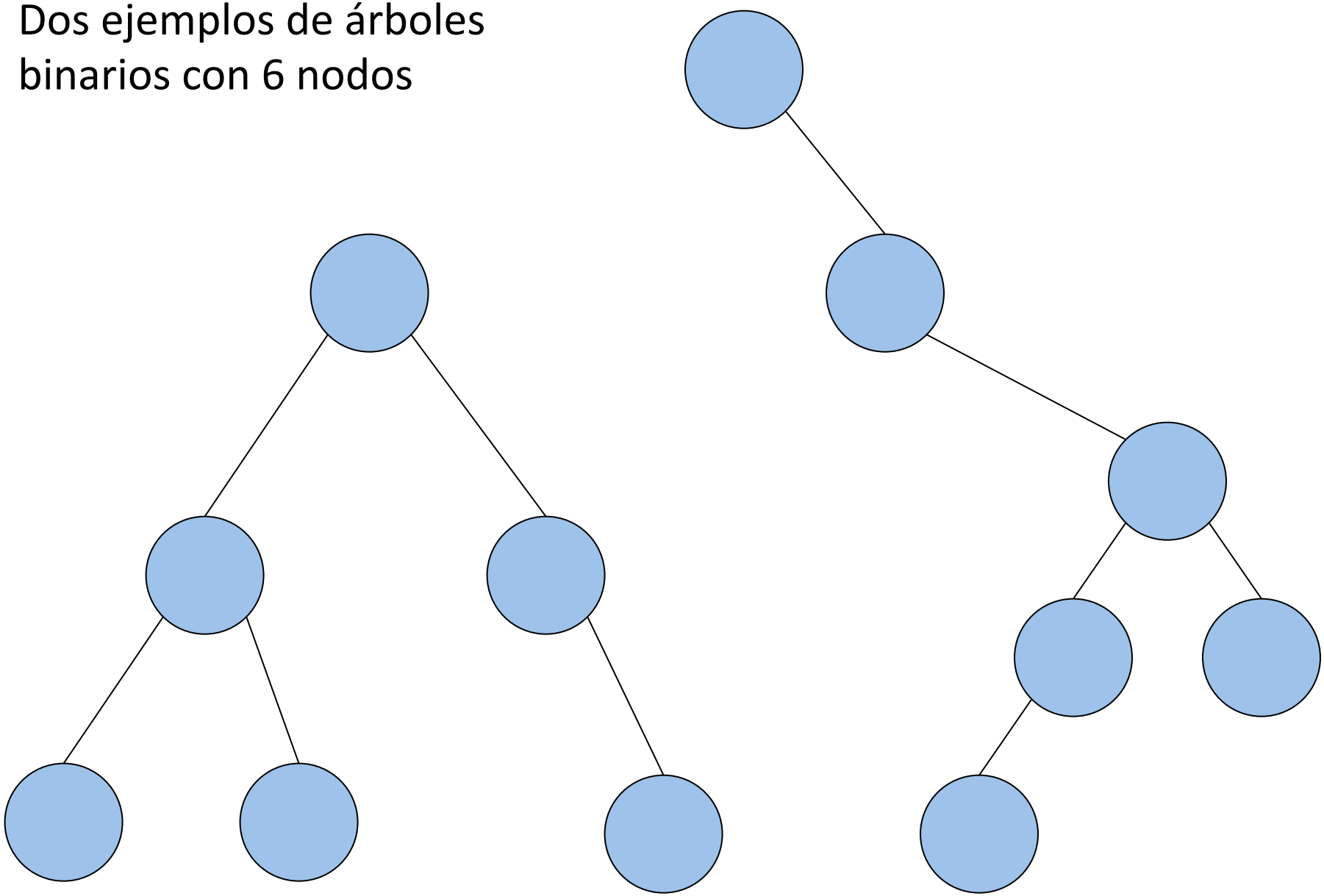
Así, podemos definir un **árbol binario** (general, no necesaria-mente de búsqueda) como

… ya sea un link nulo

… o bien un nodo con un link izquierdo y un link derecho, ambos referencias a subárboles (disjuntos) que a su vez son árboles binarios:

el subárbol izquierdo y el subárbol derecho

Dos ejemplos de árboles binarios con 6 nodos



En un **árbol binario de búsqueda** (ABB), cada nodo tiene además una clave (*key*) e información asociada:

- las claves son *comparables* (≤)

Los nodos en un ABB están almacenados de modo que sus claves cumplen la **propiedad de ABB**:

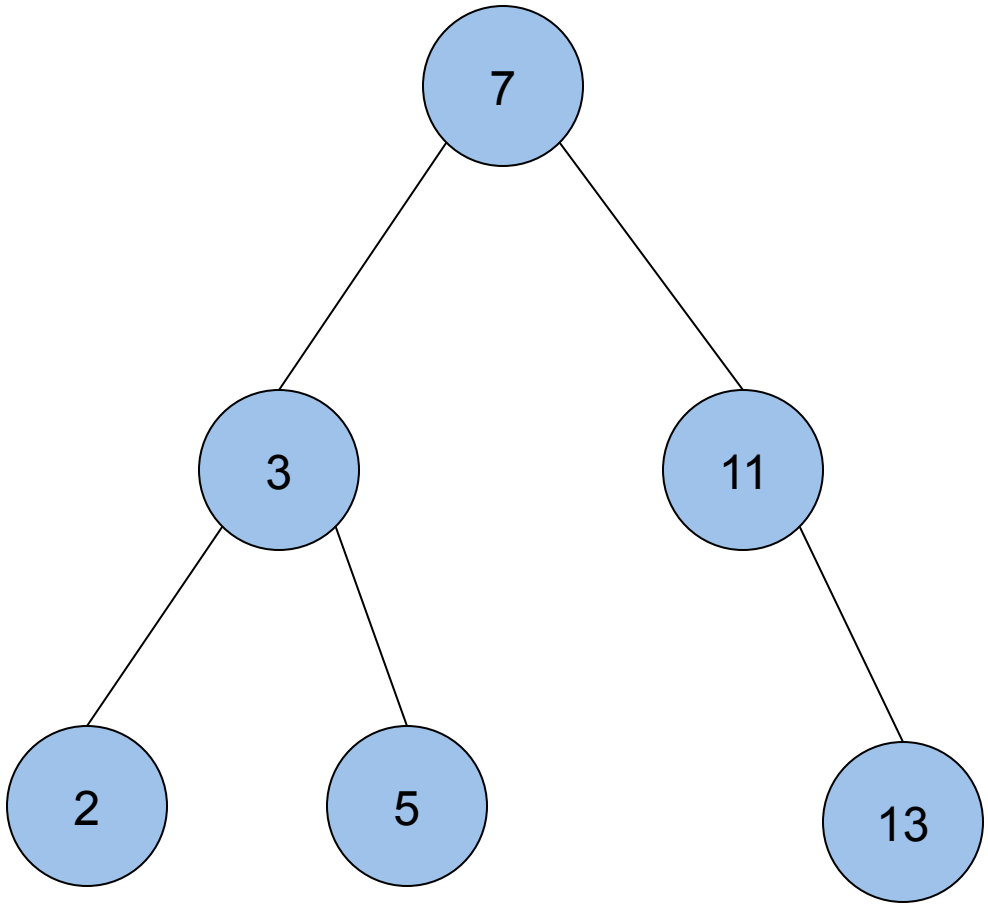
sea *x* un nodo en un ABB

si *y* es un nodo en el subárbol izquierdo de *x*, entonces $y.key \leq x.key$

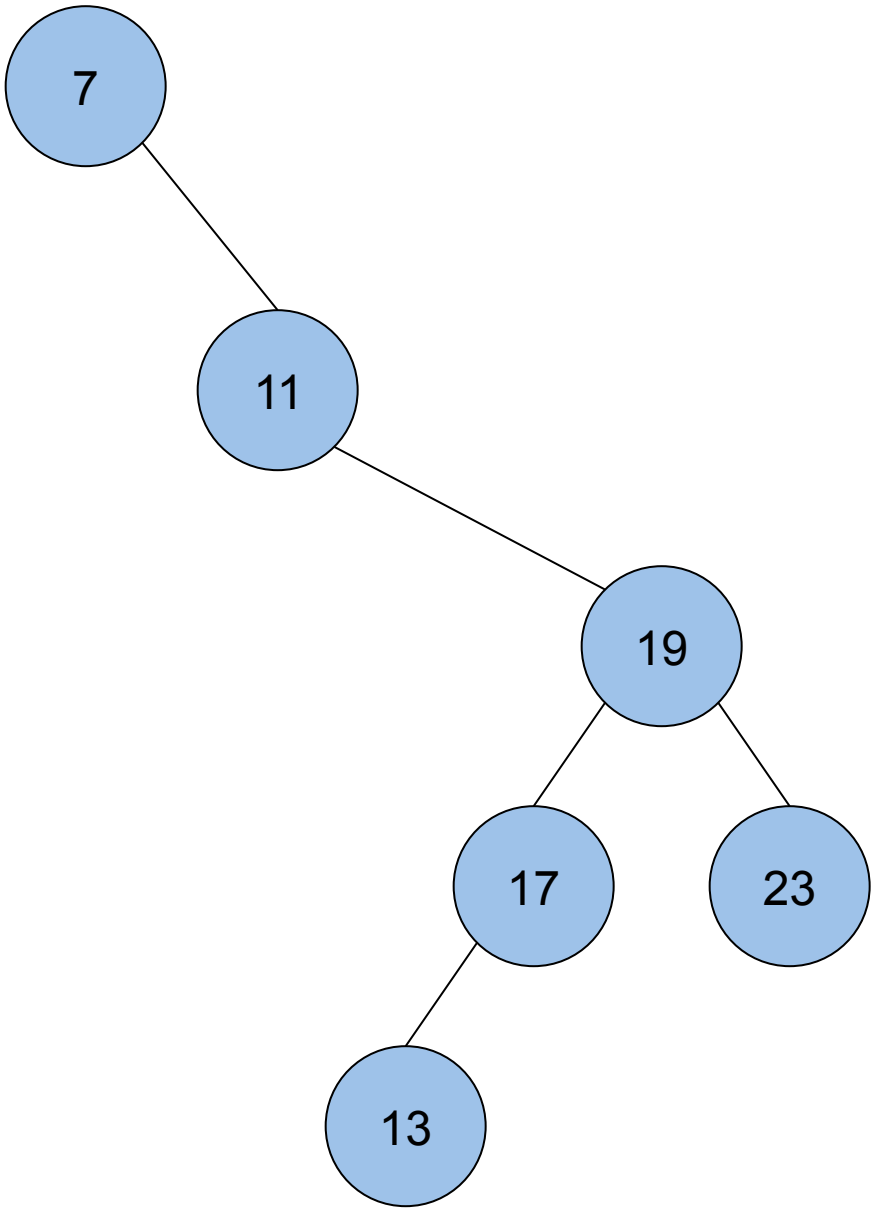
si *z* es un nodo en el subárbol derecho de *x*, entonces $x.key \leq z.key$

Un ABB cuyos nodos contienen las claves 7, 3, 11, 2, 5 y 13

Decimos que
“7 es padre de 11”
“3 es hijo izquierdo de 7”,
“13 es hijo derecho de 11”



Otro ABB con las claves
7, 11, 19, 17, 23 y 13



La propiedad de ABB permite imprimir las claves del árbol en orden, de menor a mayor:

```
inorder(x):
    if x != null:
        inorder(x.left())
        print(x.key())
        inorder(x.right())
```

Este procedimiento recursivo —hay una versión iterativa equivalente— recorre un árbol de n nodos en tiempo $\Theta(n)$

Un ABB permite insertar, eliminar y buscar objetos, por sus claves —**diccionario** …

… y, también, encontrar el objeto con la clave mínima (o máxima) —**cola de prioridades**

… e incluso el objeto con la clave sucesora (o predecesora) de una clave dada

Todas estas operaciones toman tiempo proporcional a la *altura del árbol* —el número máximo de niveles entre la raíz y una hoja árbol,

… o la longitud de la rama más larga del árbol:

la altura esperada de un ABB con n objetos construido aleatoriamente es $O(\log n)$

… aunque en general podría ser $O(n)$

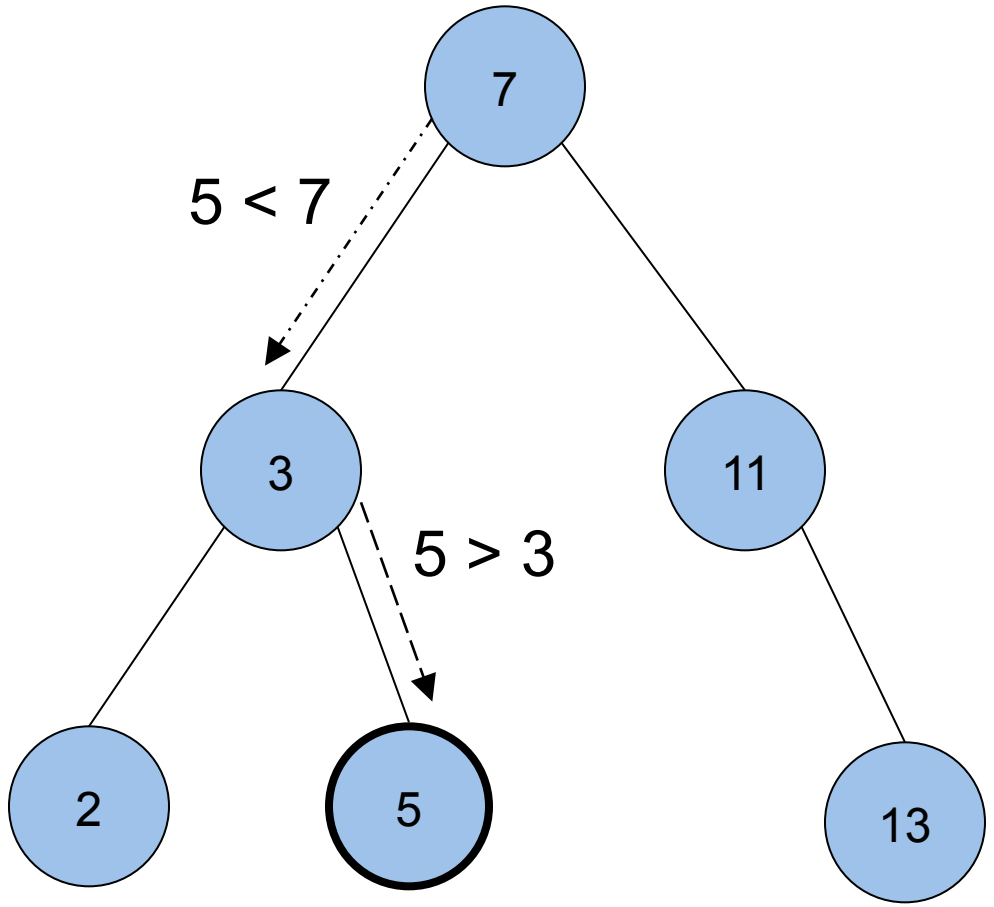
Cuando **buscamos una clave** *k* a partir del nodo *x* del árbol, el algoritmo de búsqueda hace uso de la propiedad de ABB:

- si encontramos la clave, devolvemos el nodo
- … de lo contrario, devolvemos *null*

```
find(x, k):
   if x == null || k == x.key():
      return x
   if k < x.key():
      return find(x.left(), k)
   else:
      return find(x.right(), k)
```

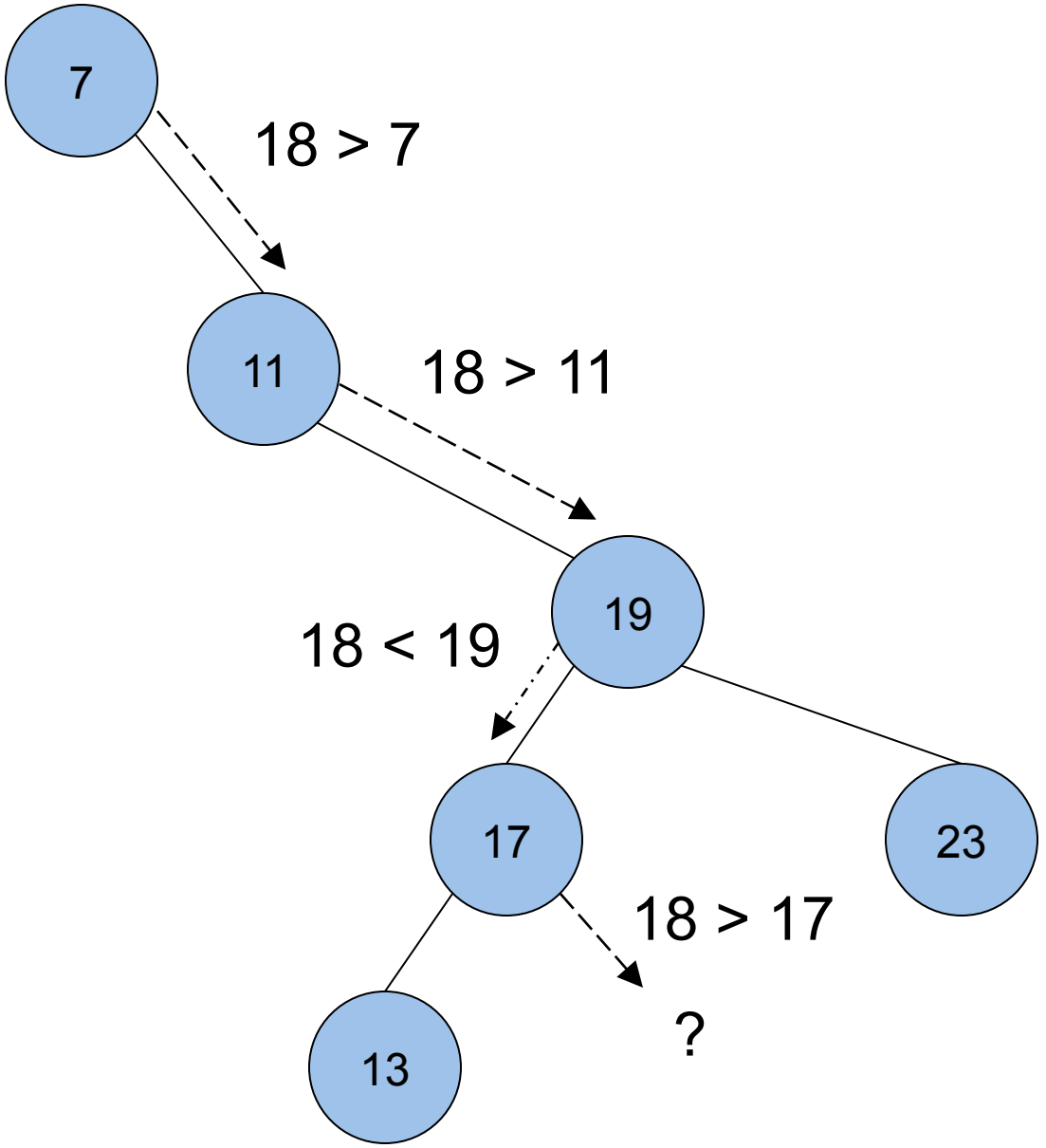
P.ej., si en el árbol de la derecha buscamos la clave 5 a partir de la raíz del árbol:

- comparamos 5 con 7, la clave en *x*; como 5 < 7, vamos al hijo izquierdo de *x*
- comparamos 5 con 3, la clave en el nuevo *x*; como 5 > 3, vamos al hijo derecho de *x*
- comparamos 5 con 5, la clave en el nuevo *x*; como son iguales, devolvemos este *x*



Similarmente, si en el árbol de la derecha buscamos la clave 18 a partir de la raíz:

- comparamos 18 sucesiva-mente con 7, 11, 19 y 17
- finalmente, como 18 > 17, vamos al hijo derecho de este nodo con clave 17
- pero este último nodo no existe
- esto significa que la clave 18 **no está** almacenada en el árbol y por lo tanto devolvemos *null*



Cuando **buscamos el nodo con la clave mínima** en el árbol, a partir del nodo *x*, por la propiedad de ABB sabemos que es el nodo descendiente de *x* que está "más a la izquierda":

```
min(x):
   while x.left() != null:
      x = x.left()
   return x
```

P.ej., si buscamos el nodo con la clave mínima a partir de la raíz en los árboles anteriores:

- en el árbol de la diap. #16, vamos de la raíz al nodo con clave 3, y de éste al nodo con clave 2

  … como este nodo no tiene hijo izquierdo, éste es el nodo con la clave mínima

- en el árbol de la diap. #17, la raíz no tiene hijo izquierdo

  … por lo tanto, la propia raíz es el nodo con la clave mínima

Cuando **buscamos el nodo sucesor** del nodo *x* en el árbol, por definición es el nodo con la clave más pequeña que sea más grande que la clave de *x*:

p.ej., en la diap. #17, el sucesor del nodo (con clave) 11 es el nodo 13

… y el sucesor del nodo 17 es el nodo 19

```
succesor(x):
   if x.right() != null:
      return min(x.right())
   y = x.p()
   while y != null && x == y.right():
      x = y
      y = y.p()
   return y
```

Al **insertar un nodo** *z* con una clave *k* en un árbol *t*,

… primero buscamos un nodo con clave *k* en *t*

… haciendo la misma búsqueda que haría `find`

… y luego, cuando determinamos que el nodo buscado no está en *t*,

… insertamos *z* en el lugar en que el nodo buscado debería haber estado si hubiera estado

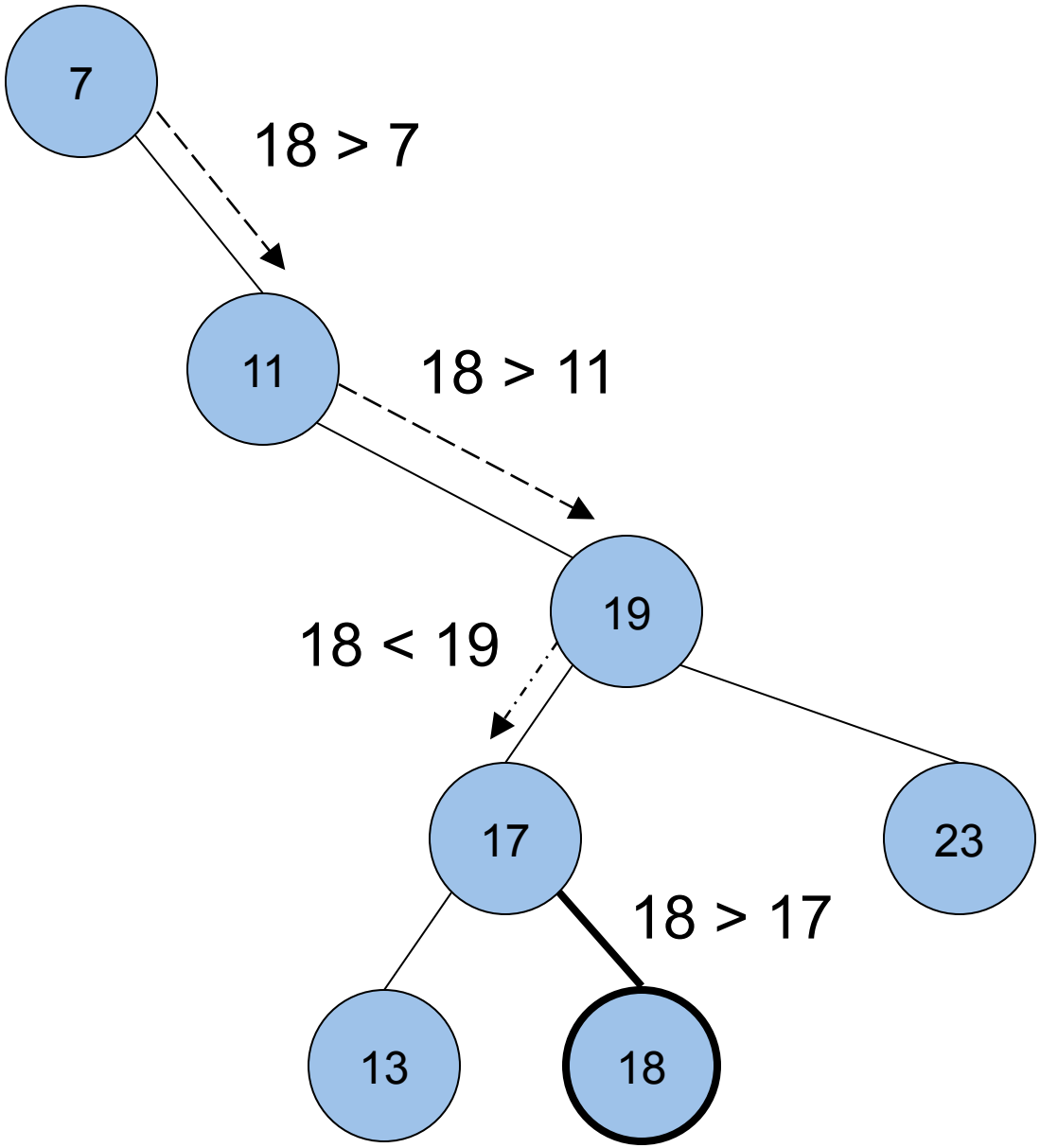
Insertamos el nodo *z* en el árbol, a partir del nodo *x*:

```
z.left() = z.right() = null

insert(z, x):
   if x == null:
      x = z
   else:
      if z.key() < x.key():
         insert(z, x.left())
      else:
         insert(z, x.right())
```

P.ej., al insertar un nodo con clave 18 en el árbol de la derecha,

- … primero hacemos la misma búsqueda sugerida anteriormente
- … y finalmente lo inserta-mos como hijo derecho del nodo con clave 17



Al **eliminar un nodo** *z*, distinguimos los casos en que *z* es una hoja, *z* tiene un hijo, o *z* tiene dos hijos

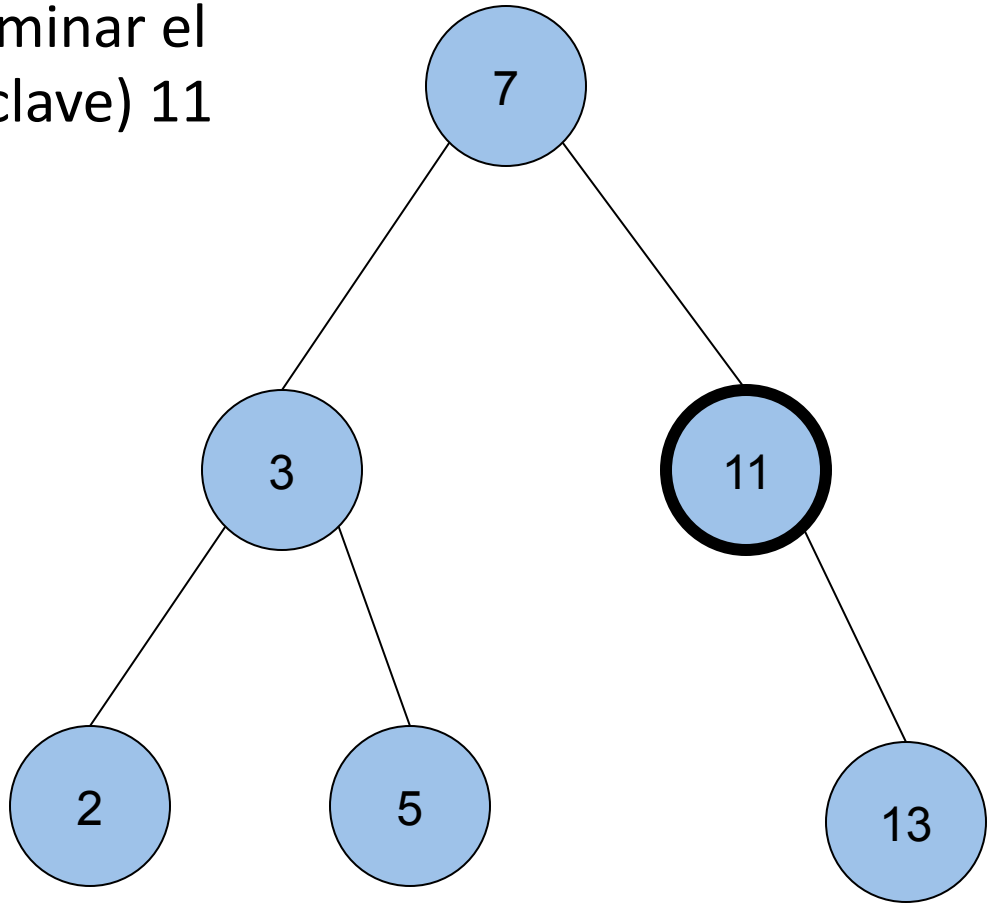
Si *z* es una hoja, entonces simplemente lo eliminamos

Si *z* tiene sólo un hijo, *w*, eliminamos *z* por la via de hacer que *w* se convierta en hijo del padre de *z* (ver diaps. #25 y 26)

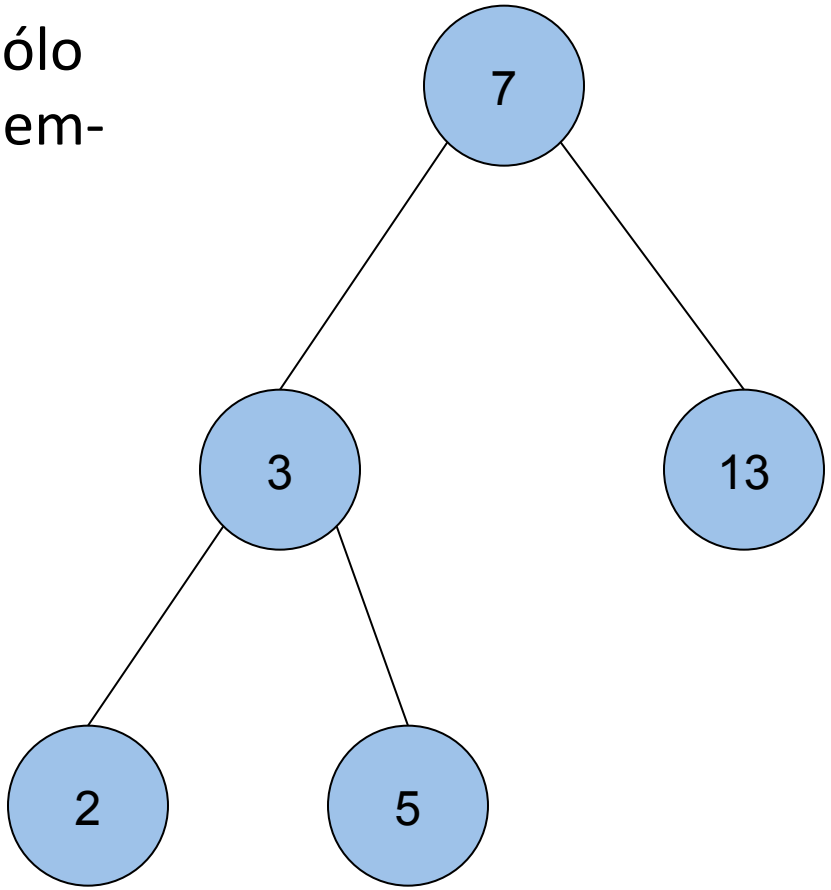
Si *z* tiene dos hijos, lo reemplazamos por su sucesor, *w*, y luego eliminamos *w* (ver diaps. #27, 28 y 29)

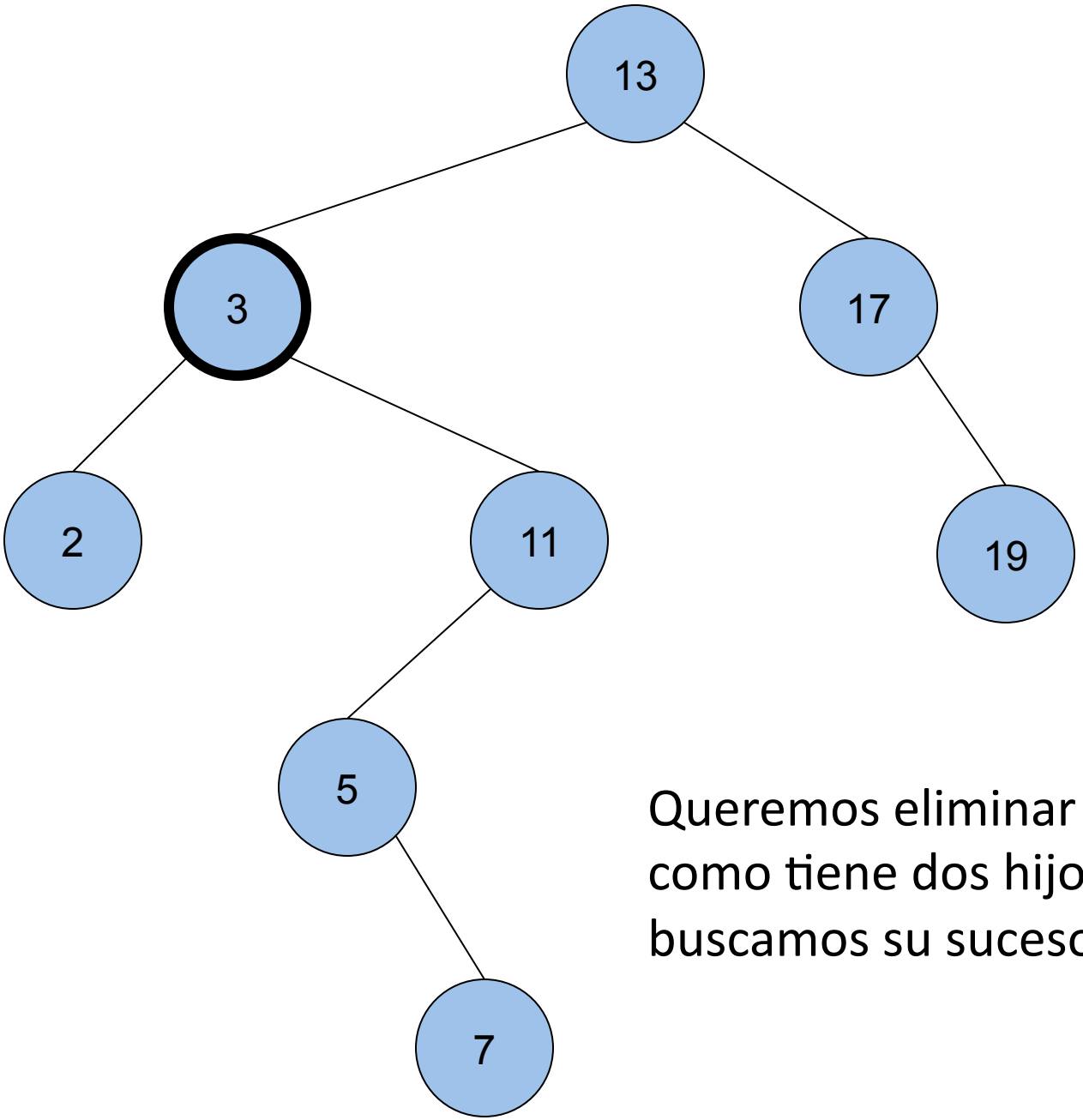
El algoritmo aparece en diap. #30

Queremos eliminar el nodo (con la clave) 11

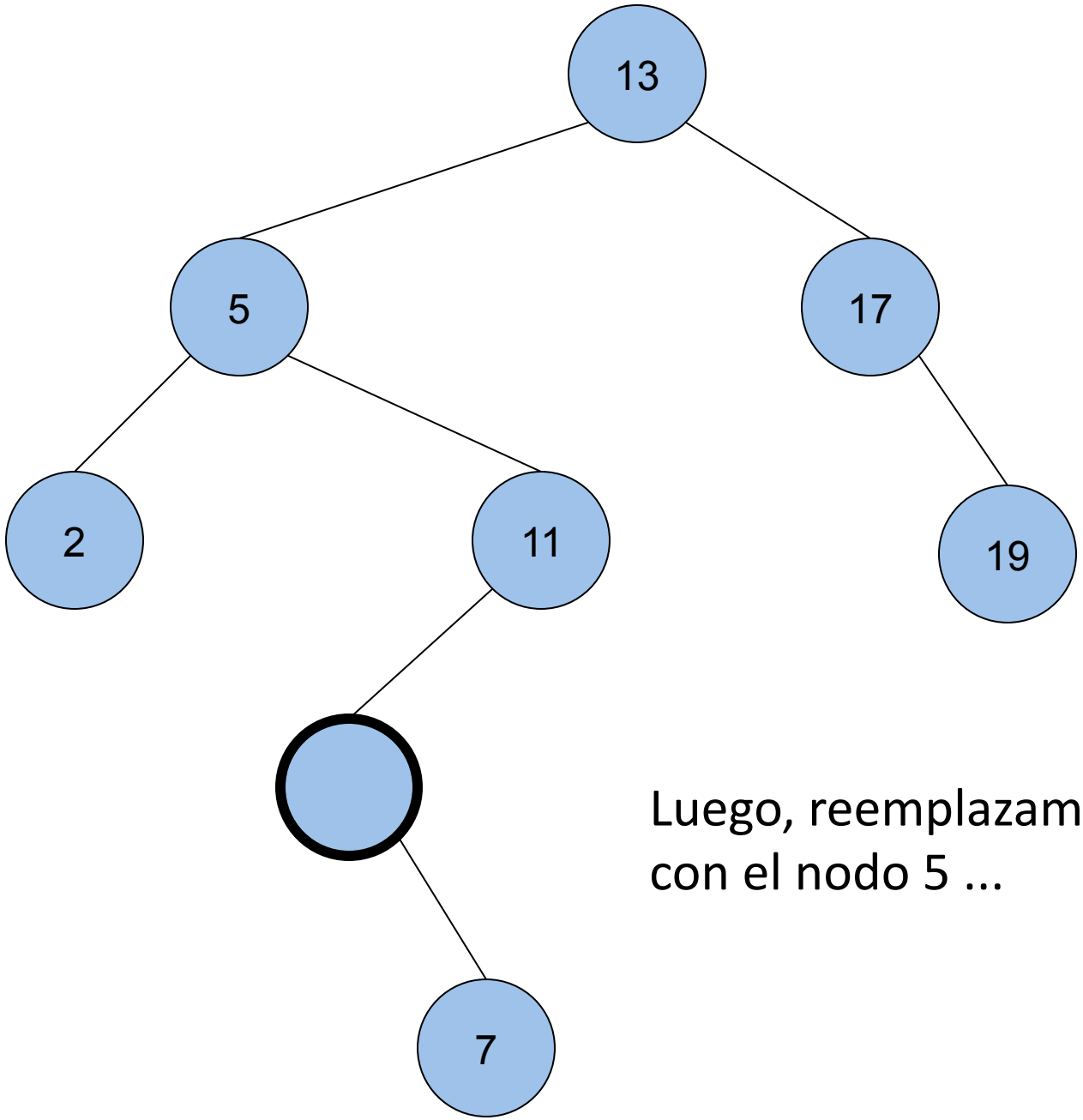


Como el nodo 11 tiene sólo un hijo, el nodo 13, lo reemplazamos por su hijo

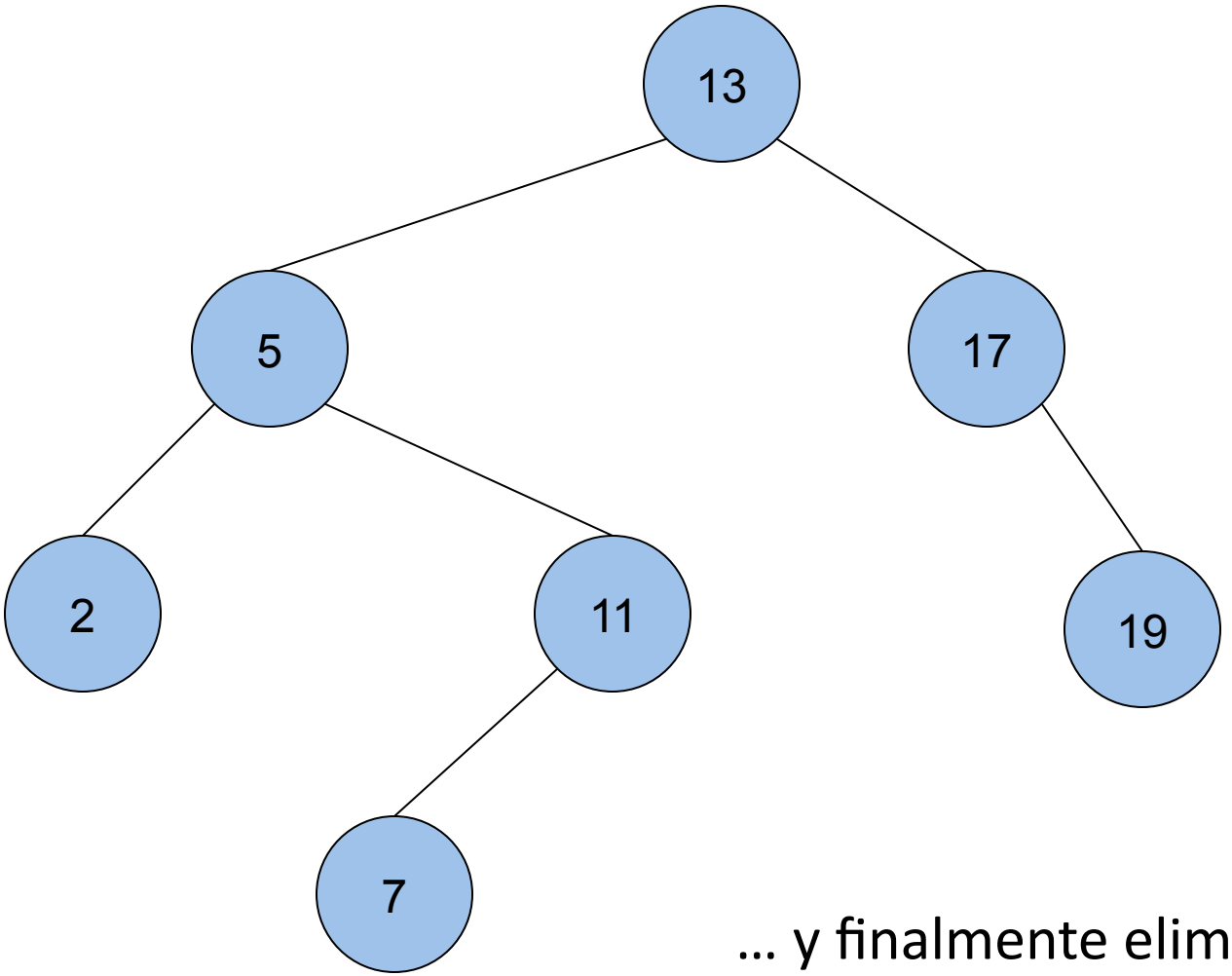




Queremos eliminar el nodo 3; como tiene dos hijos, primero buscamos su sucesor, el nodo 5



Luego, reemplazamos el nodo 3 con el nodo 5 ...



… y finalmente eliminamos el nodo 5

```
void delete(z):
   if z.left() == null && z.right() == null:
      z == null
   else:
      w = null
      if z.left() == null:
         w = z.right()
      if z.right() == null:
         w = z.left()
      if w != null:
         z = w
      else:
         w = successor(z)
         z.addKey(w.key())
         delete(w)
```

Para un ABB, podemos garantizar que las operaciones de diccionario —*buscar*, *insertar* y *eliminar*— tomen tiempo O(log*n*) en el peor caso

Para esto, **es necesario mantenerlos balanceados**

La propiedad de balance debe cumplir dos condiciones

- debe asegurar que la profundidad de un árbol con *n* nodos sea O(log*n*)
- debe ser fácil de mantener —la complejidad de esta operación no puede ser mayor que O(log*n*)

P.ej., exigir que los subárboles izquierdo y derecho tengan la misma altura no es práctico

Es una idea simple

… pero la estructura recursiva de los árboles binarios implica que esta idea se aplique a todos los nodos —cada uno es raíz de un ABB

Asegura que la profundidad es logarítmica

… pero es demasiado restrictiva

Estudiaremos dos tipos de árboles de búsqueda —uno binario, el otro no— que mantienen el balance

… a un costo razonable:

- AVL, árbol binario en que el balance está definido como una diferencia entre las alturas de los subárboles izquierdo y derecho de un nodo que no puede superar un cierto valor máximo
- 2-3, árbol (no binario) en que el balance consiste en que todas las hojas están a la misma profundidad (desde la raíz), pero los nodos pueden almacenar una o dos claves y tener dos o tres hijos

Un **árbol AVL** es un ABB que cumple con la *propiedad de balance* AVL:

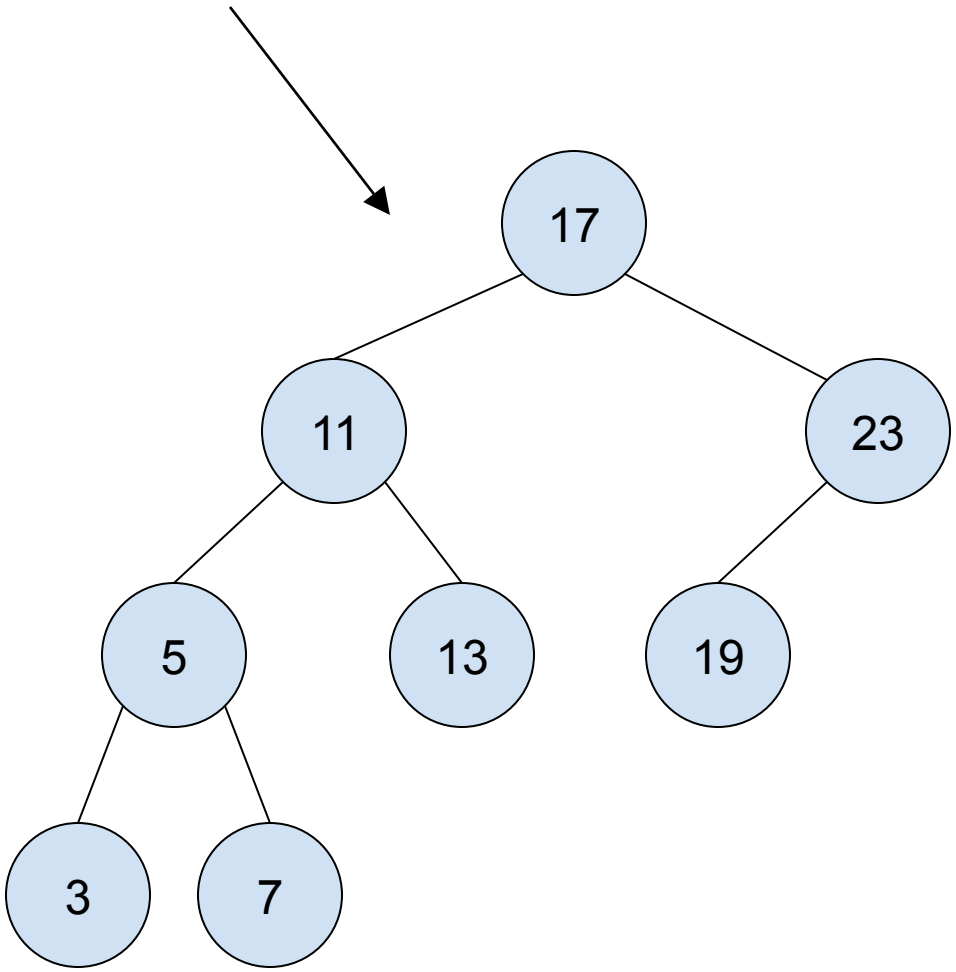
> *Para cualquier nodo del árbol, las alturas de los subárboles izquierdo y derecho pueden diferir a lo más en 1*

Fueron propuestos por G. Adel'son-Vel'skii y E. Landis en 1962

La propiedad de balance AVL implica que el árbol tiene altura logarítmica:

- ver diaps. #53 y 54

Árbol AVL



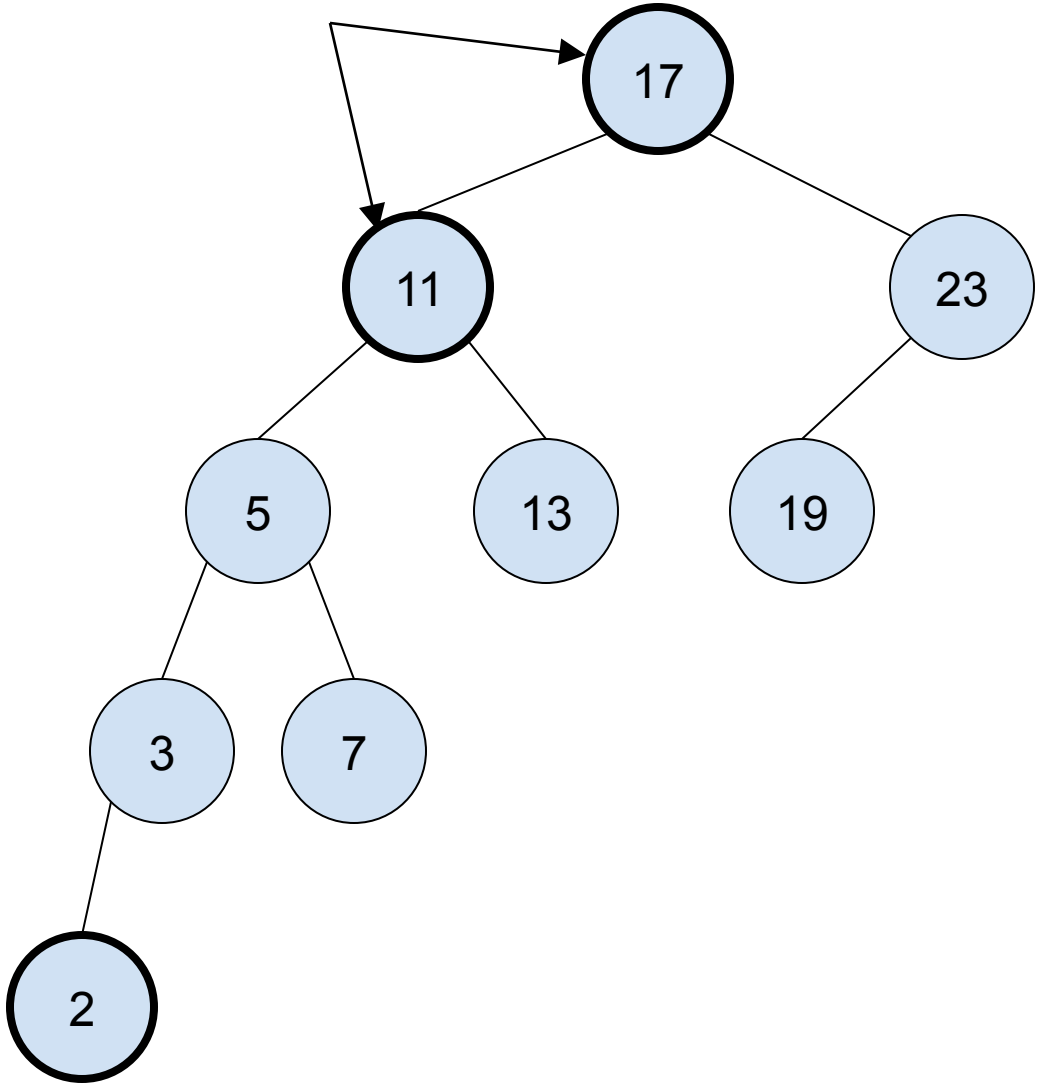
Las operaciones *insert* y *delete* no son tan simples como antes

Una inserción o una eliminación puede destruir el balance de varios nodos:

> … p.ej., en la diapositiva siguiente, la inserción del nodo (con clave) 2 produce desbalance en los nodos 11 y 17

El balance **debe ser restaurado** antes que la operación pueda considerarse completa

Nodos desbalanceados luego de
la inserción del nodo con clave 2



find no cambia

… y ahora es O(log*n*) en el peor caso

Agregamos a cada nodo *x* un **atributo de balance** adicional:

x.balance = –1 / 0 / +1 ,

… dependiendo de si el subárbol izquierdo es más alto, ambos subárboles tienen la misma altura, o si el subárbol derecho es más alto

Con esto, después de una inserción, sólo los nodos que están en la ruta desde el punto de inserción hacia la raíz podrían haber quedado desbalanceados:

- sólo esos nodos tienen sus subárboles modificados

Siguiendo esa ruta, sea *x* el primer nodo (el más profundo) desbalanceado:

- $x.balance < -1$, o $x.balance > +1$
- si rebalanceamos *x*, habremos rebalanceado todo el árbol

Como *x* tiene a los más dos hijos y el desbalance significa que las alturas de los dos subárboles de *x* difieren en 2, hay cuatro casos de inserción posibles:

1) en el subárbol izquierdo del hijo izquierdo de *x*

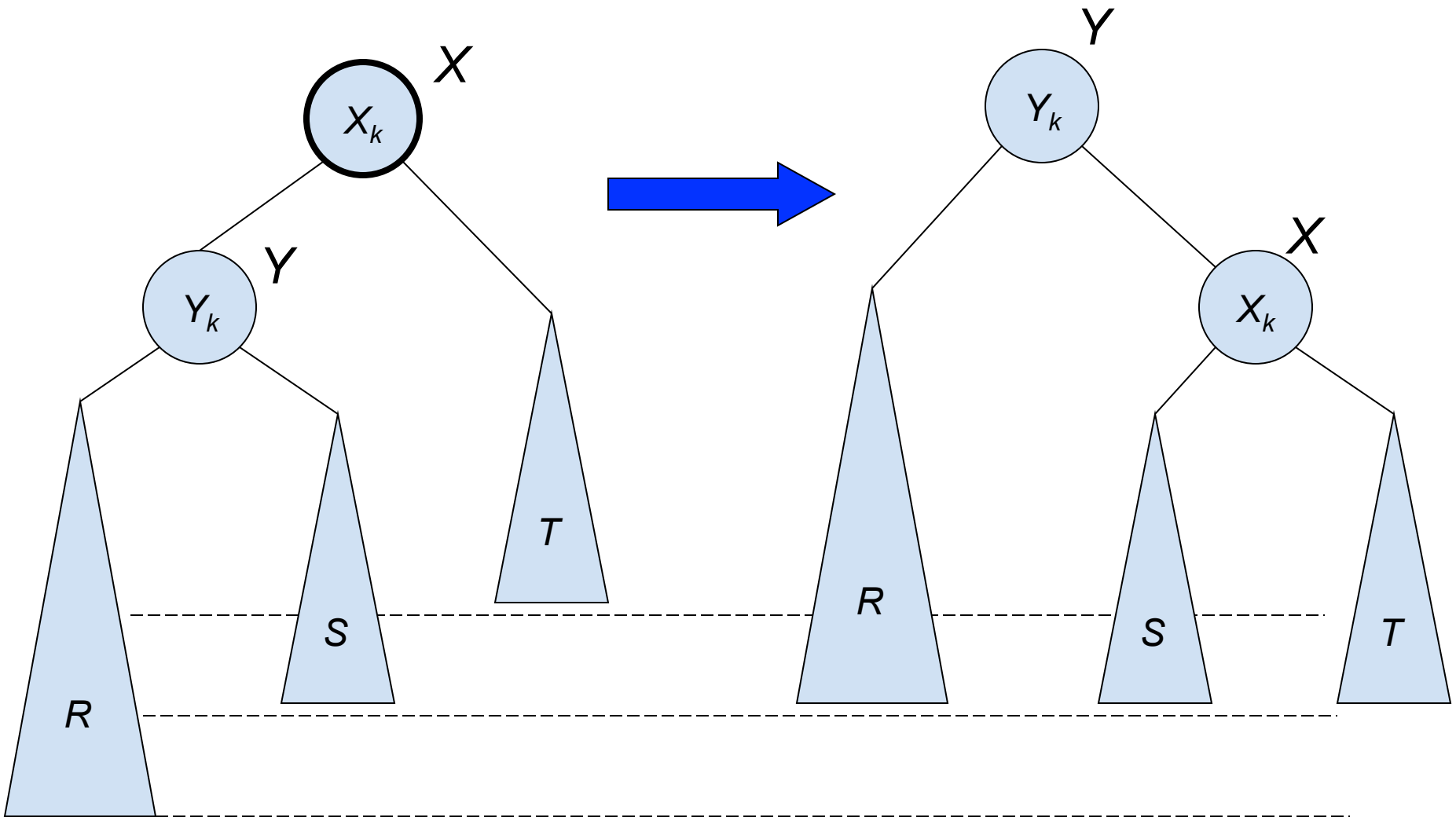
2) en el subárbol derecho del hijo izquierdo de *x*

3) en el subárbol izquierdo del hijo derecho de *x*

4) en el subárbol derecho del hijo derecho de *x*

Los casos de un mismo color (1 y 4, y 2 y 3) son simétricos entre ellos con respecto a *x*

Una **rotación simple** resuelve el caso básico A; p.ej., una *rotación simple a la derecha* en torno a *X*–*Y* para resolver el caso 1:



La **rotación** preserva la propiedad de ABB:

R sigue siendo el subárbol izquierdo del nodo *Y* (clave Y_k)

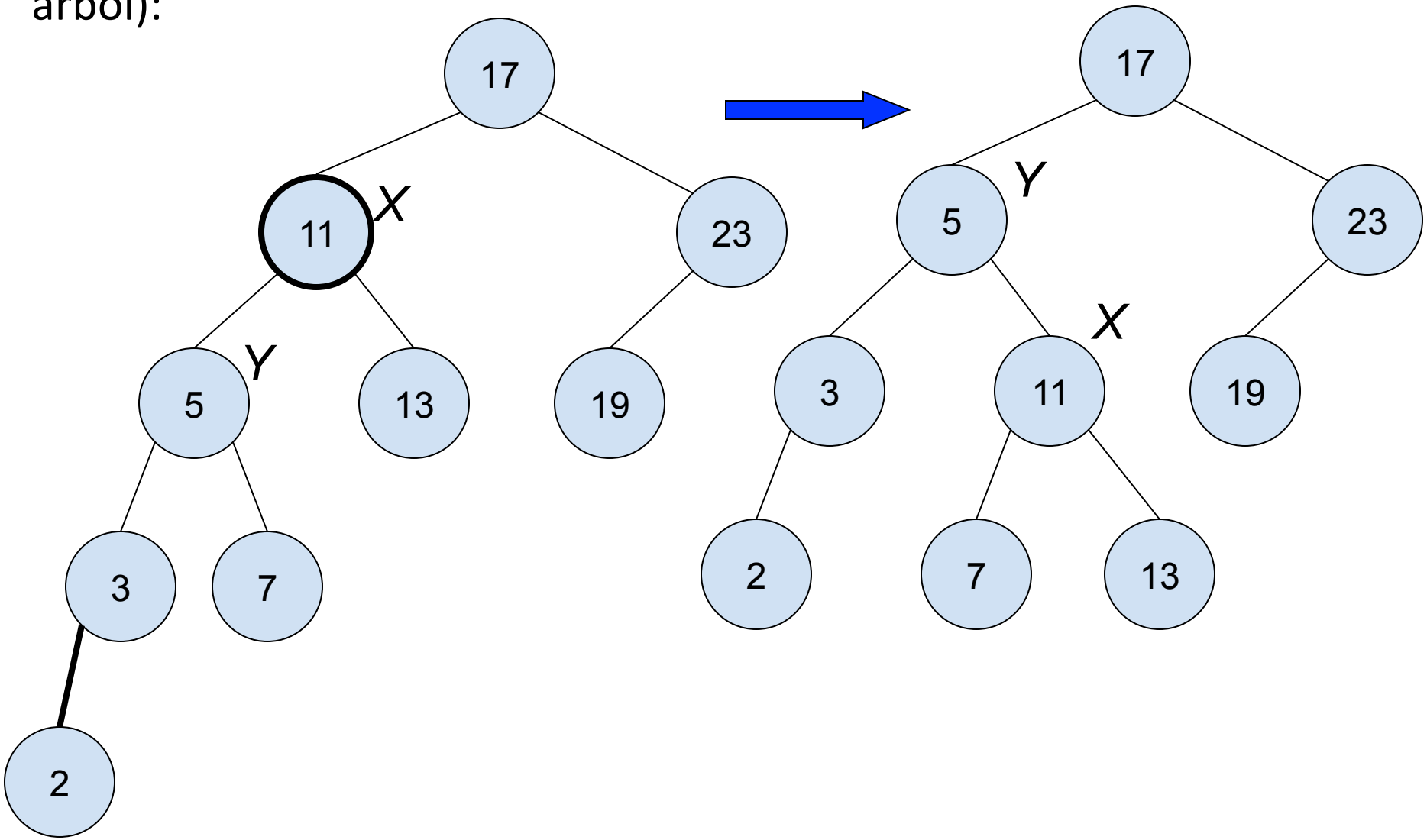
T sigue siendo el subárbol derecho del nodo *X* (clave X_k)

…:

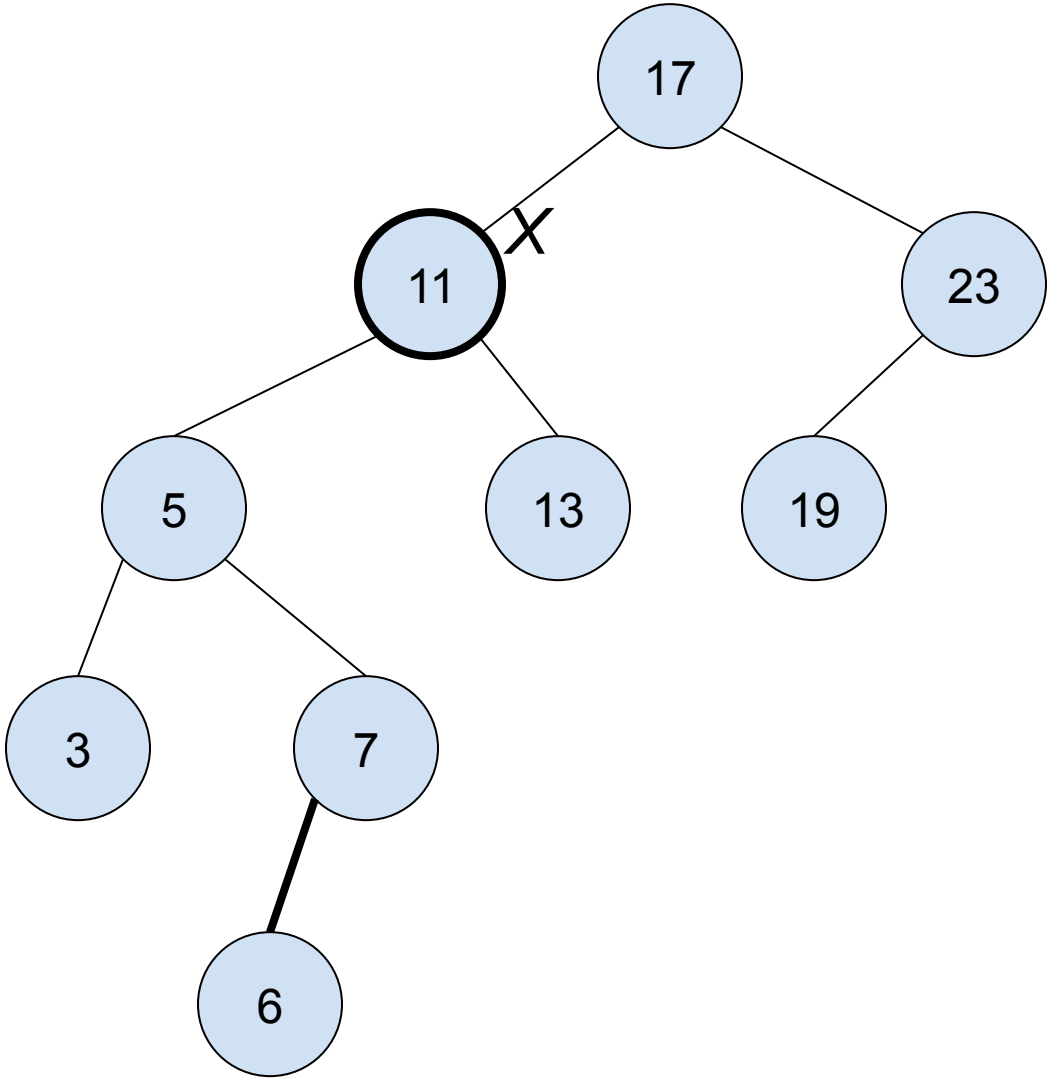
Como en el árbol original Y es el hijo izquierdo de $X \Rightarrow Y_k < X_k$ $\Rightarrow X$ puede ser hijo derecho de Y en el árbol rotado

En el árbol original las claves en S son mayores que Y_k (S es el subárbol derecho de Y) y son menores que X_k (están en el subárbol izquierdo de X) $\Rightarrow S$ puede ser el subárbol izquierdo de X en el árbol rotado

Ejemplo del caso A: El nodo con clave 2 fue insertado en el árbol de la izquierda, rompiendo el balance del nodo *X* (y la propiedad AVL del árbol):



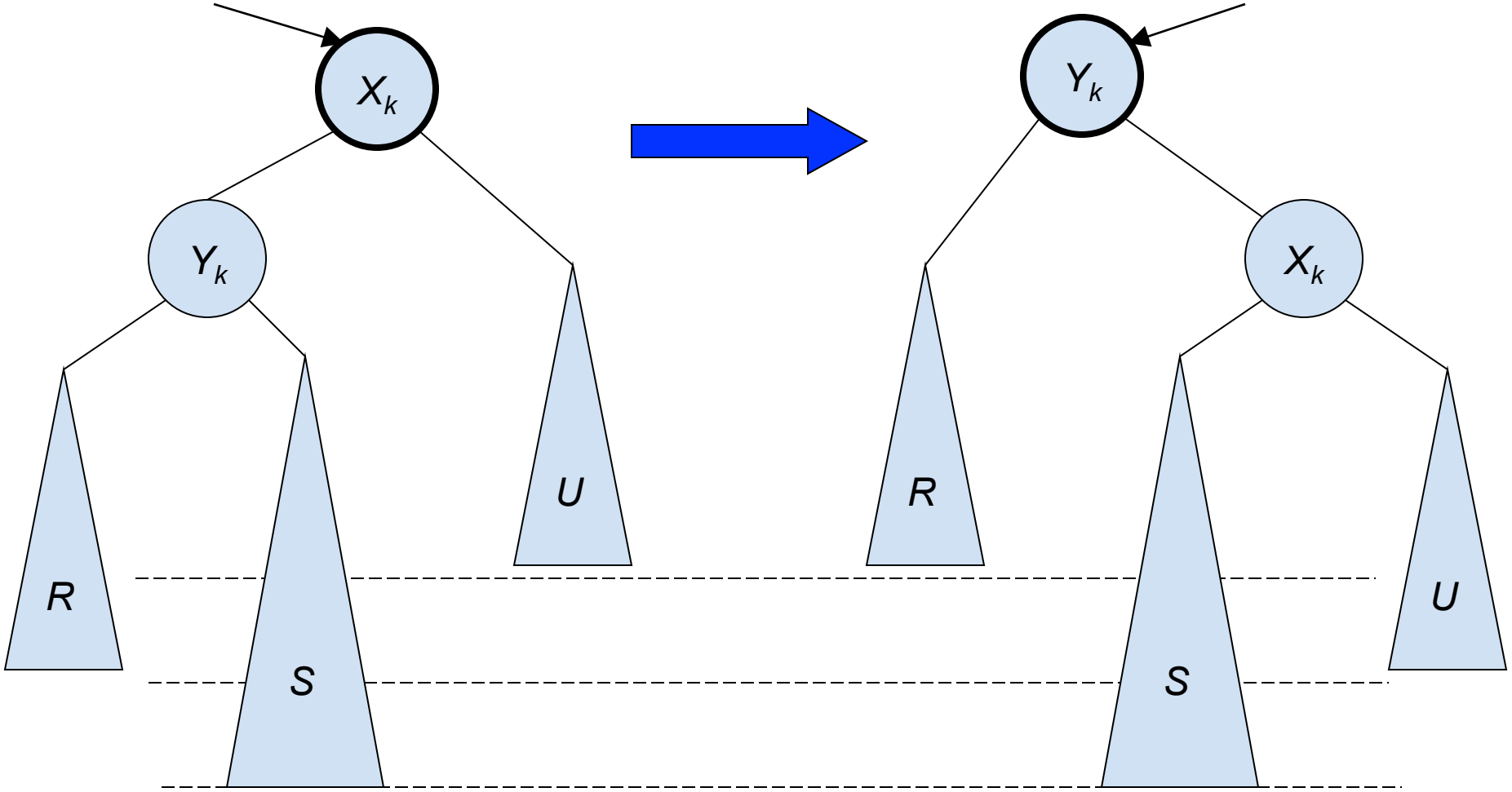
Ejemplo del caso B:



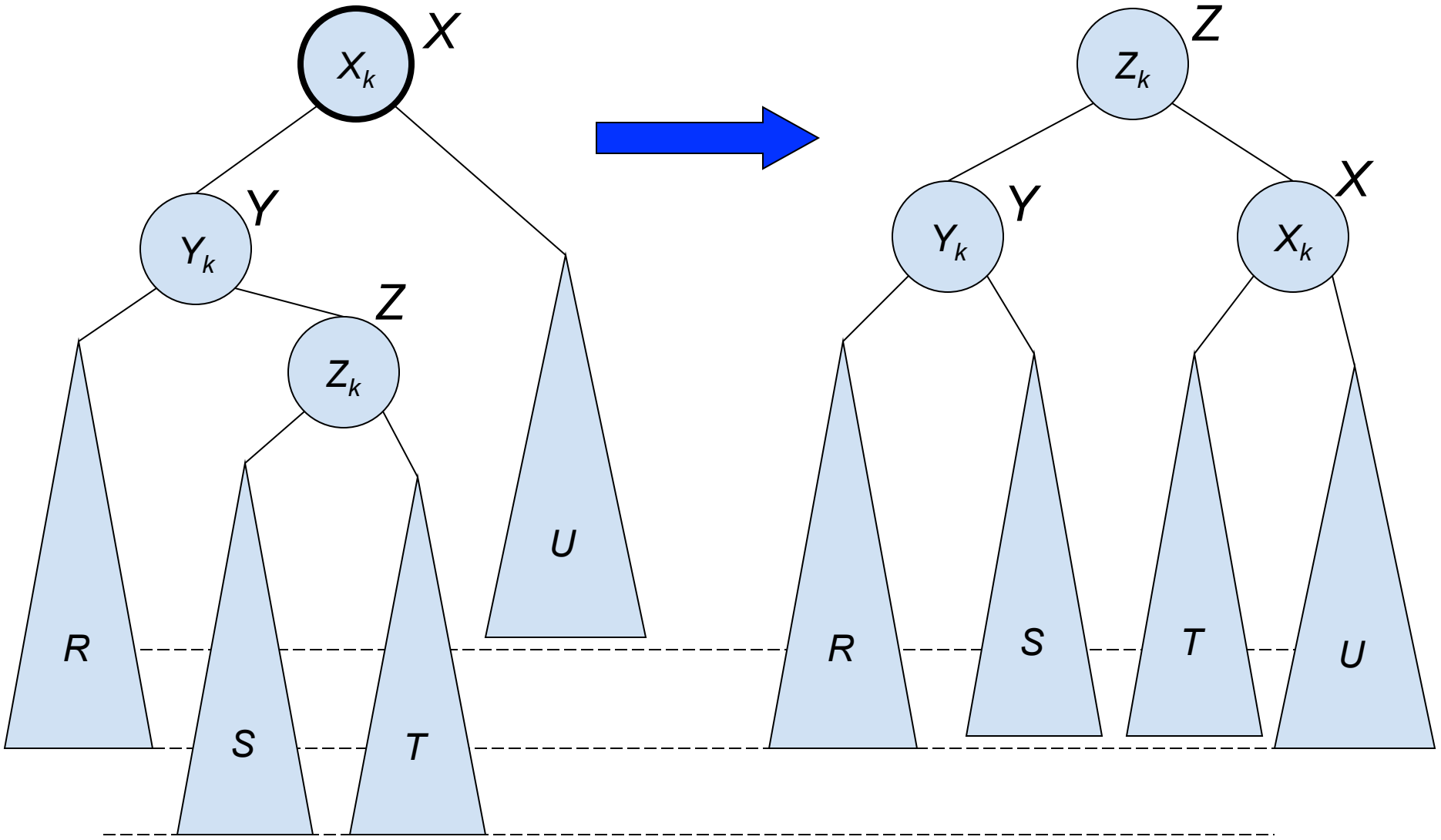
Una rotación simple *no* resuelve el caso básico B:

Nodo desbalanceado antes de la rotación

Nodo desbalanceado después de una rotación simple



Una **rotación doble** resuelve el caso básico B (el nodo insertado está en *S* o en *T*, pero no en ambos):



En el árbol original, *S* o *T* (pero no ambos), tiene una profundidad igual a la de *U* más dos

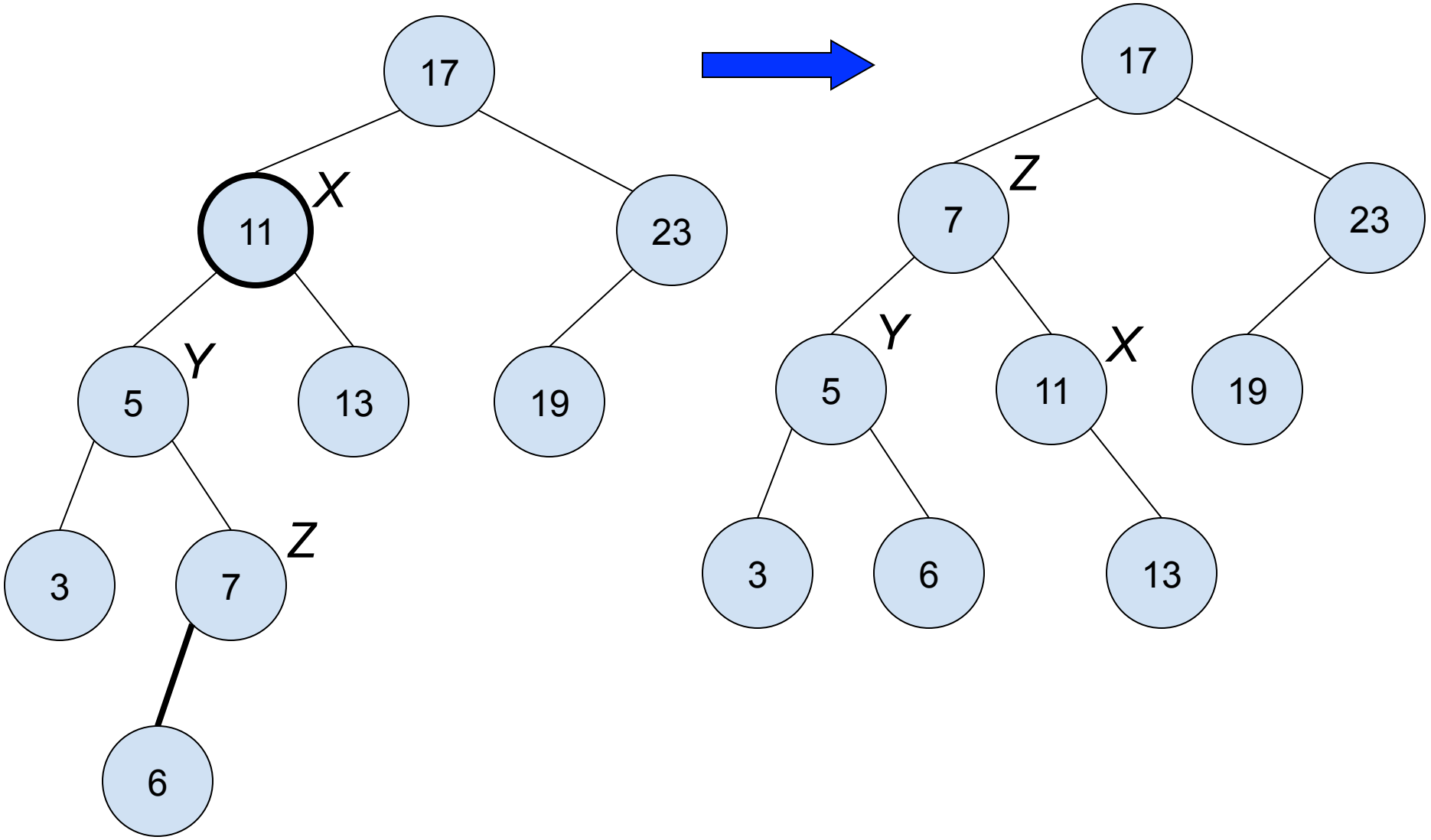
Primero, se hace una rotación simple a la izquierda en torno a *Y*–*Z*, sin involucrar a *X* (no se muestra):

- *S* se convierte en el subárbol derecho de *Y*
- *Y* se convierte en el hijo izquierdo de *Z*
- *Z* reemplaza a *Y* como hijo izquierdo de *X*

Luego, se hace una rotación simple a la derecha en torno a *X*–*Z*:

- *X* se convierte en el hijo derecho de *Z*
- *T* se convierte en el subárbol izquierdo de *X*
- *Z* ocupa el lugar que tenía *X* en el árbol original

Solución del ejemplo del caso B:



La altura de un árbol AVL es Θ(log*n*), como se demuestra a continuación

Un árbol AVL de altura h tiene al menos $F_{h+3} - 1$ nodos:

- F_i es el i-ésimo número de Fibonacci

Sea S_h el número de nodos del árbol AVL más pequeño de altura h:

$$S_0 = 1 \qquad S_1 = 2$$

Este árbol debe tener subárboles de alturas $h-1$ y $h-2$:

- al menos un subárbol tiene altura $h-1$
- la propiedad de balance implica que las alturas de los subárboles pueden diferir a lo más en 1

Estos subárboles deben tener el menor número de nodos para sus alturas:

$$S_h = S_{h-1} + S_{h-2} + 1$$

Por inducción, podemos probar la afirmación inicial

Sabemos que $F_k = \phi^k/\sqrt{5}$, en que $\phi = (1+\sqrt{5})/2 = 1.618\ldots$ (se puede demostrar por inducción)

Luego, un árbol AVL de altura h tiene al menos $\phi^{h+3}/\sqrt{5}$ nodos

Por lo tanto, la altura de un árbol AVL de n nodos es logarítmica con respecto a n