

Fundamentos de análisis y diseño de algoritmos

Árboles de búsqueda binaria

Abril de 2018

Propiedad de un árbol de búsqueda binaria

Árboles y recorrido *inorden*

Operaciones mínimo, máximo, sucesor y predecesor

Inserción y eliminación

Árboles

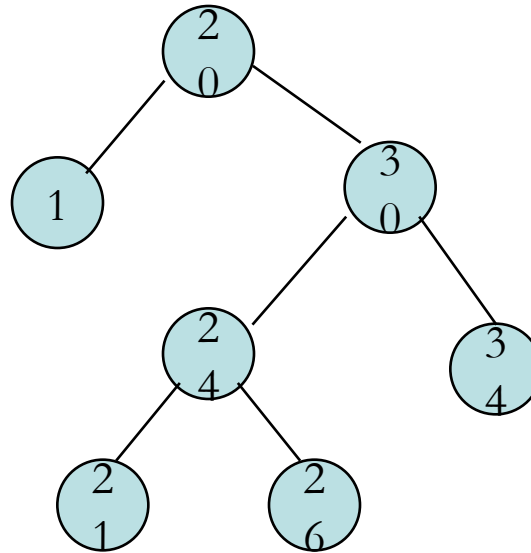
Por qué son importantes los árboles

- Operaciones básicas como insertar, borrar y buscar, toman un tiempo proporcional a la altura del árbol
- Para un árbol binario completo con n nodos, las operaciones básicas toman $\Theta(\lg n)$
- Si el árbol se construye como una cadena lineal de n nodos, tomarían $\Theta(n)$

Árboles

Árbol de búsqueda binaria

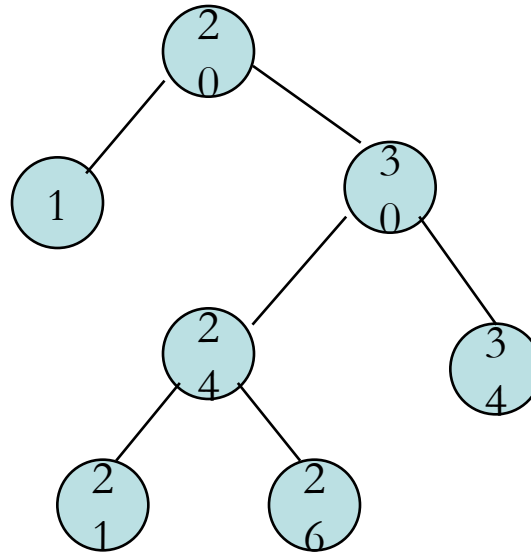
Es un árbol binario en el cual se cumple que, para cada nodo x , los nodos del subarbol izquierdo son menores o iguales a x y que, los nodos del subarbol derecho son mayores o iguales a x



Árboles

Propiedad del árbol de búsqueda binaria

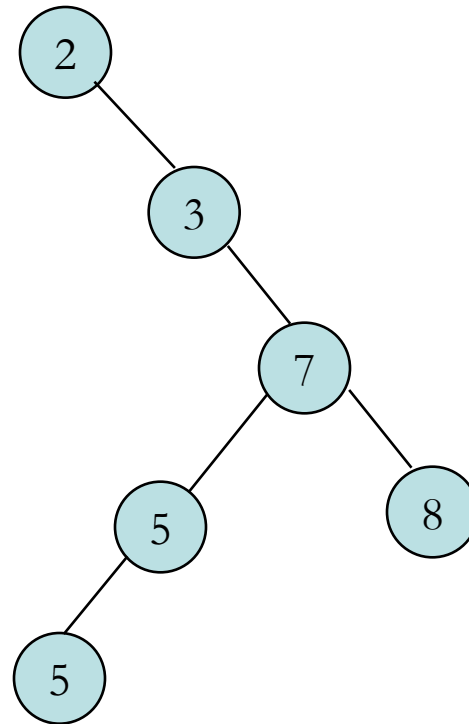
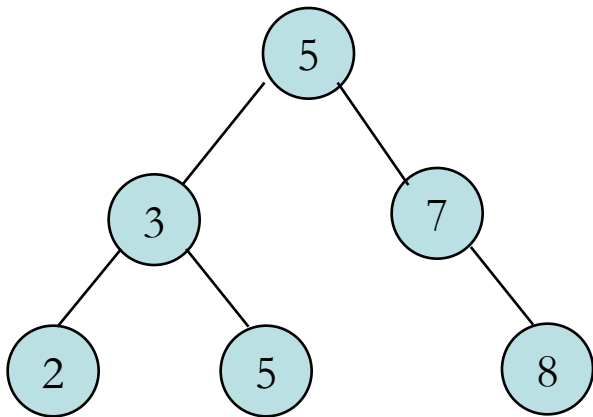
Sea x un nodo del árbol. Si y es un nodo en el subarbol izquierdo de x , entonces $\text{key}[y] \leq \text{key}[x]$. Si y es un nodo en el subarbol derecho de x , entonces $\text{key}[y] \geq \text{key}[x]$



Árboles

Propiedad del árbol de búsqueda binaria

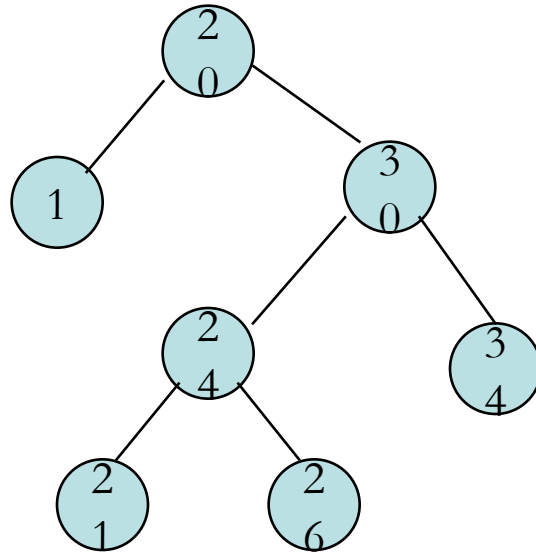
Sea x un nodo del árbol. Si y es un nodo en el subarbol izquierdo de x , entonces $\text{key}[y] \leq \text{key}[x]$. Si y es un nodo en el subarbol derecho de x , entonces $\text{key}[y] \geq \text{key}[x]$



Árboles

Árbol de búsqueda binaria

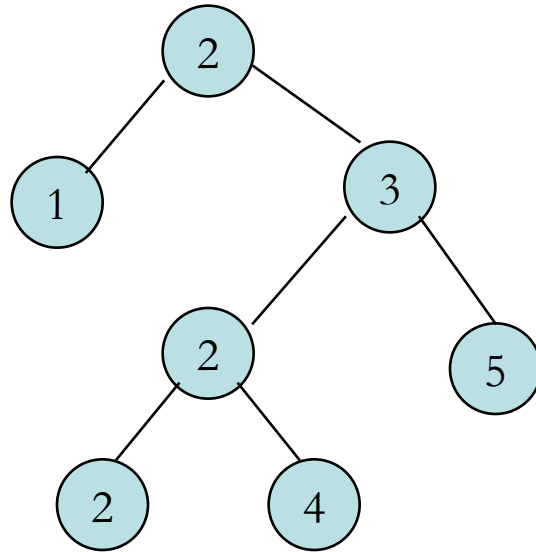
Indique si el siguiente árbol es de búsqueda binaria



Árboles

Árbol de búsqueda binaria

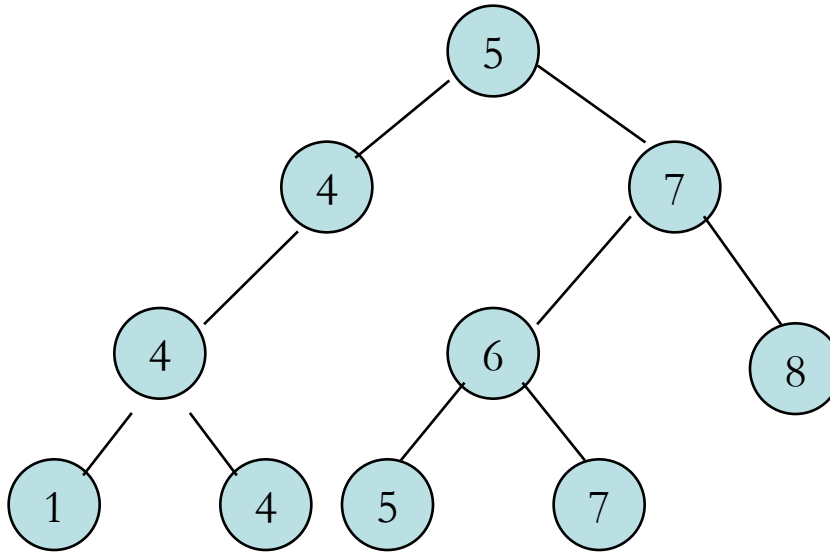
Indique si el siguiente árbol es de búsqueda binaria



Árboles

Árbol de búsqueda binaria

Indique si el siguiente árbol es de búsqueda binaria



Árboles

Los árboles de búsqueda binaria tienen otra característica, si son recorridos en *inorden*, producen una lista de las llaves ordenada ascendentemente

INORDER-TREE-WALK(x)

if $x \neq \text{nil}$

then INORDER-TREE-WALK(left[x])

print key[x]

INORDER-TREE-WALK(right[x])

Árboles

- Recorra los árboles de búsqueda binaria previos, en inorden
- Demuestre que la complejidad del algoritmo **INORDER-TREE-WALK**(x) es $\Theta(n)$

Árboles

Consulta de un árbol de búsqueda binaria

- Búsqueda de una llave
- Mínimo
- Máximo
- Sucesor de un nodo
- Predecesor de un nodo

Cada una de estas operaciones se puede hacer en $O(h)$ donde h es la altura del árbol

Árboles

Buscar un nodo con llave k dado un árbol con apuntador a la raíz x

TREE-SEARCH(x, k)

if $x = \text{nil}$ or $k = \text{key}[x]$

then return x

if $k < \text{key}[x]$

then return TREE-SEARCH(left[x], k)

else return TREE-SEARCH(right[x], k)

Árboles

Búsqueda iterativa

ITERATIVE-TREE-SEARCH(x, k)

while $x \neq \text{nil}$ and $k \neq \text{key}[x]$

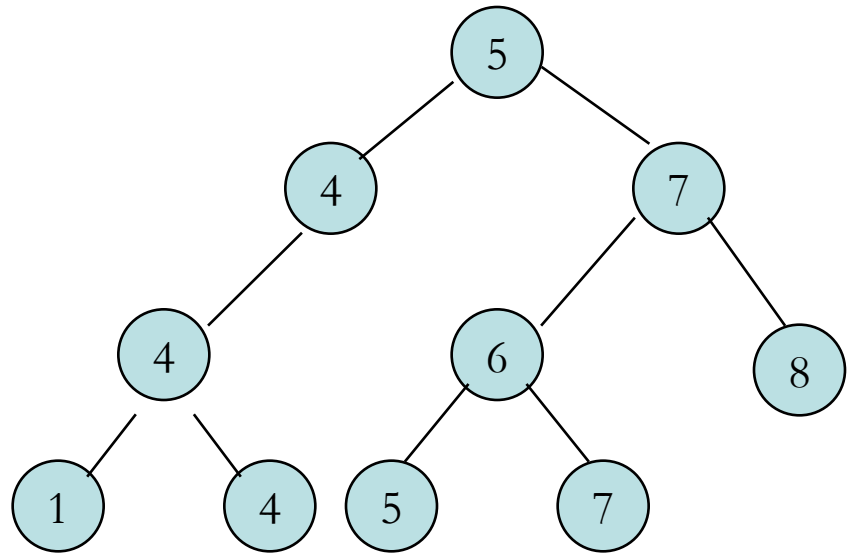
do if $k < \text{key}[x]$

then $x \leftarrow \text{left}[x]$

else $x \leftarrow \text{right}[x]$

Árboles

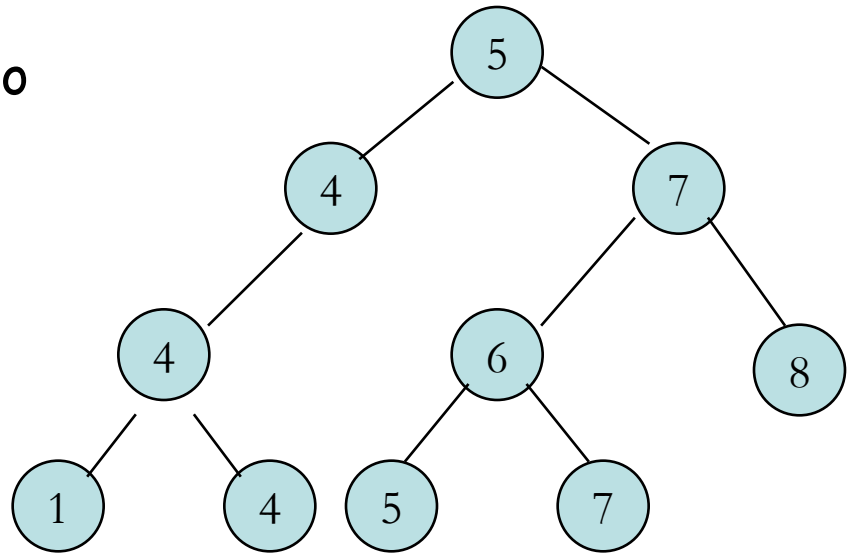
En un árbol de búsqueda binaria dónde se ubica el elemento mínimo?



Árboles

En un árbol de búsqueda binaria dónde se ubica el elemento mínimo?

Idea: seguir los apuntadores al hijo izquierdo desde la raíz hasta que se encuentre nil



Árboles

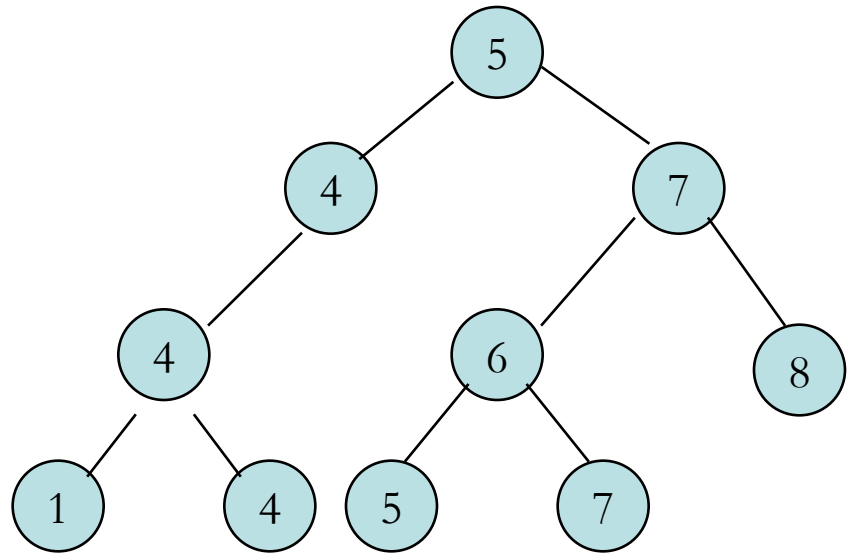
En un árbol de búsqueda binaria dónde se ubica el elemento mínimo?

TREE-MINIMUN(x)

while left[x] ≠ nil

do $x \leftarrow \text{left}[x]$

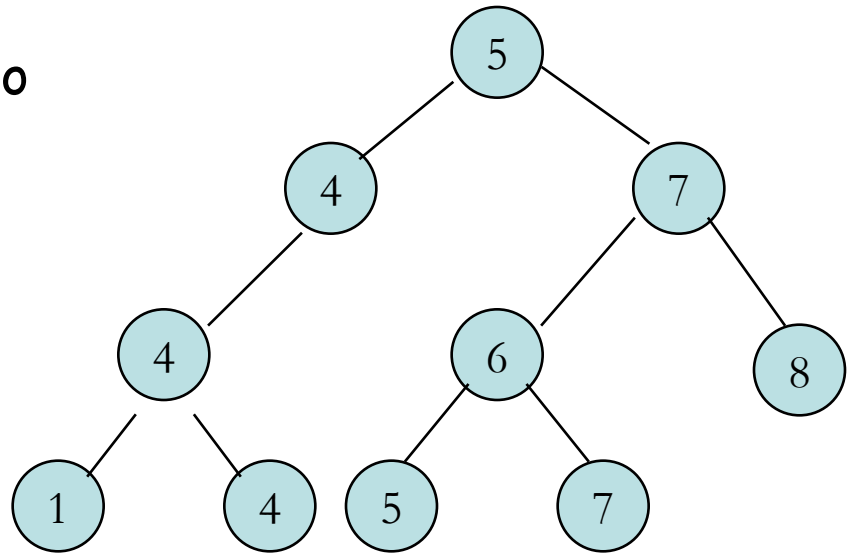
return x



Árboles

En un árbol de búsqueda binaria dónde se ubica el elemento máximo?

Idea: seguir los apuntadores al hijo derecho desde la raíz hasta que se encuentre nil



Árboles

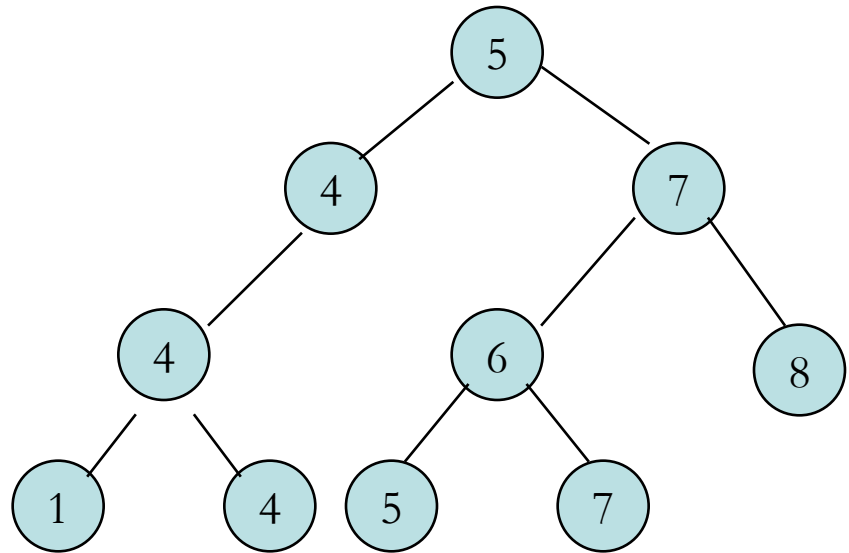
En un árbol de búsqueda binaria dónde se ubica el elemento máximo?

TREE-MAXIMUM(x)

while right[x] \neq nil

do $x \leftarrow \text{right}[x]$

return x

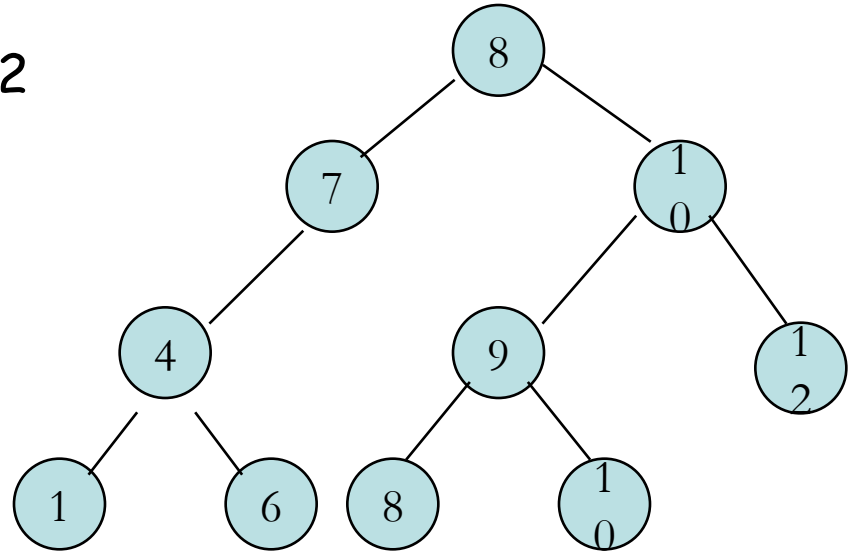


Árboles

Sucesor

Dado un nodo x donde $\text{key}[x]=k$, el sucesor de x es el nodo y tal que $\text{key}[y]$ es la llave más pequeña, mayor que $\text{key}[x]$

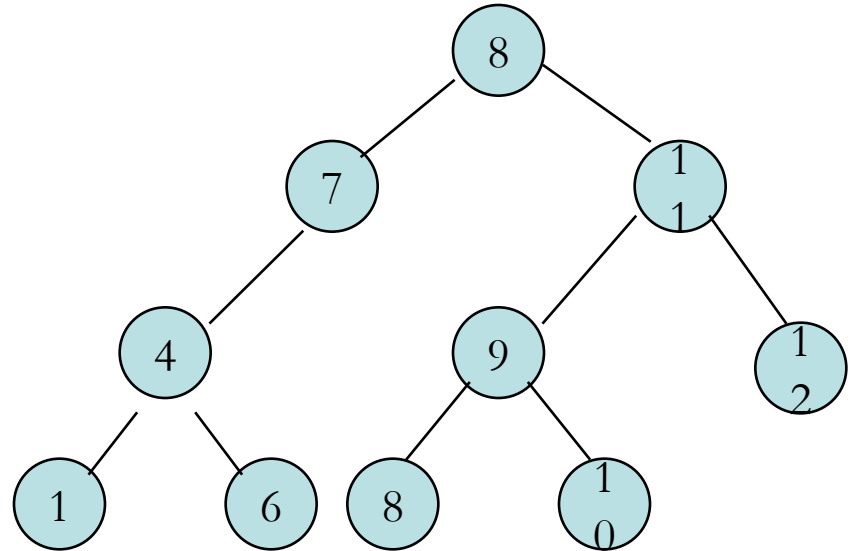
Cuál es el sucesor de 7, 9, 10 y 12



Árboles

TREE-SUCCESSOR(x)

```
if right[x] ≠ nil
  then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ nil and x = right[y]
  do x ← y
  y ← p[y]
return y
```



Explique el código anterior

para el caso de TREE-SUCCESSOR(4)

Árboles

TREE-SUCCESSOR(x)

if right[x] ≠ nil

then return TREE-MINIMUM(right[x])

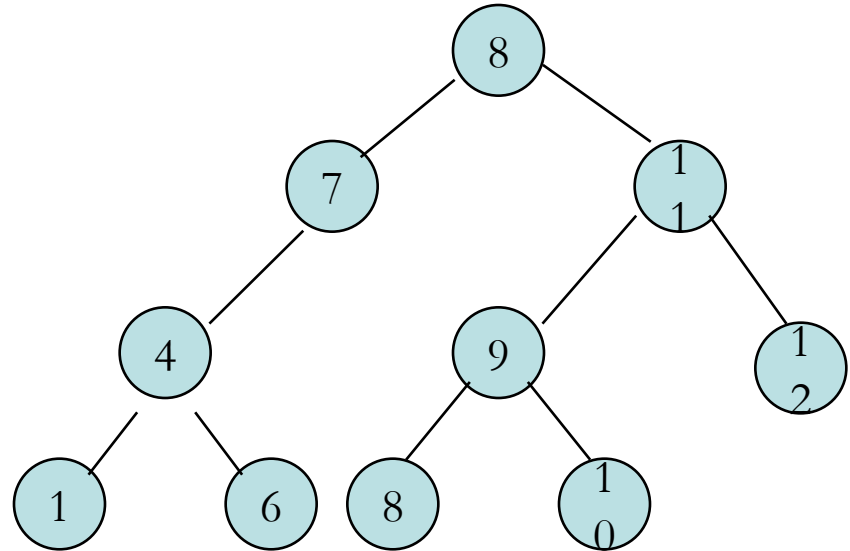
y ← p[x]

while y ≠ nil and x = right[y]

do x ← y

y ← p[y]

return y



Explique el código anterior

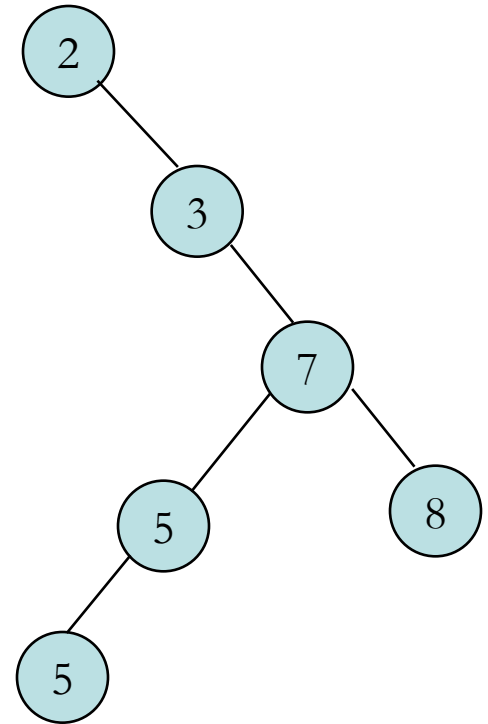
para el caso de **TREE-SUCCESSOR**(7)

Árboles

TREE-SUCCESSOR(x)

```
if right[x] ≠ nil
  then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ nil and x = right[y]
  do x ← y
  y ← p[y]
return y
```

Explique el código anterior
para el caso de **TREE-SUCCESSOR**(8)



TREE-INSERT(x)

$y \leftarrow \text{nil}$

$x \leftarrow \text{root}[T]$

while $x \neq \text{nil}$

do $y \leftarrow x$

if $\text{key}[z] < \text{key}[x]$

then $x \leftarrow \text{left}[x]$

else $x \leftarrow \text{right}[x]$

$p[z] \leftarrow y$

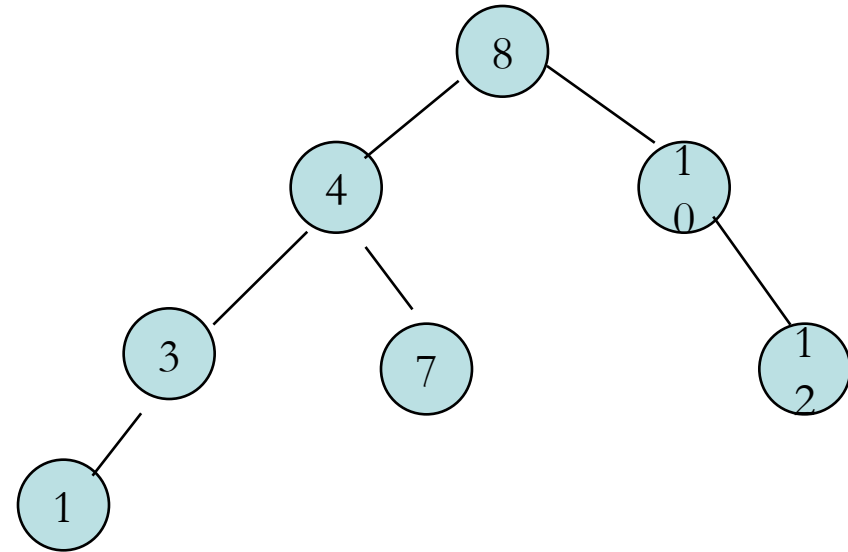
if $y = \text{nil}$

then $\text{root}[T] \leftarrow z$

else if $\text{key}[z] < \text{key}[y]$

then $\text{left}[y] \leftarrow z$

else $\text{right}[y] \leftarrow z$



Explique el código para el caso de **TREE-INSERT**(z), donde $\text{key}[z]=5$

TREE-INSERT(x)

$y \leftarrow \text{nil}$

$x \leftarrow \text{root}[T]$

while $x \neq \text{nil}$

do $y \leftarrow x$

if $\text{key}[z] < \text{key}[x]$

then $x \leftarrow \text{left}[x]$

else $x \leftarrow \text{right}[x]$

$p[z] \leftarrow y$

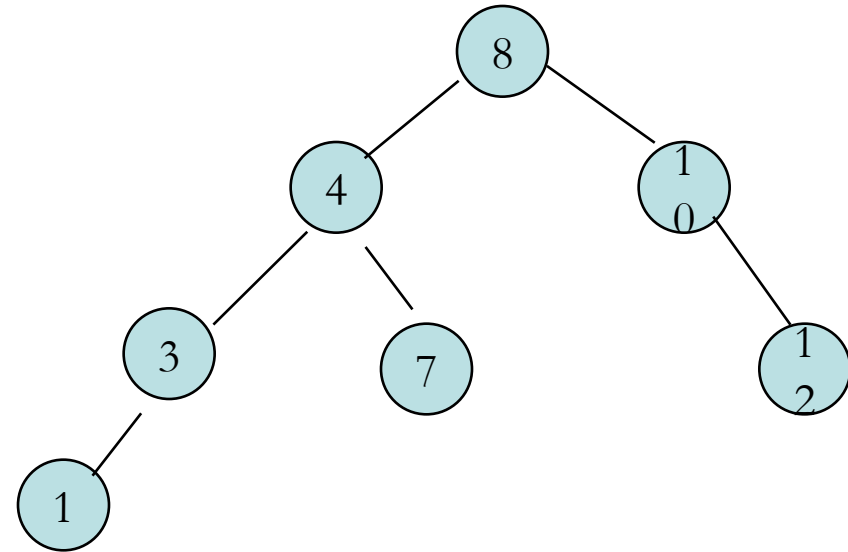
if $y = \text{nil}$

then $\text{root}[T] \leftarrow z$

else if $\text{key}[z] < \text{key}[y]$

then $\text{left}[y] \leftarrow z$

else $\text{right}[y] \leftarrow z$



Explique el código para el caso de **TREE-INSERT**(z), donde $\text{key}[z]=9$

TREE-INSERT(x)

$y \leftarrow \text{nil}$

$x \leftarrow \text{root}[T]$

while $x \neq \text{nil}$

do $y \leftarrow x$

if $\text{key}[z] < \text{key}[x]$

then $x \leftarrow \text{left}[x]$

else $x \leftarrow \text{right}[x]$

$p[z] \leftarrow y$

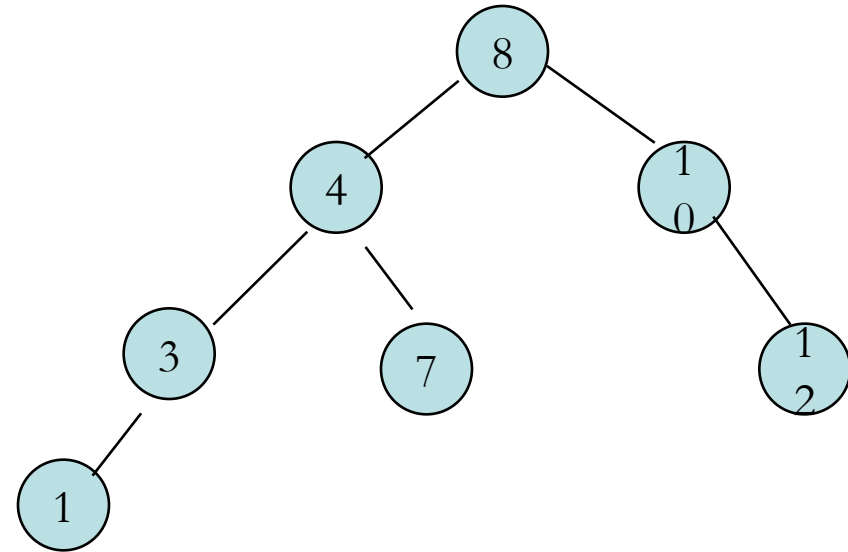
if $y = \text{nil}$

then $\text{root}[T] \leftarrow z$

else if $\text{key}[z] < \text{key}[y]$

then $\text{left}[y] \leftarrow z$

else $\text{right}[y] \leftarrow z$



Explique el código para el caso de **TREE-INSERT**(z), donde $\text{key}[z]=11$

TREE-INSERT(x)

$y \leftarrow \text{nil}$

$x \leftarrow \text{root}[T]$

while $x \neq \text{nil}$

do $y \leftarrow x$

if $\text{key}[z] < \text{key}[x]$

then $x \leftarrow \text{left}[x]$

else $x \leftarrow \text{right}[x]$

$p[z] \leftarrow y$

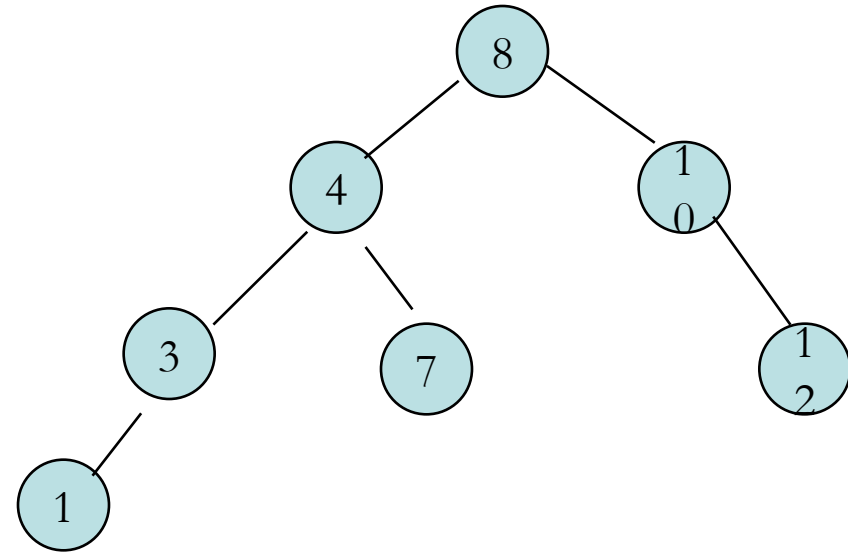
if $y = \text{nil}$

then $\text{root}[T] \leftarrow z$

else if $\text{key}[z] < \text{key}[y]$

then $\text{left}[y] \leftarrow z$

else $\text{right}[y] \leftarrow z$



La complejidad es de $O(h)$

TREE-DELETE(x)

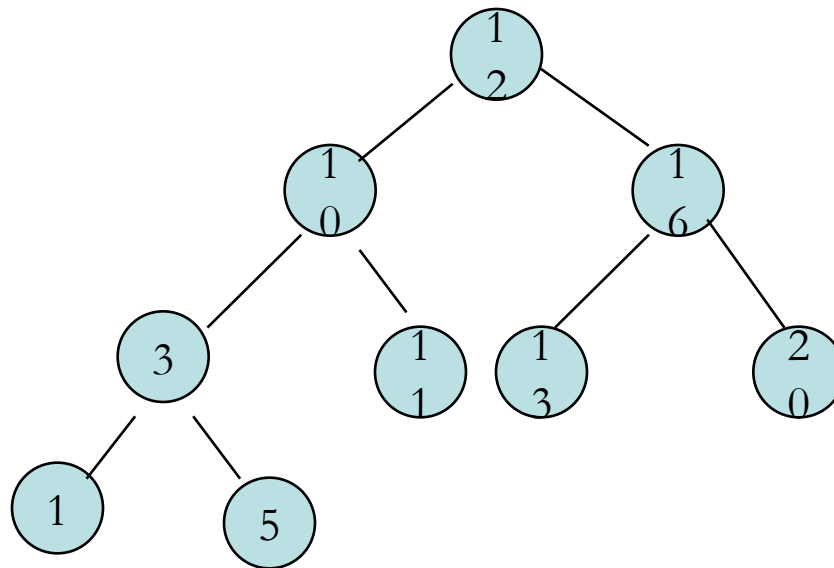
```
if left[z]=nil or right[z]=nil
    then  $y \leftarrow z$ 
    else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
if left[y]≠nil
    then  $x \leftarrow \text{left}[y]$ 
    else  $x \leftarrow \text{right}[y]$ 
if  $x \neq \text{nil}$ 
    then  $p[x] \leftarrow p[y]$ 
if  $p[y] = \text{nil}$ 
    then  $\text{root}[T] \leftarrow x$ 
    else if  $y = \text{left}[p[y]]$ 
        then  $\text{left}[p[y]] \leftarrow x$ 
        else  $\text{right}[p[y]] \leftarrow x$ 
if  $y \neq z$ 
    then  $\text{key}[z] \leftarrow \text{key}[y]$ 
return  $y$ 
```

Caso 1:

Borrar z y z no tiene hijos.

TREE-DELETE(T, z), donde $\text{key}[z]=5$

Qué se debe hacer?



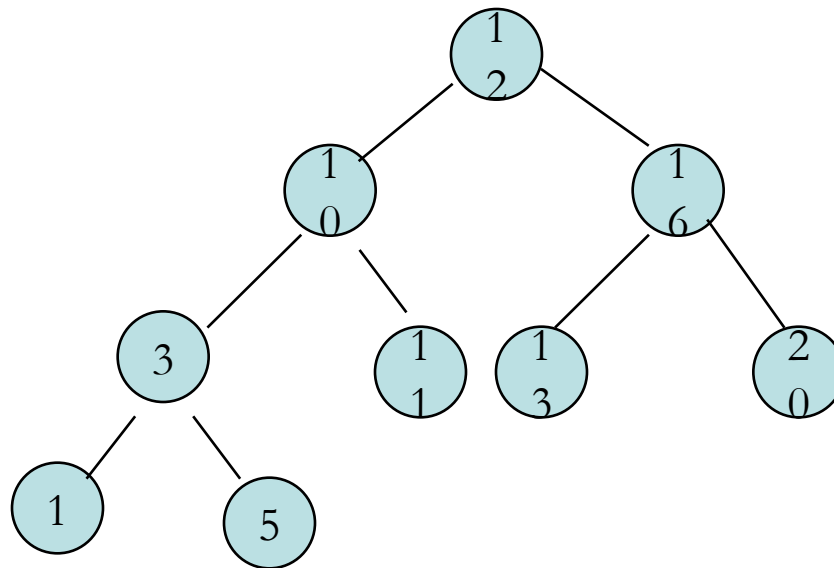
Caso 1:

Borrar z y z no tiene hijos.

TREE-DELETE(T, z), donde $\text{key}[z]=5$

El padre de z debe ahora apuntar a nil

$p[z] \leftarrow \text{nil}$

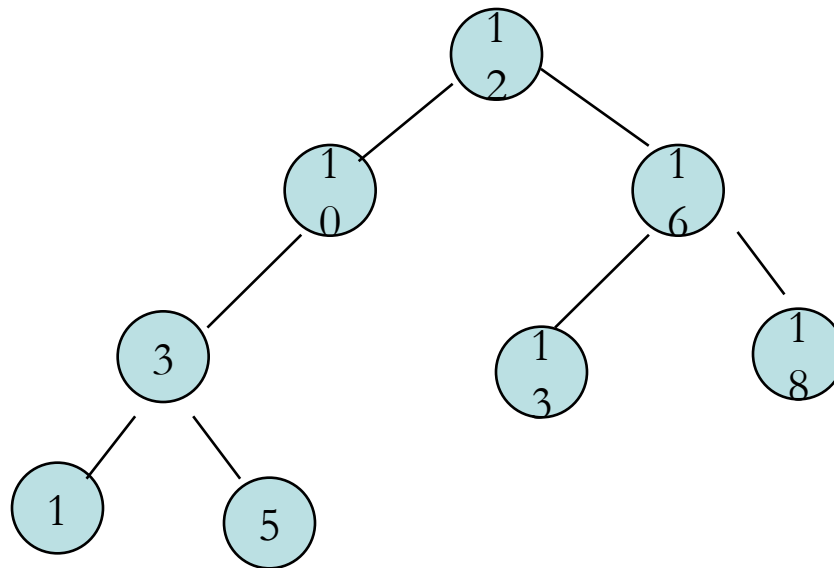


Caso 2:

Borrar z y z tiene un solo hijo

TREE-DELETE(T, z), donde $\text{key}[z]=10$

Qué se debe hacer?

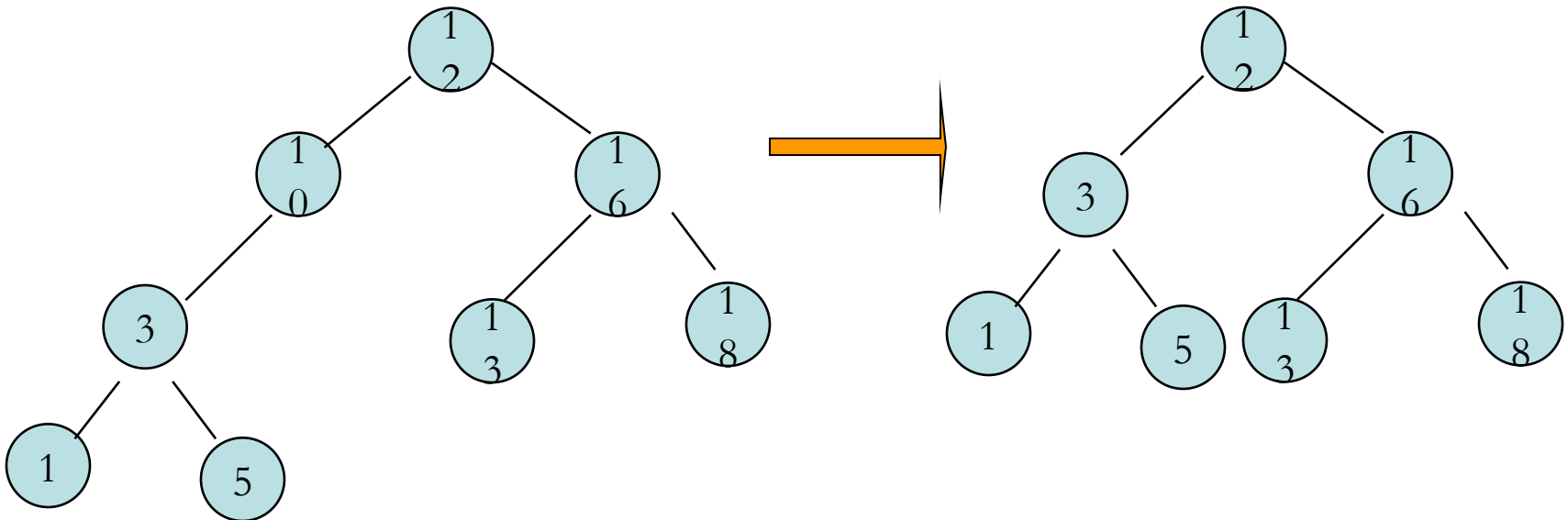


Caso 2:

Borrar z y z tiene un solo hijo

TREE-DELETE(T, z), donde $\text{key}[z]=10$

Se separa z del árbol

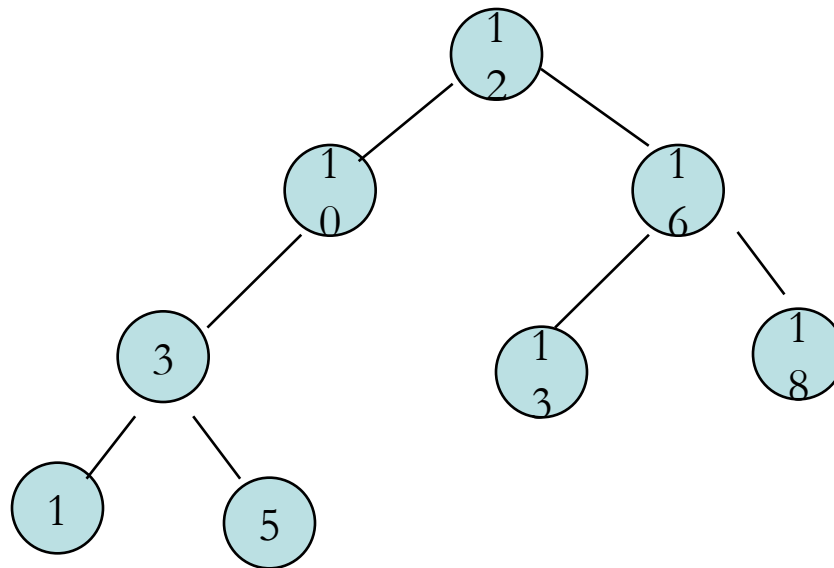


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Qué se debe hacer?

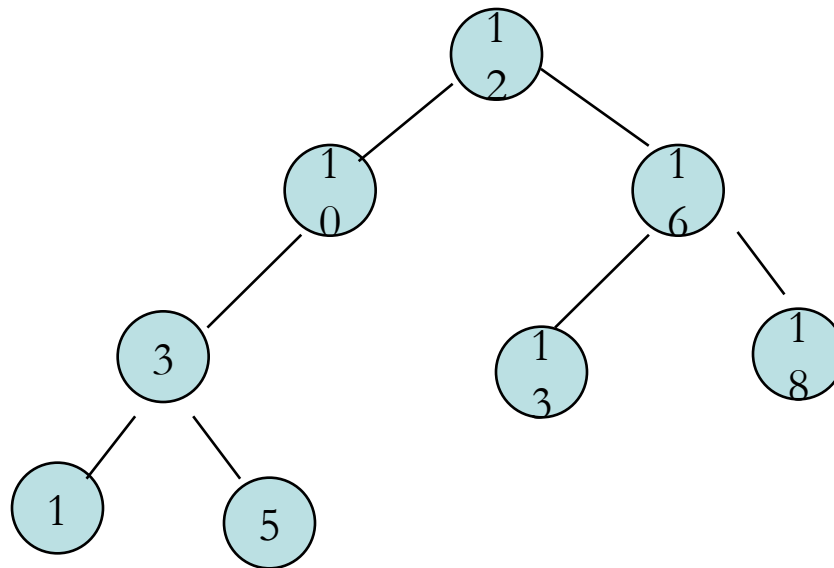


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Qué se debe hacer?



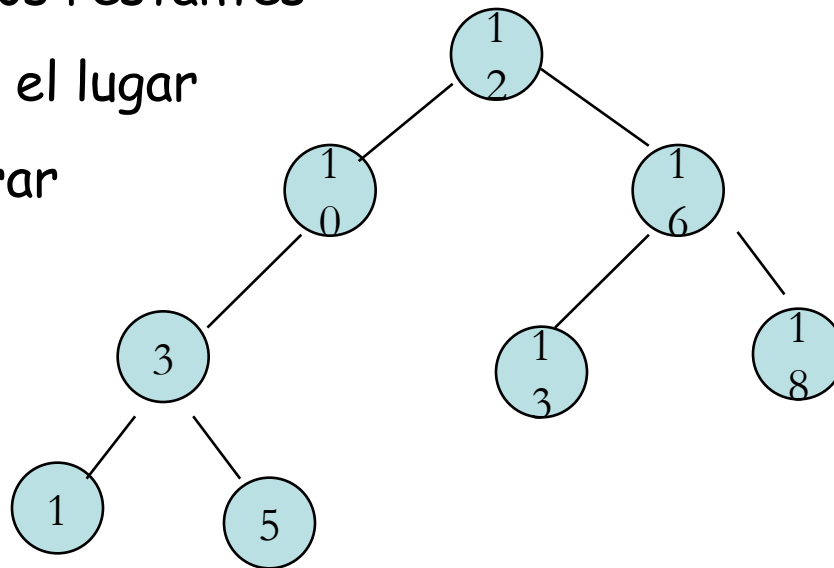
Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Qué se debe hacer?

Cuál de los nodos restantes
debería ocupar el lugar
del nodo a borrar

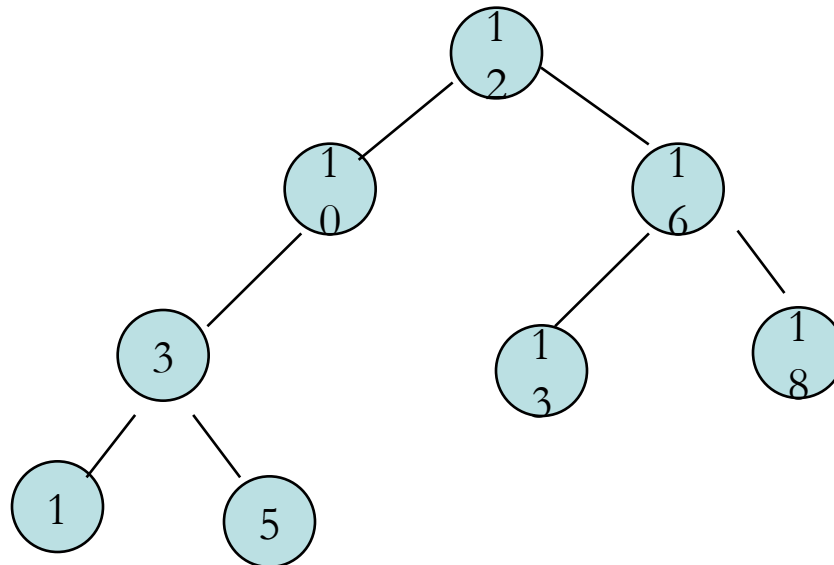


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Se separa(elimina) su sucesor y del árbol
y se reemplaza su contenido con el
de z

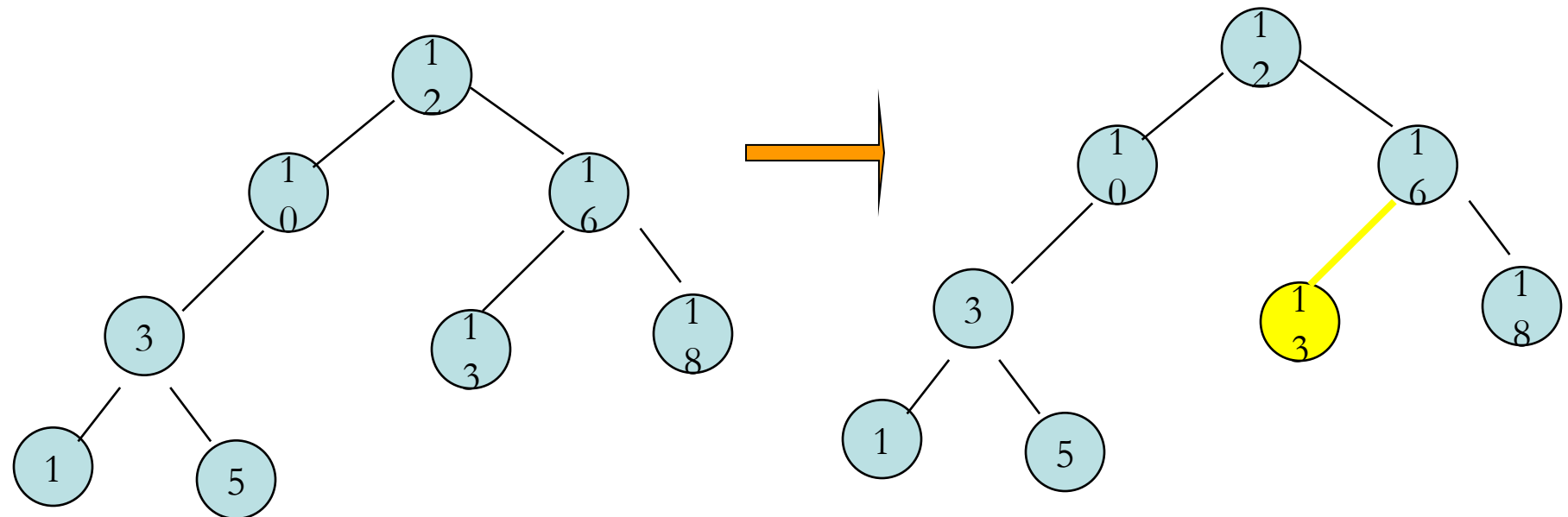


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Se separa(elimina) su sucesor y del árbol
y se reemplaza su contenido con el
de z

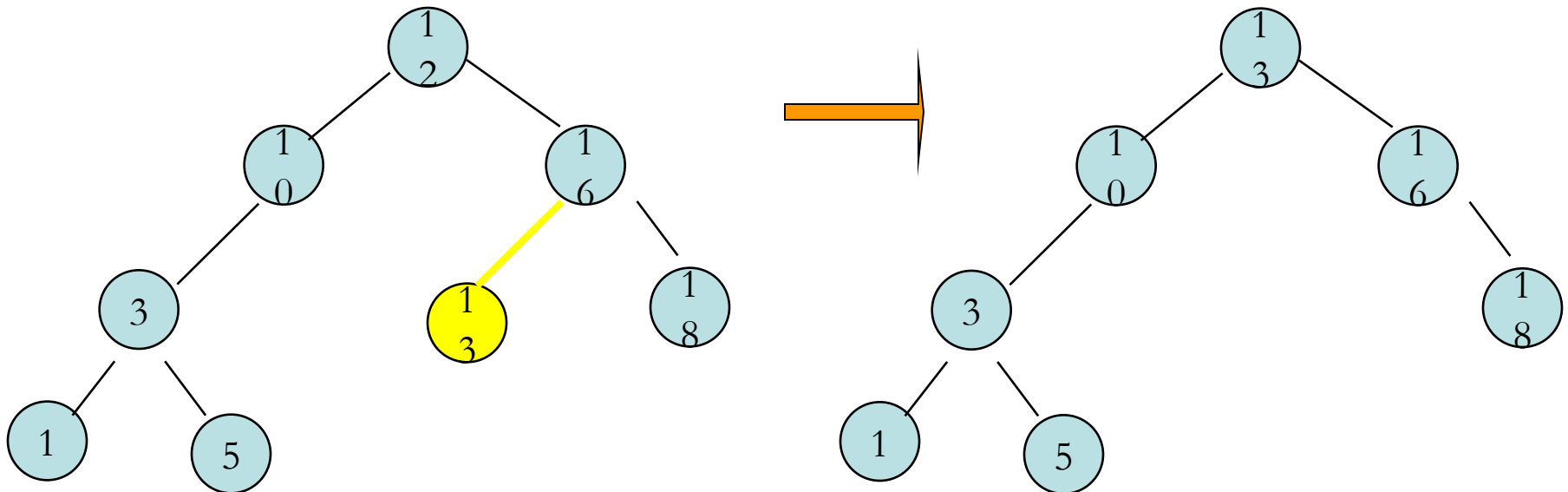


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T, z), donde $\text{key}[z]=12$

Se separa(elimina) su sucesor y del árbol
y se reemplaza su contenido con el
de z



TREE-DELETE(x)

if left[z]=nil or right[z]=nil

then $y \leftarrow z$

else $y \leftarrow \text{TREE-SUCCESSOR}(z)$

if left[y]≠nil

then $x \leftarrow \text{left}[y]$

else $x \leftarrow \text{right}[y]$

if $x \neq \text{nil}$

then $p[x] \leftarrow p[y]$

if $p[y] = \text{nil}$

then $\text{root}[T] \leftarrow x$

else if $y = \text{left}[p[y]]$

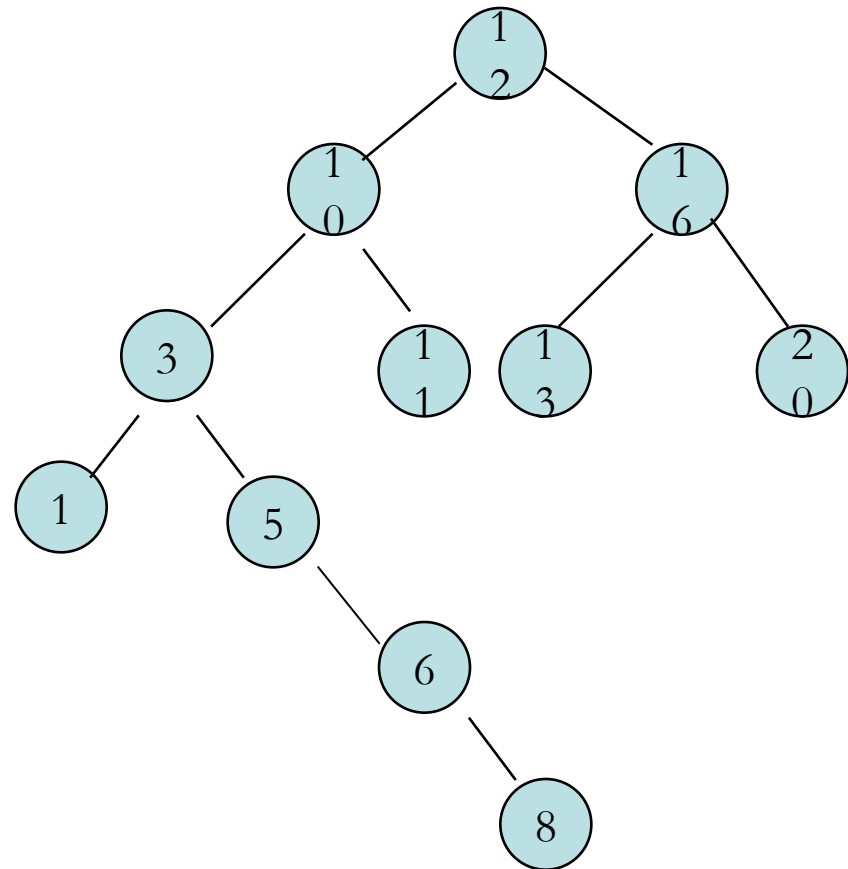
then $\text{left}[p[y]] \leftarrow x$

else $\text{right}[p[y]] \leftarrow x$

if $y \neq z$

then $\text{key}[z] \leftarrow \text{key}[y]$

return y



Siga el algoritmo TREE-DELETE(T,z)
donde z es el nodo tal que $\text{key}[z]=11$

TREE-DELETE(x)

if left[z]=nil or right[z]=nil

then $y \leftarrow z$

else $y \leftarrow \text{TREE-SUCCESSOR}(z)$

if left[y]≠nil

then $x \leftarrow \text{left}[y]$

else $x \leftarrow \text{right}[y]$

if $x \neq \text{nil}$

then $p[x] \leftarrow p[y]$

if $p[y] = \text{nil}$

then $\text{root}[T] \leftarrow x$

else if $y = \text{left}[p[y]]$

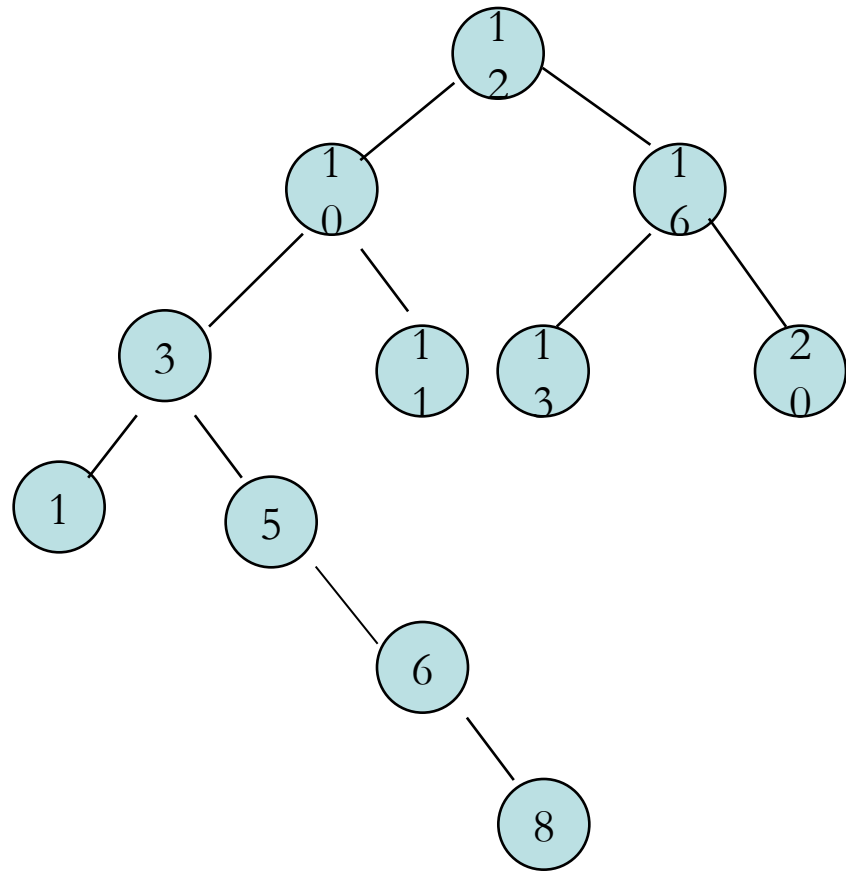
then $\text{left}[p[y]] \leftarrow x$

else $\text{right}[p[y]] \leftarrow x$

if $y \neq z$

then $\text{key}[z] \leftarrow \text{key}[y]$

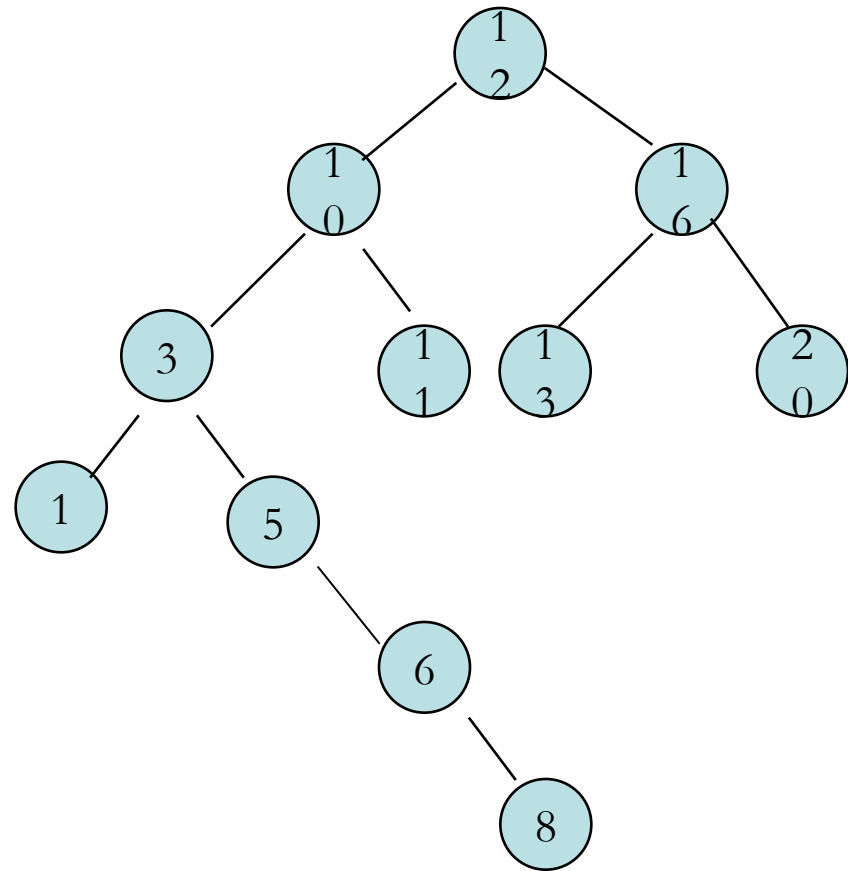
return y



Siga el algoritmo TREE-DELETE(T,z)
donde z es el nodo tal que $\text{key}[z]=6$

TREE-DELETE(x)

```
if left[z]=nil or right[z]=nil
  then y ← z
  else y ← TREE-SUCCESSOR(z)
if left[y]≠nil
  then x ← left[y]
  else x ← right[y]
if x≠nil
  then p[x] ← p[y]
if p[y]=nil
  then root[T] ← x
  else if y=left[p[y]]
    then left[p[y]] ← x
    else right[p[y]] ← x
if y≠z
  then key[z] ← key[y]
return y
```



Siga el algoritmo TREE-DELETE(T,z)
donde z es el nodo tal que **key[z]=10**

Referencias

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press. Chapter 12

Gracias

Próximo tema:

Estructuras de datos: Árboles rojinegros