Fundamentos de análisis y diseño de algoritmos

Árboles de búsqueda binaria

Propiedad de un árbol de búsqueda binaria

Árboles y recorrido *inorden*

Operaciones mínimo, máximo, sucesor y predecesor

Inserción y eliminación

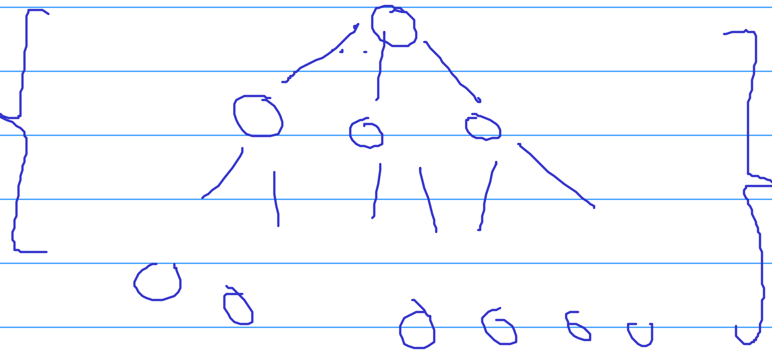
Árboles

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 $\Theta(n)$

Arbol completo

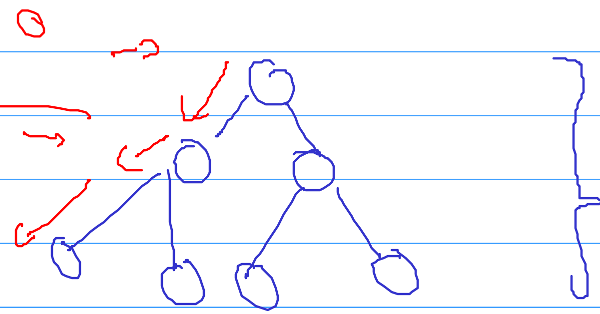
$h-1$



h

→ Mejor Caso

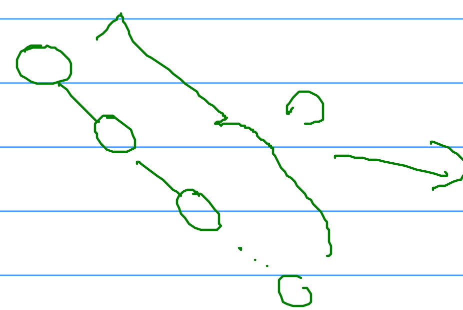
$\Theta(\log(n))$



n nodos

$h = \log(n)$

Arbol cadena lineal → Peor Caso

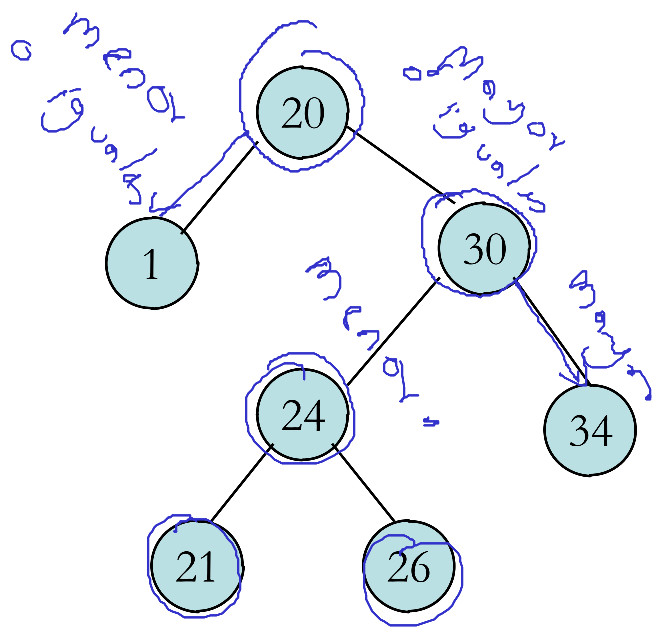


$\rightarrow \Theta(n)$

Árboles

Árbol de búsqueda binaria

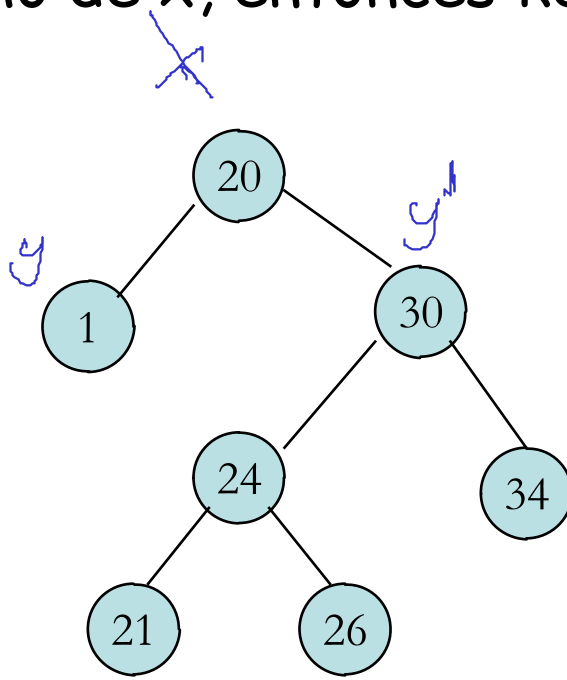
Es un árbol binario en el cual se cumple que, para cada nodo x, los nodos del subarbol izquierdo son menores o iguales a x y que, los nodos del subarbol derecho son mayores o iguales a x



Árboles

Propiedad del árbol de búsqueda binaria

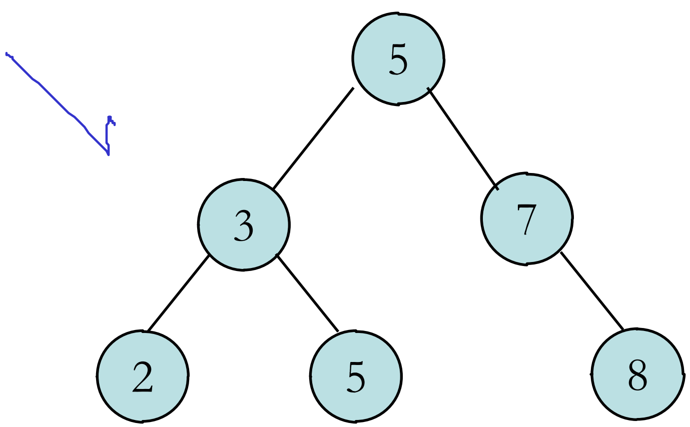
Sea x un nodo del árbol. Si y es un nodo en el subarbol izquierdo de x, entonces key[y]≤key[x]. Si y es un nodo en el subarbol derecho de x, entonces key[y]≥key[x]

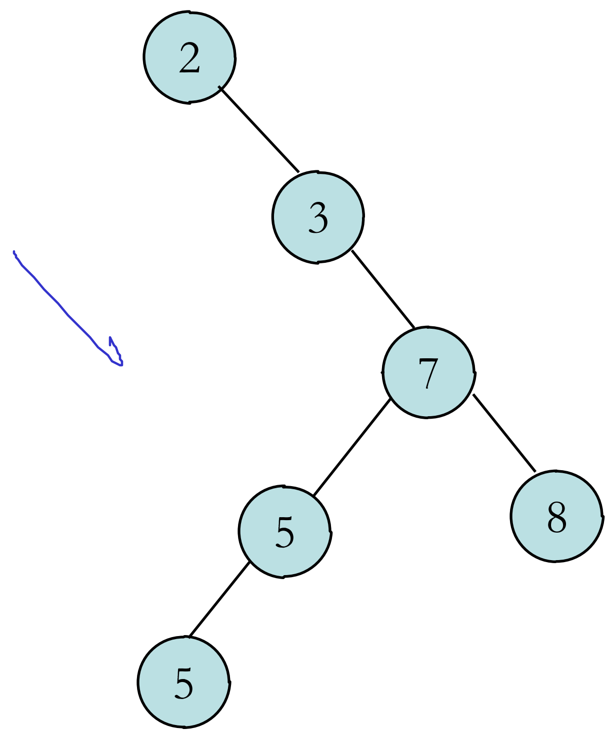


Árboles

Propiedad del árbol de búsqueda binaria

Sea x un nodo del árbol. Si y es un nodo en el subarbol izquierdo de x, entonces key[y]≤key[x]. Si y es un nodo en el subarbol derecho de x, entonces key[y]≥key[x]

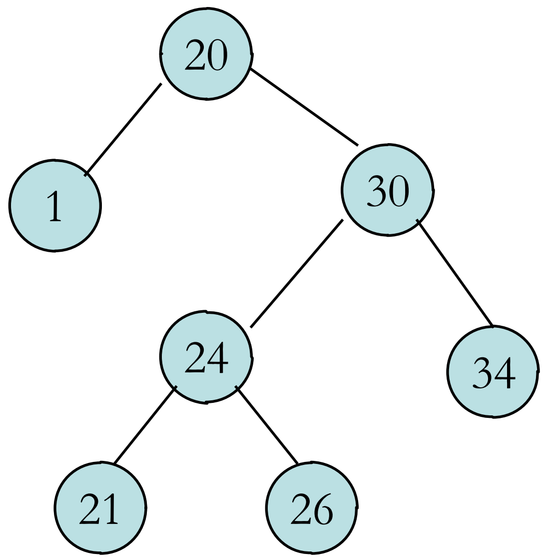




Árboles

Árbol de búsqueda binaria

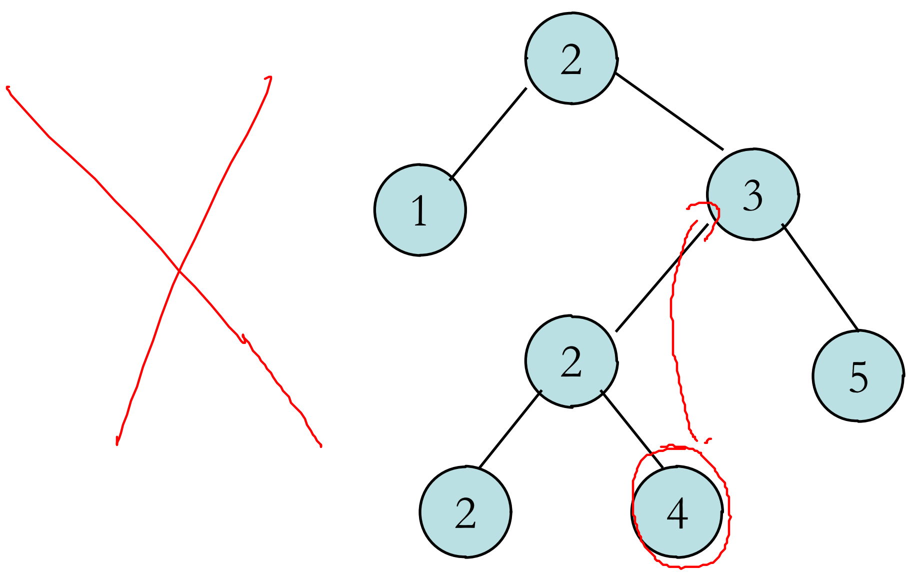
Indique si el siguiente árbol es de búsqueda binaria



Árboles

Árbol de búsqueda binaria

Indique si el siguiente árbol es de búsqueda binaria

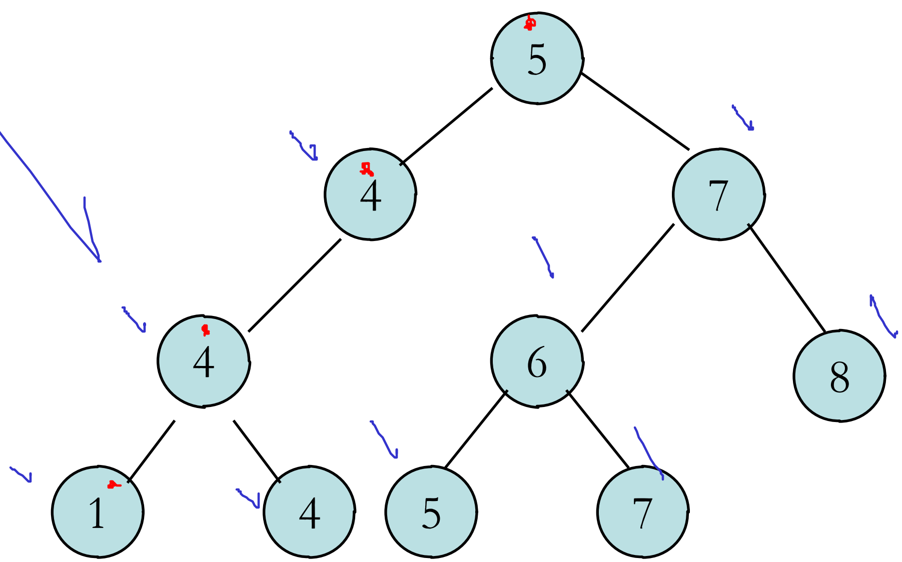


4 > 3

Árboles

Árbol de búsqueda binaria

Indique si el siguiente árbol es de búsqueda binaria



Inorden

1) Izq

2) Nodo

3) Der

{1, 4, 4, 4, 5, 5, 6, 7, 7, 8}

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 ≠ 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 $\Theta(n)$

n = # nodo

R/ Tiene que recorrerlos todos

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 raiz x

TREE-SEARCH(x,k) árbol

Buscamos

if x=nil or k=key[x]

then return x

if k<key[x]

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

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

Árboles

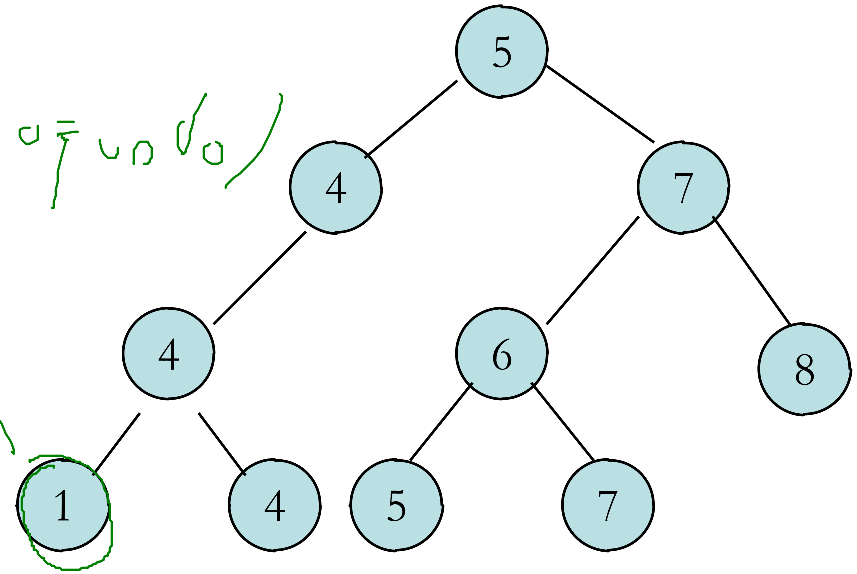
Búsqueda iterativa

```
ITERATIVE-TREE-SEARCH(x,k)
  while x≠nil and k≠key[x]
    do if k<key[x]
           then x←left[x]
           else  x←right[x]
```

Árboles

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

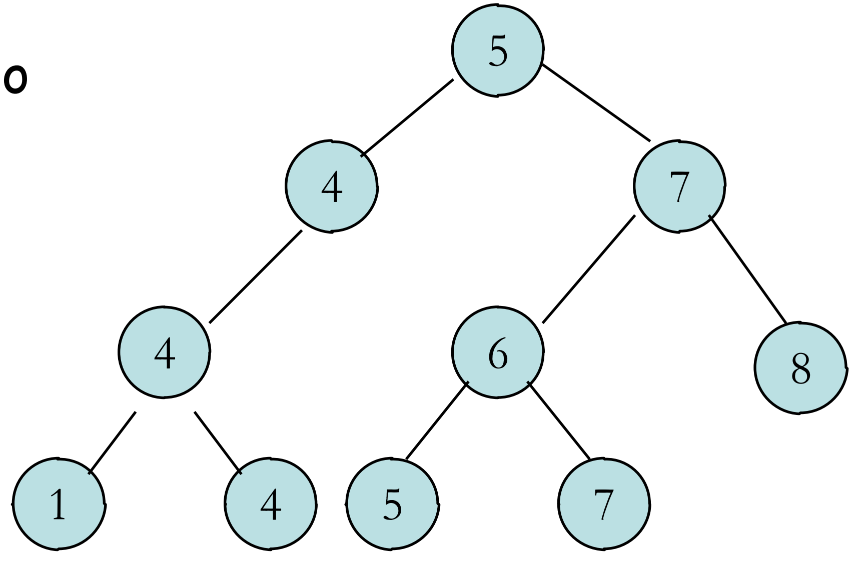
El nodo (profundo) más a la izquierda



Árboles

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

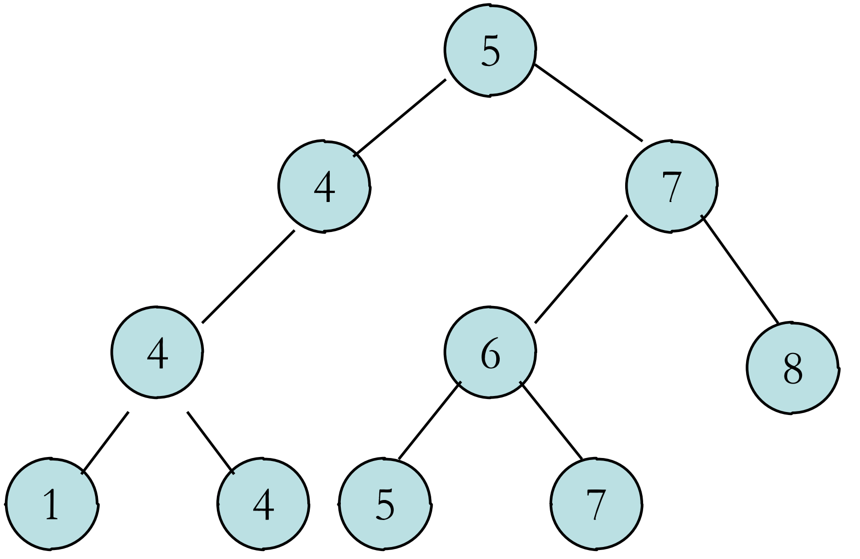
Idea: seguir los apuntadores al hijo izquierdo desde la raiz hasta que se encuentre nil



Árboles

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

```
TREE-MINIMUN(x)
  while left[x] ≠nil
    do x←left[x]
  return x
```

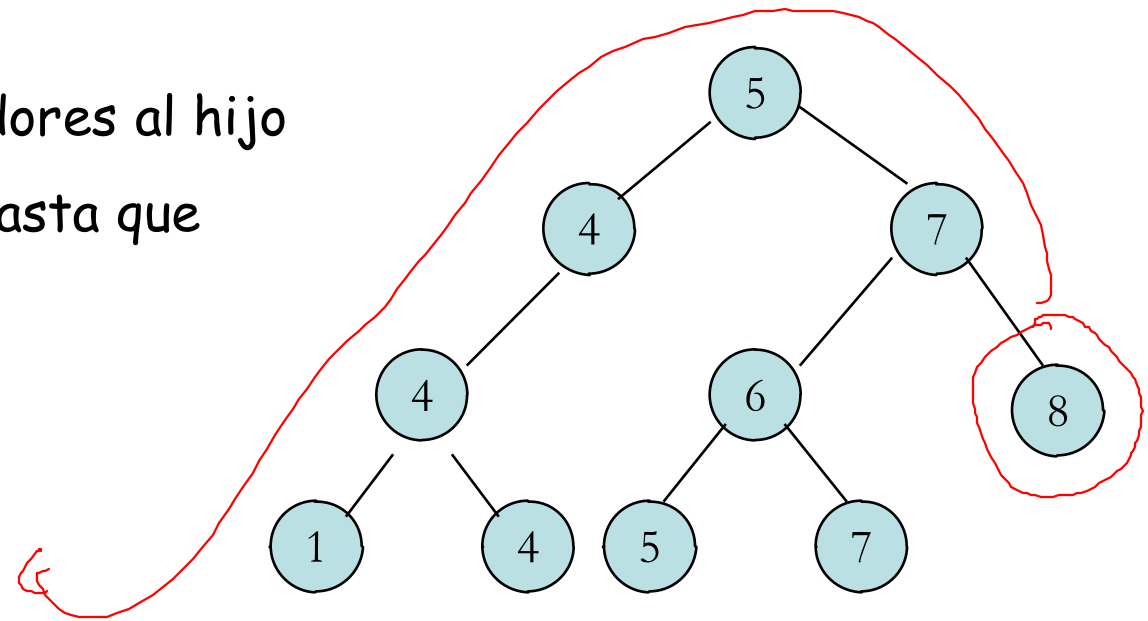


Árboles

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

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

el elmento más a la derecha



Árboles

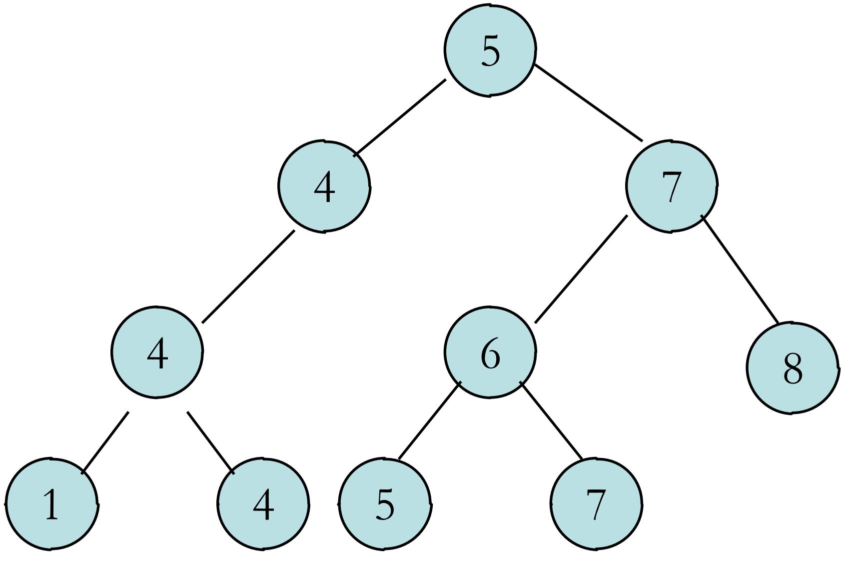
En un árbol de búsqueda binaria dónde se ubica el elemento máximo?

TREE-MAXIMUM(x)

while right[x] $\neq$nil

do x←right[x]

return x

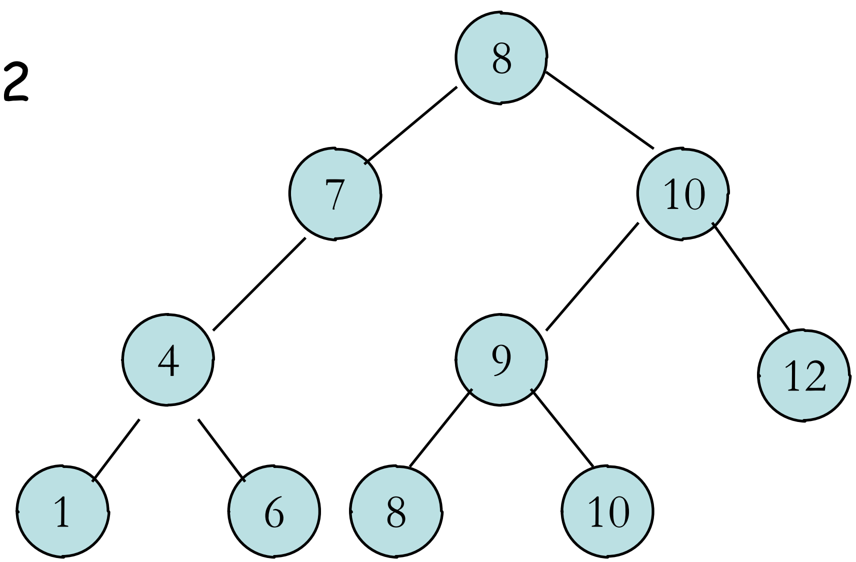


Árboles

Sucesor

Dado un nodo x donde key[x]=k, el sucesor de x es el nodo y tal que key[y] es la llave más pequeña, mayor que 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])

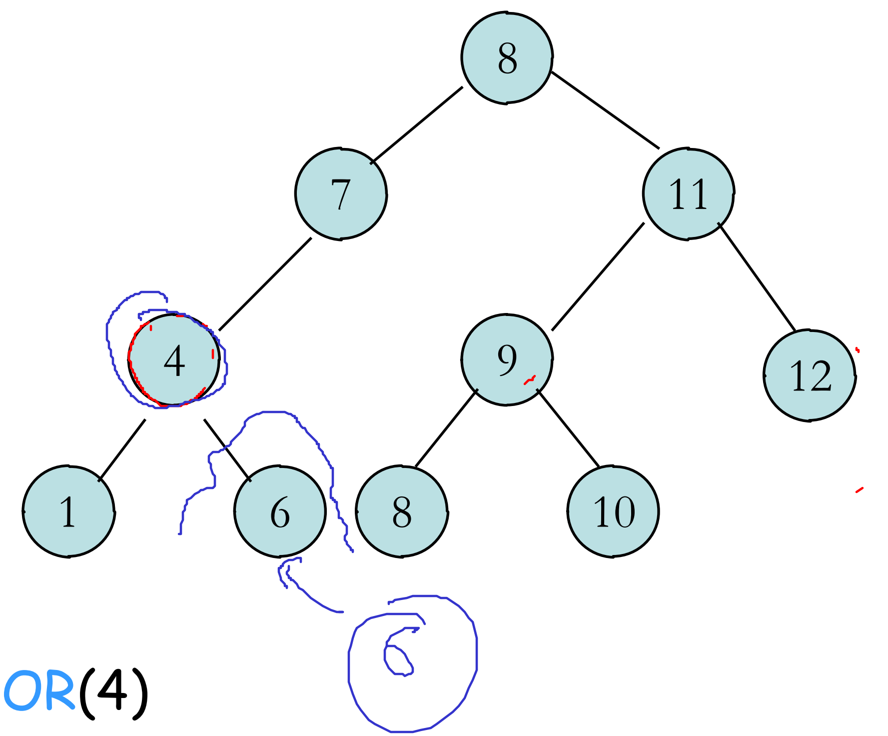
y←p[x]

while y≠nil and x=right[y]

do x←y

y←p[y]

return y

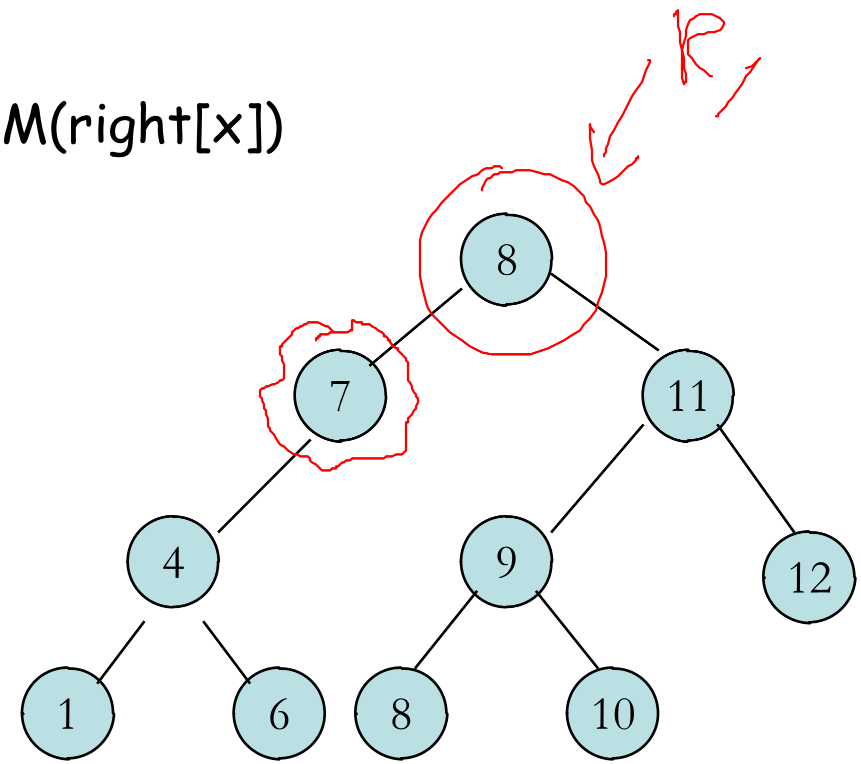


Explique el código anterior

para el caso de TREE-SUCCESSOR(4)

Árboles

```
TREE-SUCCESSOR(x)
  if right[x]≠nil
     then return TREE-MINIMUM(right[x])
  y←p[x]                      y = 8
  while y≠nil and x=right[y]
     do x←y                 7 = 11
        y←p[y]
  return y
```



Explique el código anterior

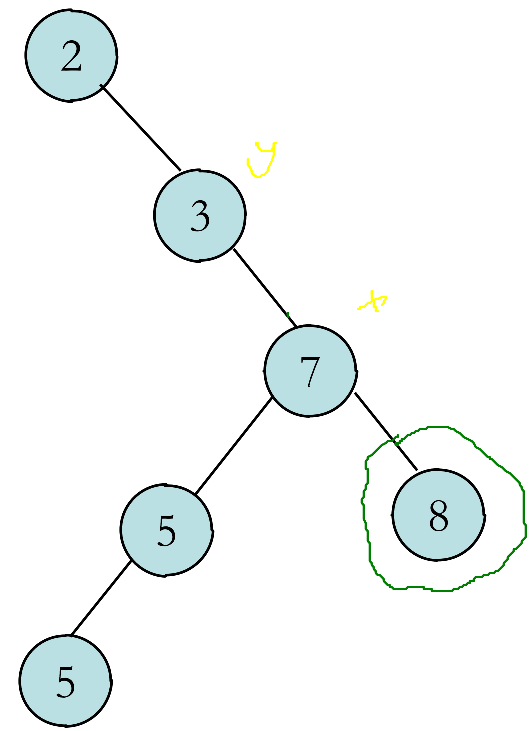
para el caso de TREE-SUCCESSOR(7)

Árboles

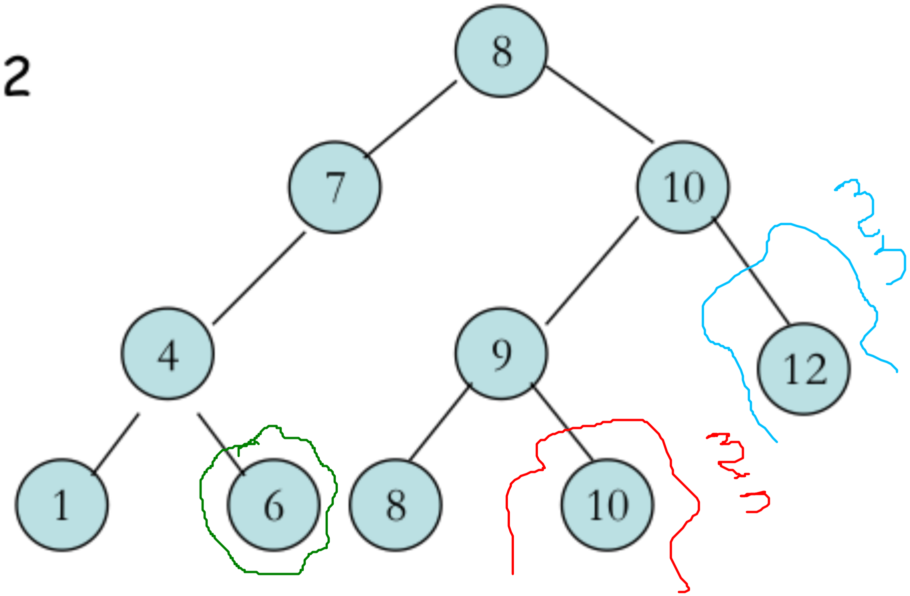
```
TREE-SUCCESSOR(x)                          x= 8
   if right[x]≠nil
      then return TREE-MINIMUM(right[x])     x         right(y)
   y←p[x]                     y= 7          8  =  8
   while y≠nil and x=right[y]
      do x←y                  x= 7   3   2
         y←p[y]               y= 3   2   nil
   return y                       nil
```

Explique el código anterior

para el caso de TREE-SUCCESSOR(8)



0 y 12



Succ(9)

R/ 10

Succ 6

X = 6

y = 4

X = 4

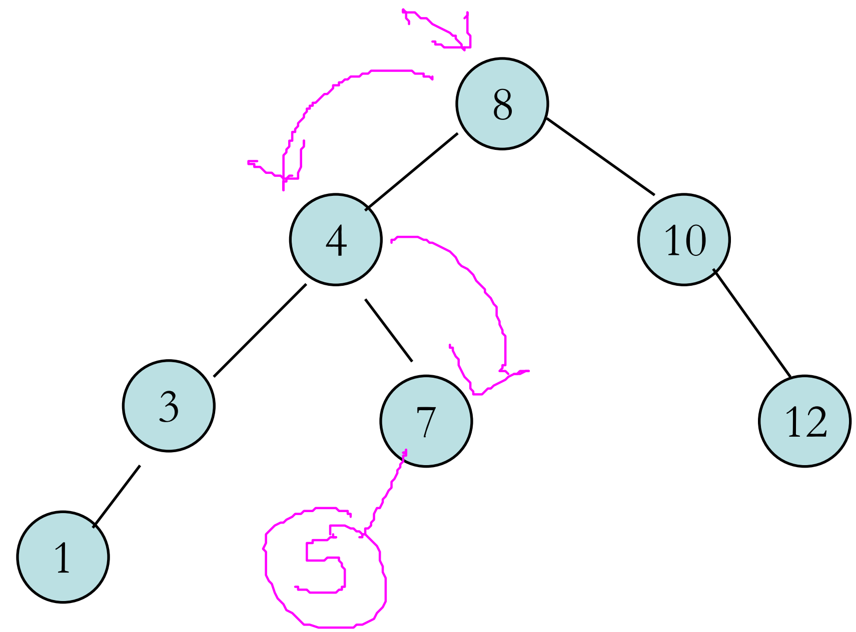
y = 7

$\boxed{R/\ 7}$

Succ(10)

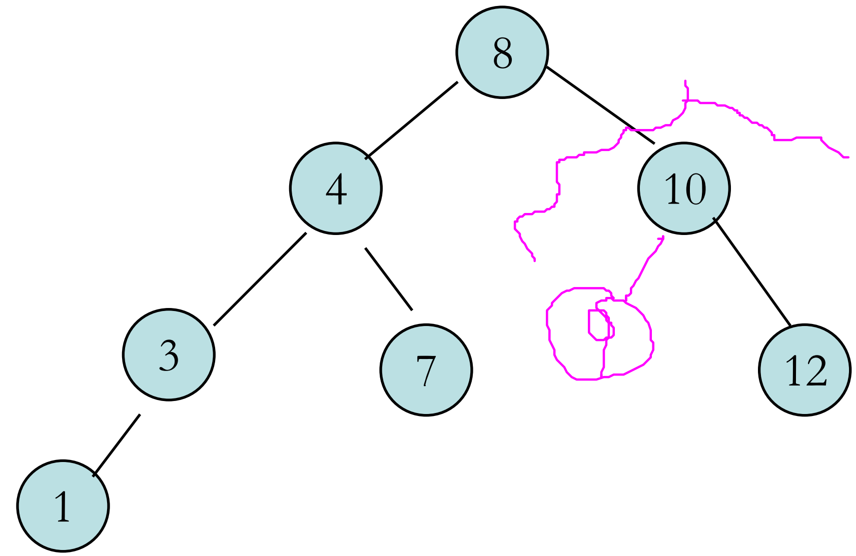
R/ 10

```
TREE-INSERT(x)
 y←nil
 x←root[T]
 while x≠nil
    do y←x
       if key[z]<key[x]
          then x←left[x]
          else x←right[x]
 p[z]←y
 if y=nil
    then root[T]←z
    else if key[z]<key[y]
            then left[y]←z
            else right[y]←z
```



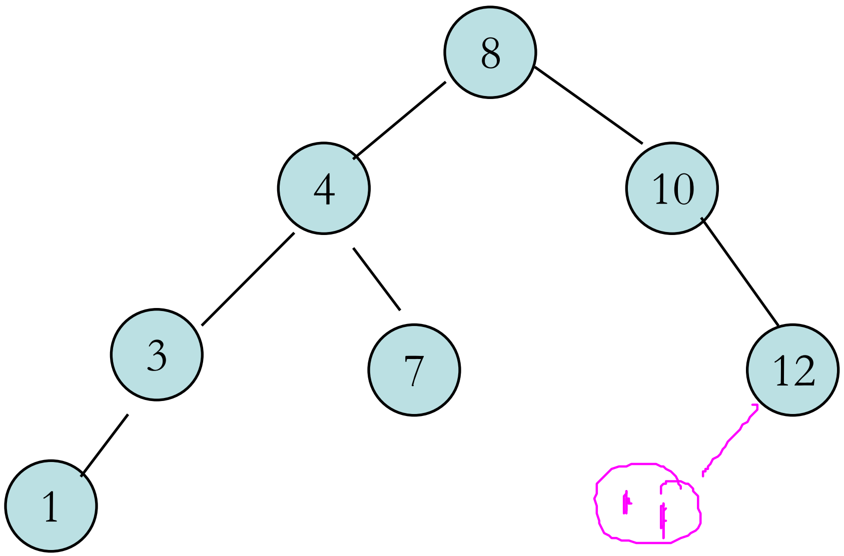
Explique el código para el caso de TREE-INSERT(z), donde key[z]=5

```
TREE-INSERT(x)
  y←nil
  x←root[T]
  while x≠nil
     do y←x
        if key[z]<key[x]
           then x←left[x]
           else x←right[x]
  p[z]←y
  if y=nil
     then root[T]←z
     else if key[z]<key[y]
              then left[y]←z
              else right[y]←z
```



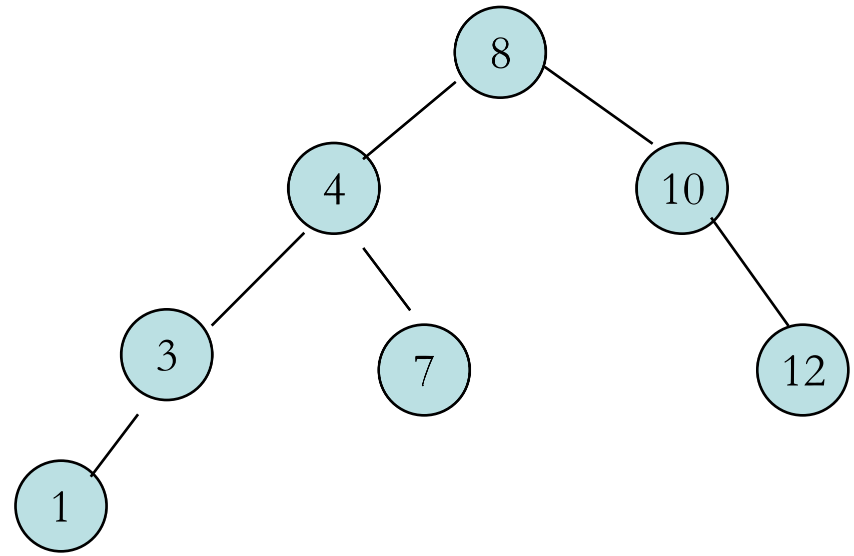
Explique el código para el caso de TREE-INSERT(z), donde key[z]=9

```
TREE-INSERT(x)
  y←nil
  x←root[T]
  while x≠nil
     do y←x
        if key[z]<key[x]
           then x←left[x]
           else x←right[x]
  p[z]←y
  if y=nil
     then root[T]←z
     else if key[z]<key[y]
             then left[y]←z
             else right[y]←z
```



Explique el código para el caso de TREE-INSERT(z), donde key[z]=11

```
TREE-INSERT(x)
  y←nil
  x←root[T]
  while x≠nil
     do y←x
        if key[z]<key[x]
           then x←left[x]
           else x←right[x]
  p[z]←y
  if y=nil
     then root[T]←z
     else if key[z]<key[y]
           then left[y]←z
           else right[y]←z
```

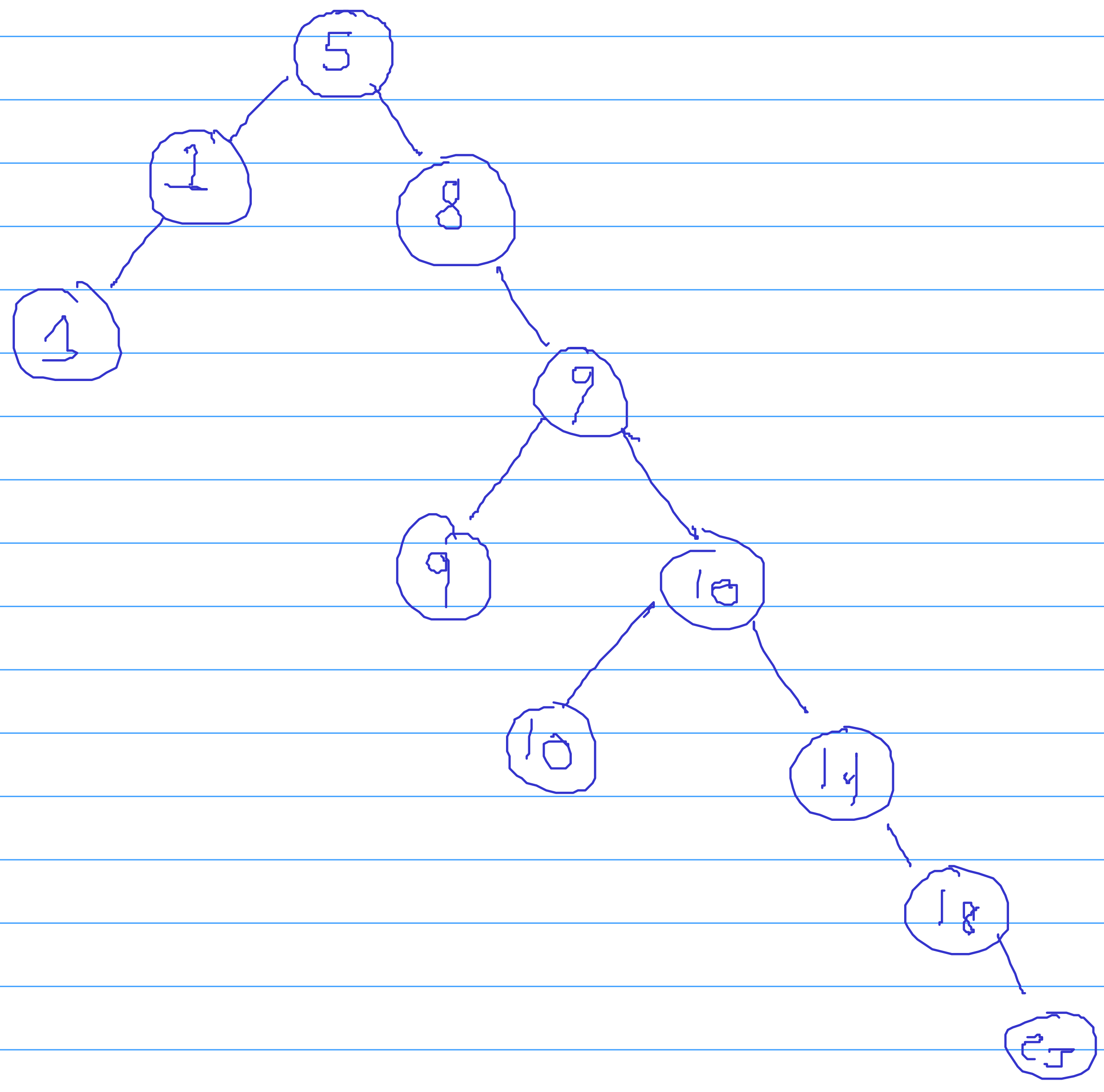


La complejidad es de O(h)

el algoritmo siempre recorre la altura del árbol

Generar Arbol binario Búsqueda

{5, 8, 9, 1, 10, 14, 18, 23, 9, 10, 1}



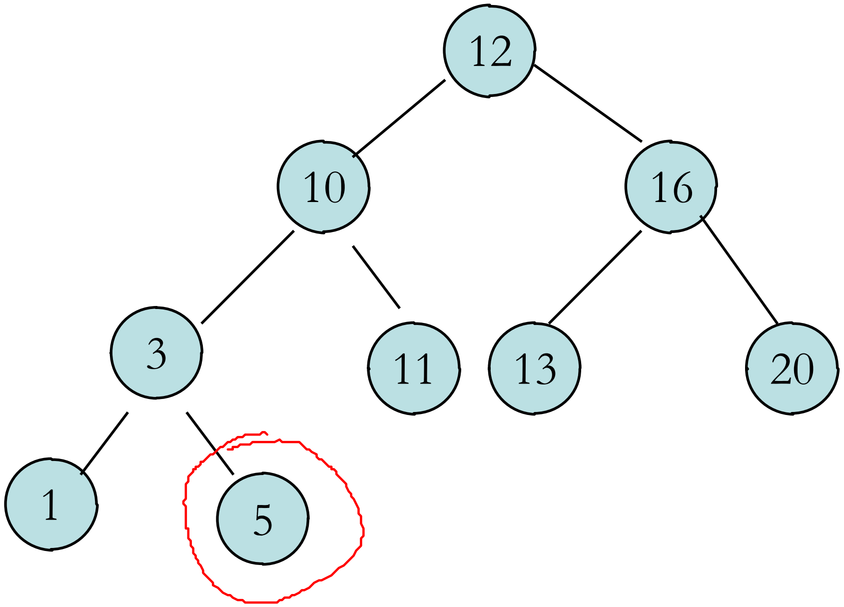
```
TREE-DELETE(x)
 if left[z]=nil or right[z]=nil
    then y←z
    else y←TREE-SUCCESSOR(z)
if left[y]≠nil
    then x←left[y]
    else x←right[y]
if x≠nil
    then p[x]←p[y]
if p[y]=nil
    then root[T]←x
    else if y=left[p[y]]
              then left[p[y]]←x
              else right[p[y]]←x
if y≠z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

Caso 1:

Borrar z y z no tiene hijos.

TREE-DELETE(T,z), donde key[z]=5

Qué se debe hacer?



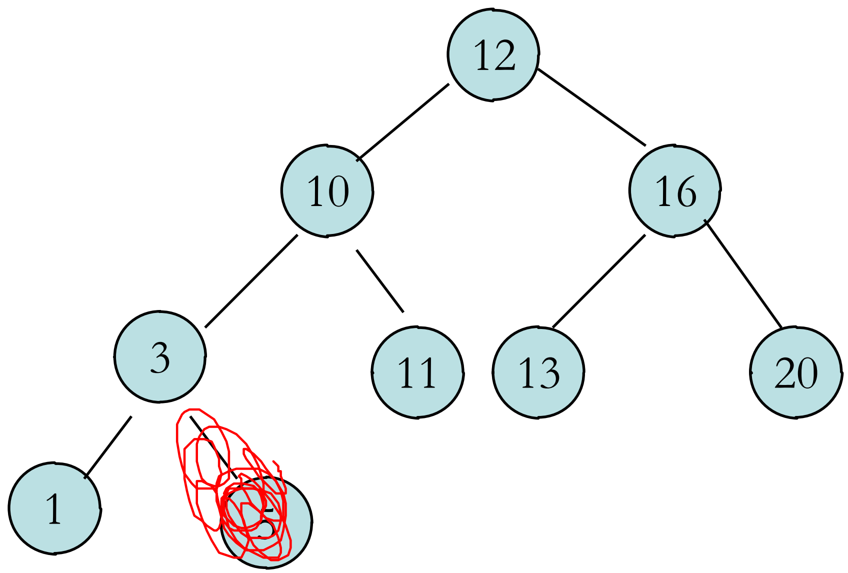
Caso 1:

Borrar z y z no tiene hijos.

TREE-DELETE(T,z), donde key[z]=5

El padre de z debe ahora apuntar a nil

p[z]←nil

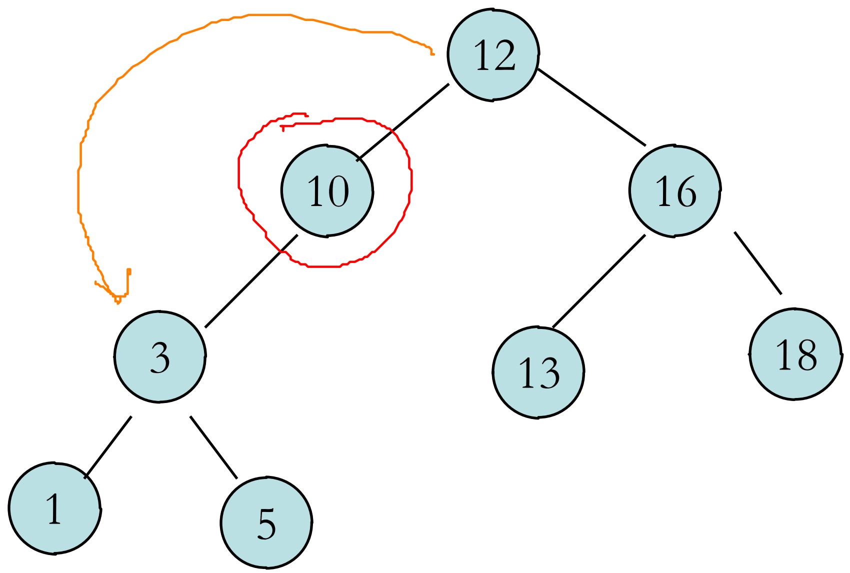


Caso 2:

Borrar z y z tiene un solo hijo

TREE-DELETE(T,z), donde key[z]=10

Qué se debe hacer?

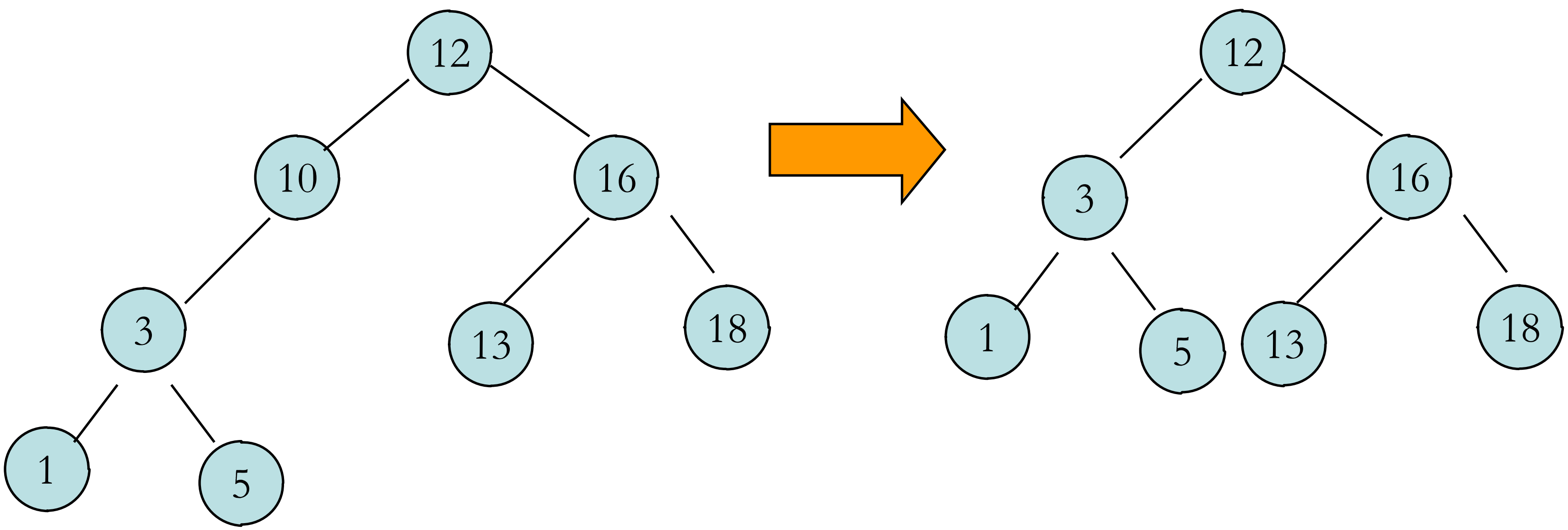


Caso 2:

Borrar z y z tiene un solo hijo

TREE-DELETE(T,z), donde key[z]=10

Se separa z del árbol

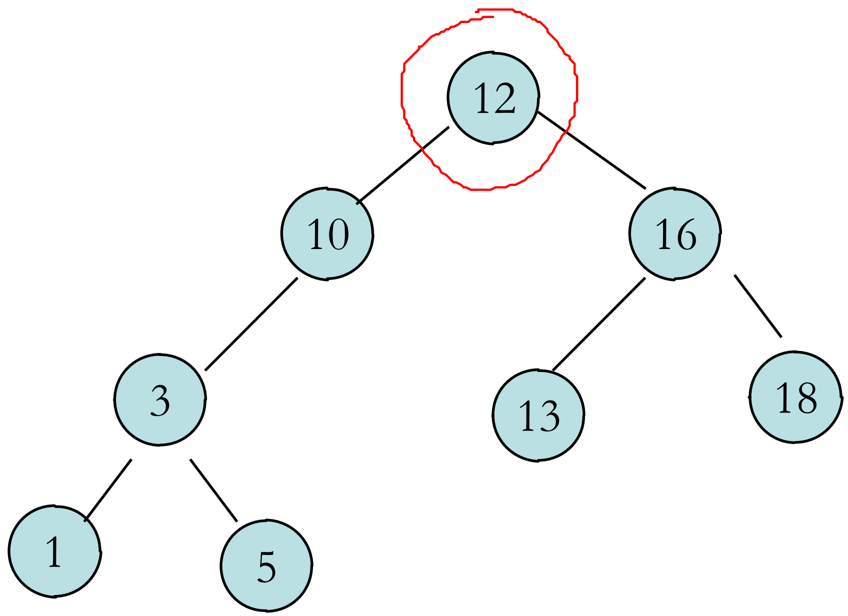


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T,z), donde key[z]=12

Qué se debe hacer?

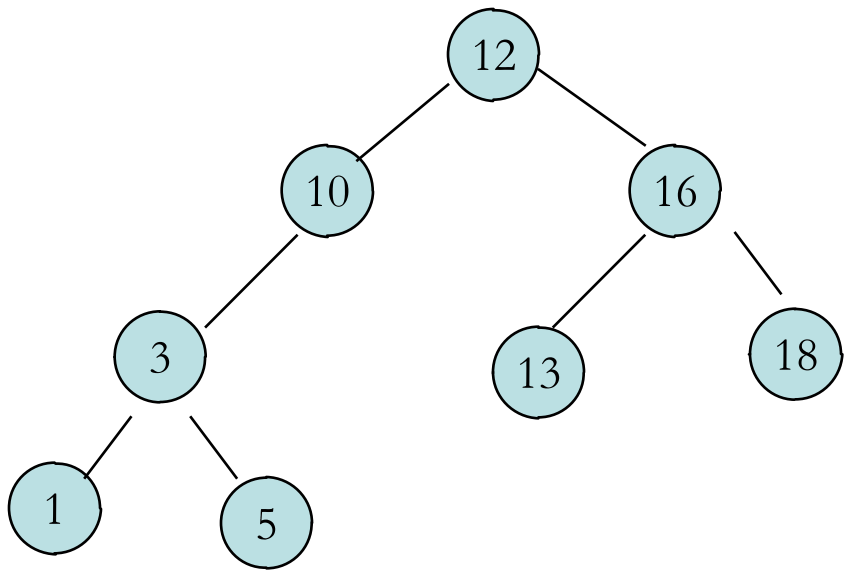


Caso 3:

Borrar z y z tiene dos hijos

TREE-DELETE(T,z), donde key[z]=12

Qué se debe hacer?



Caso 3:

Borrar z y z tiene dos hijos

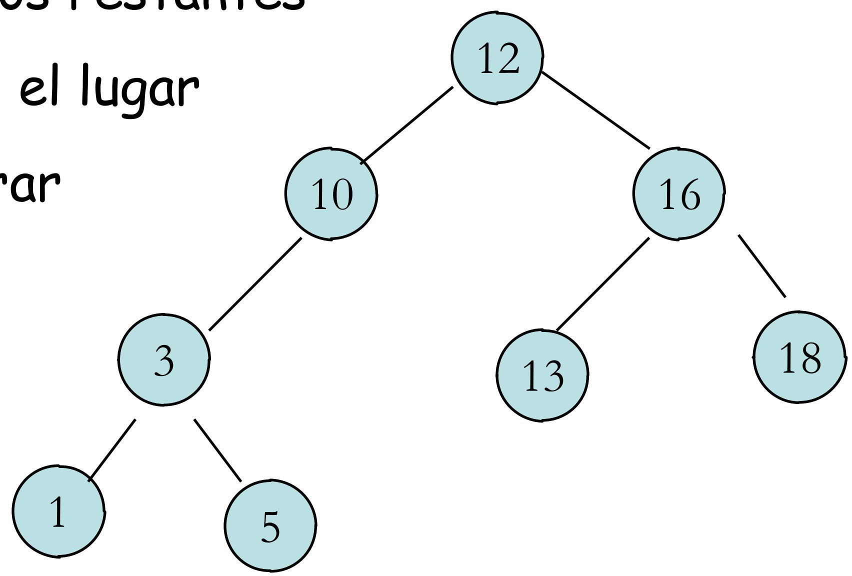
TREE-DELETE(T,z), donde key[z]=12

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

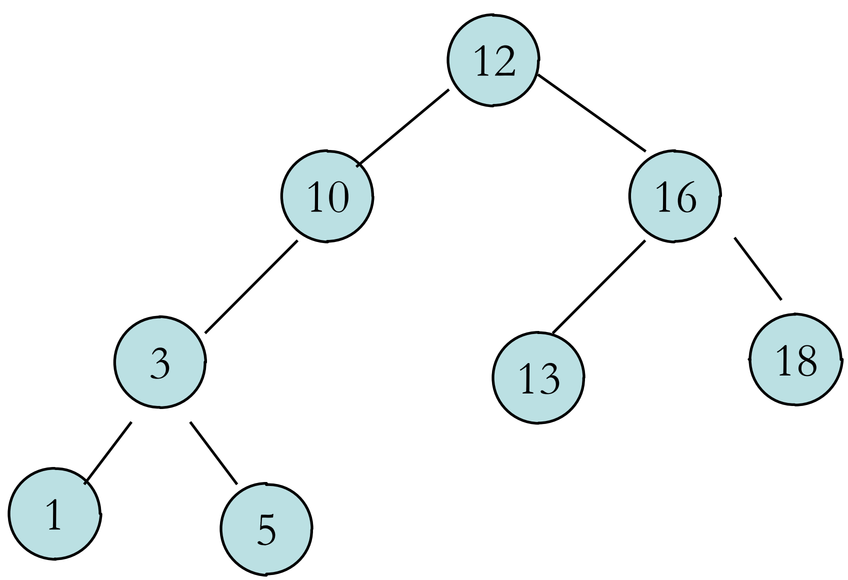
TREE-DELETE(T,z), donde key[z]=12

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

Borrar

Caso 1

Caso 2



Caso 3:

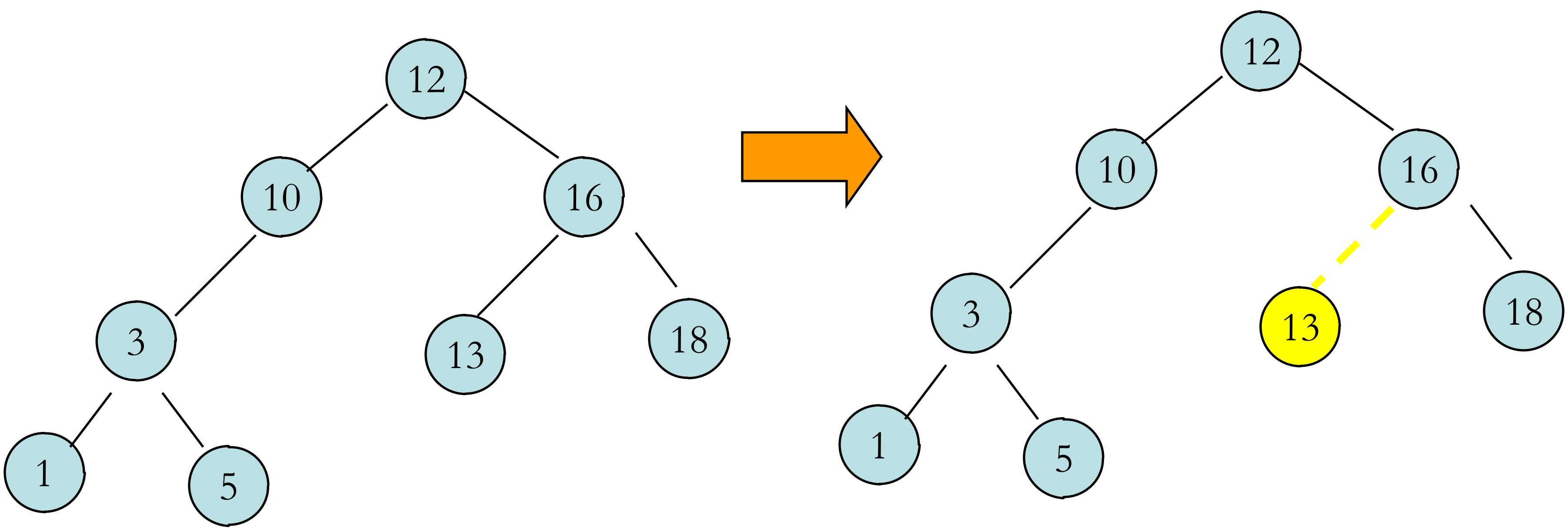
Borrar z y z tiene dos hijos

TREE-DELETE(T,z), donde key[z]=12

Se separa(elimina) su sucesor y del árbol

y se reemplaza su contenido con el

de z



Caso 3:

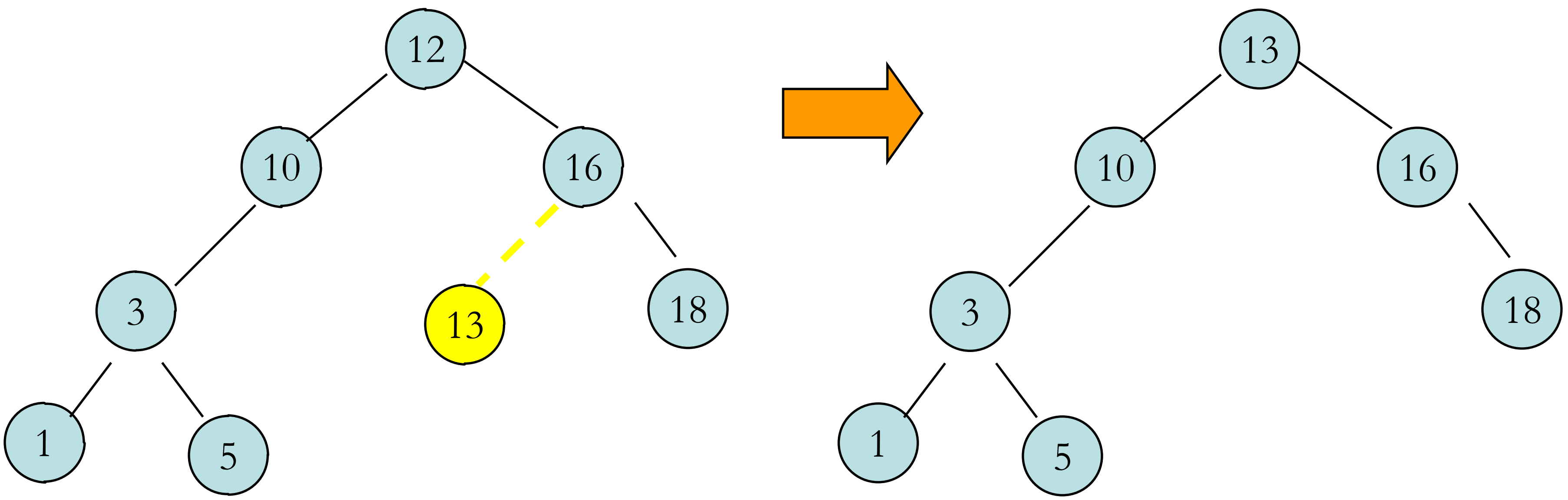
Borrar z y z tiene dos hijos

TREE-DELETE(T,z), donde key[z]=12

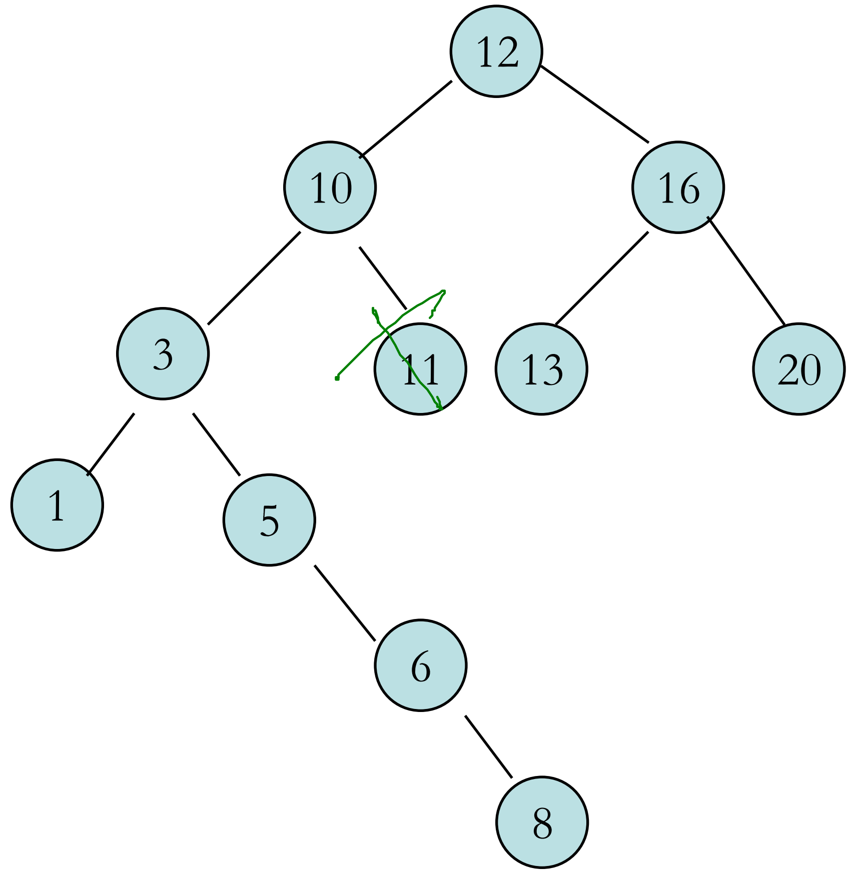
Se <u>separa(elimina)</u> su sucesor y del árbol

y se reemplaza su contenido con el

de z

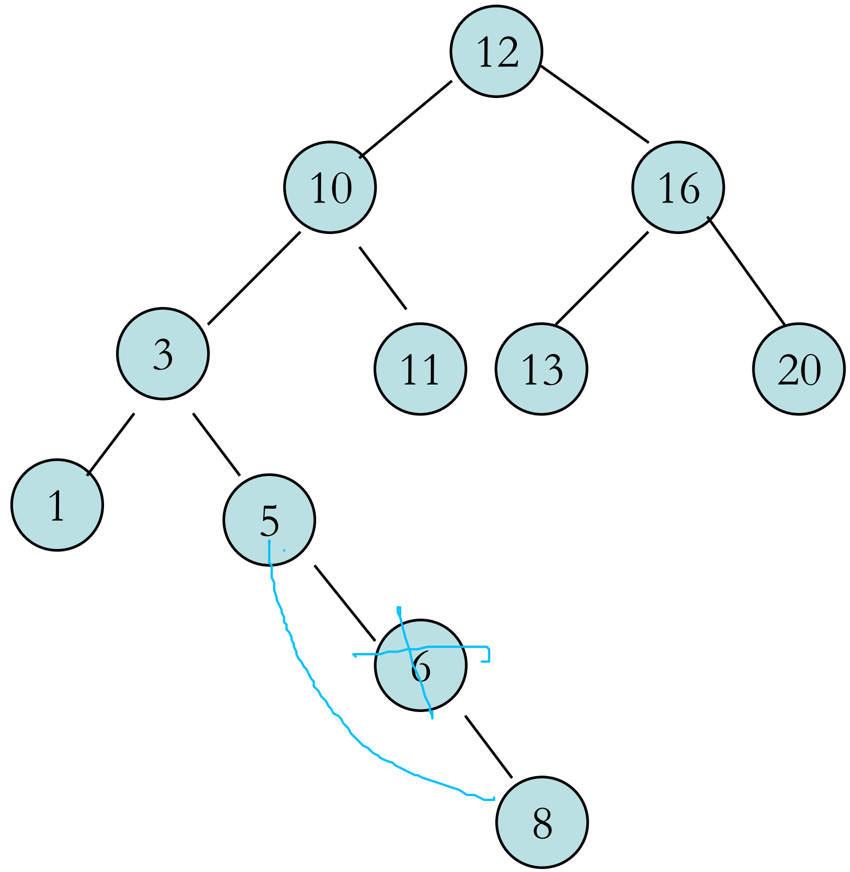


```
TREE-DELETE(x)
 if left[z]=nil or right[z]=nil
    then y←z
    else y←TREE-SUCCESSOR(z)
if left[y]≠nil
    then x←left[y]
    else x←right[y]
if x≠nil
    then p[x]←p[y]
if p[y]=nil
    then root[T]←x
    else if y=left[p[y]]
              then left[p[y]]←x
              else right[p[y]]←x
if y≠z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



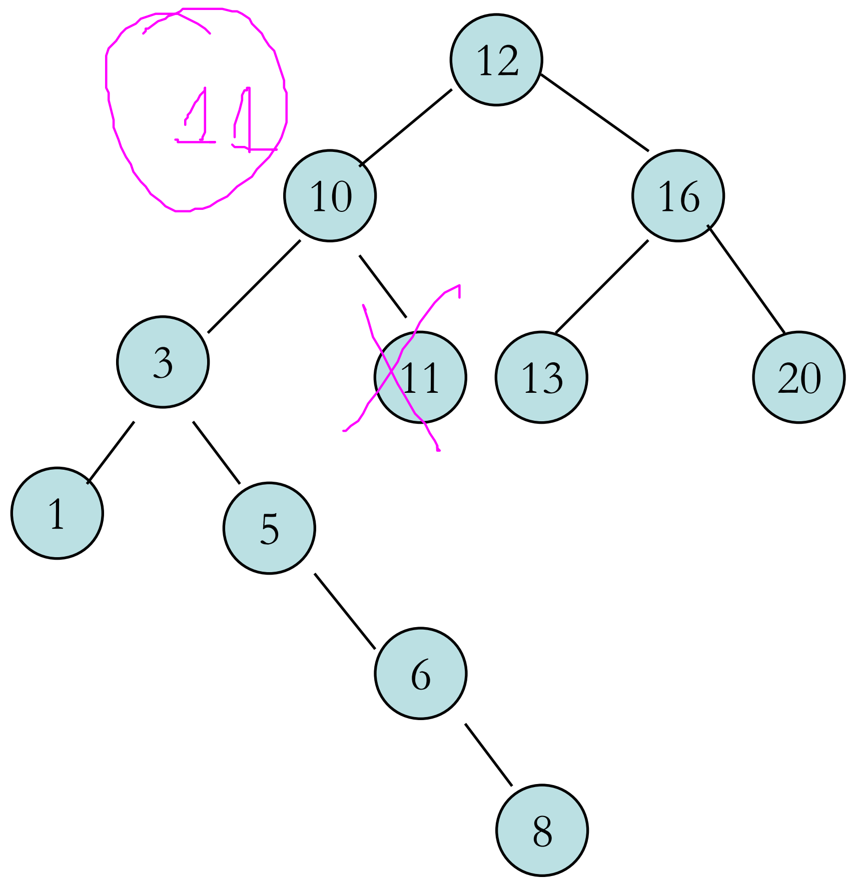
Siga el algoritmo TREE-DELETE(T,z) donde z es el nodo tal que key[z]=11

```
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]=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
```

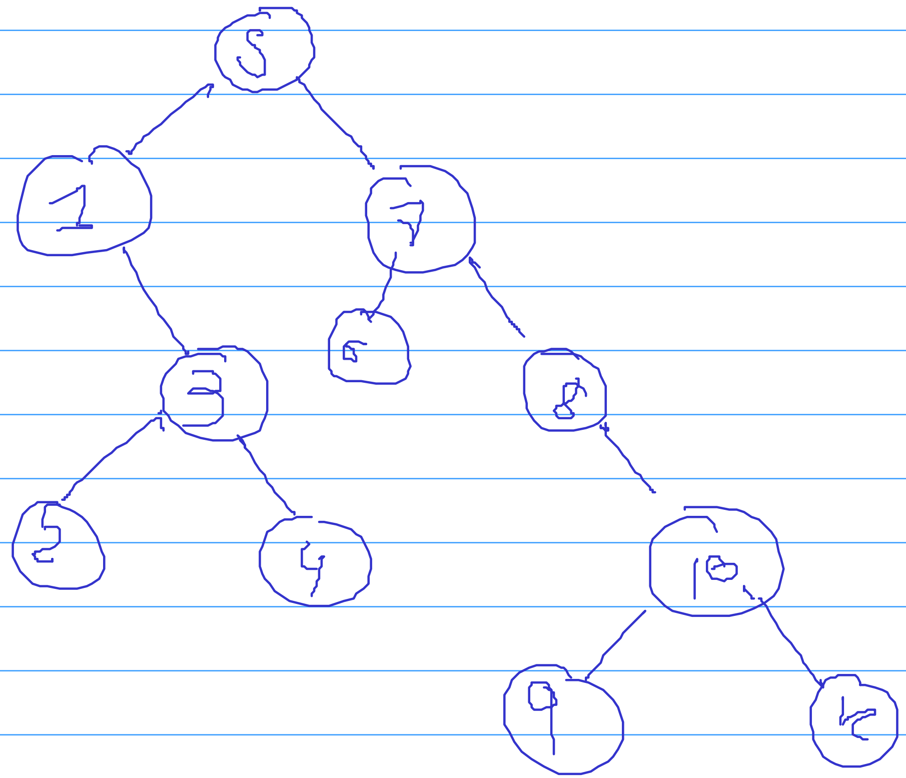


Siga el algoritmo TREE-DELETE(T,z) donde z es el nodo tal que key[z]=10

Crear Arbol

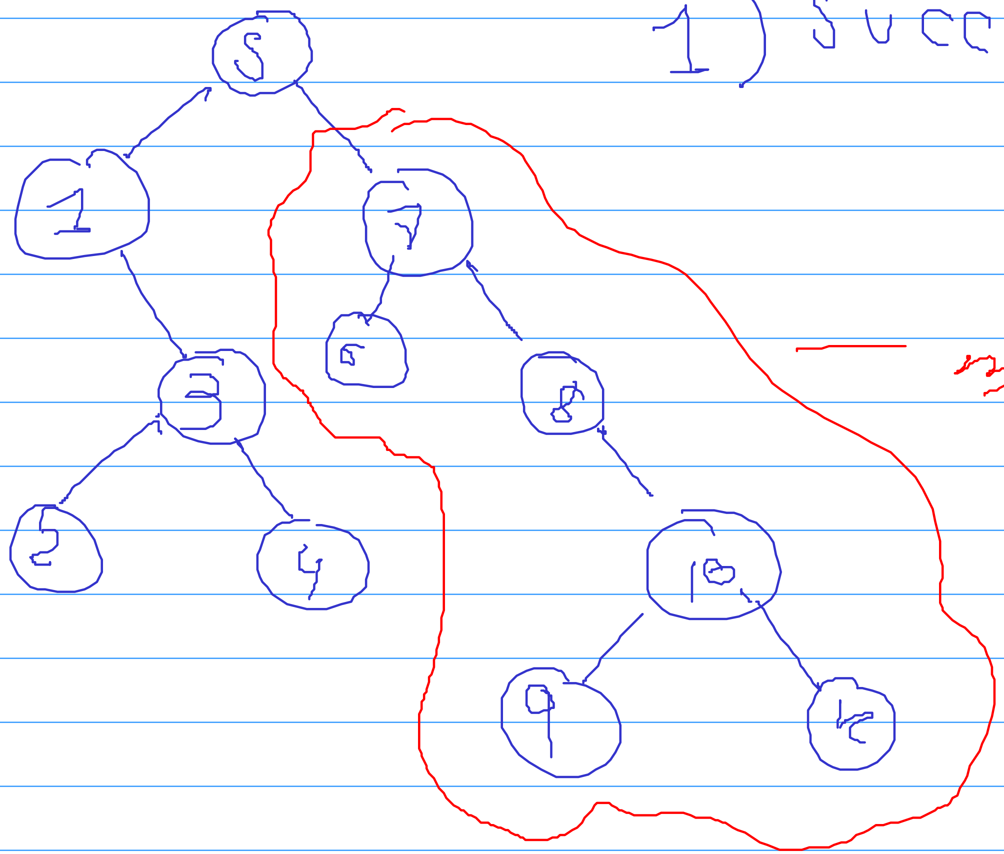
{5, 7, 1, 3, 4, 8, 6, 2, 10, 12, 9}

Borrar {5, 4, 8, 9}

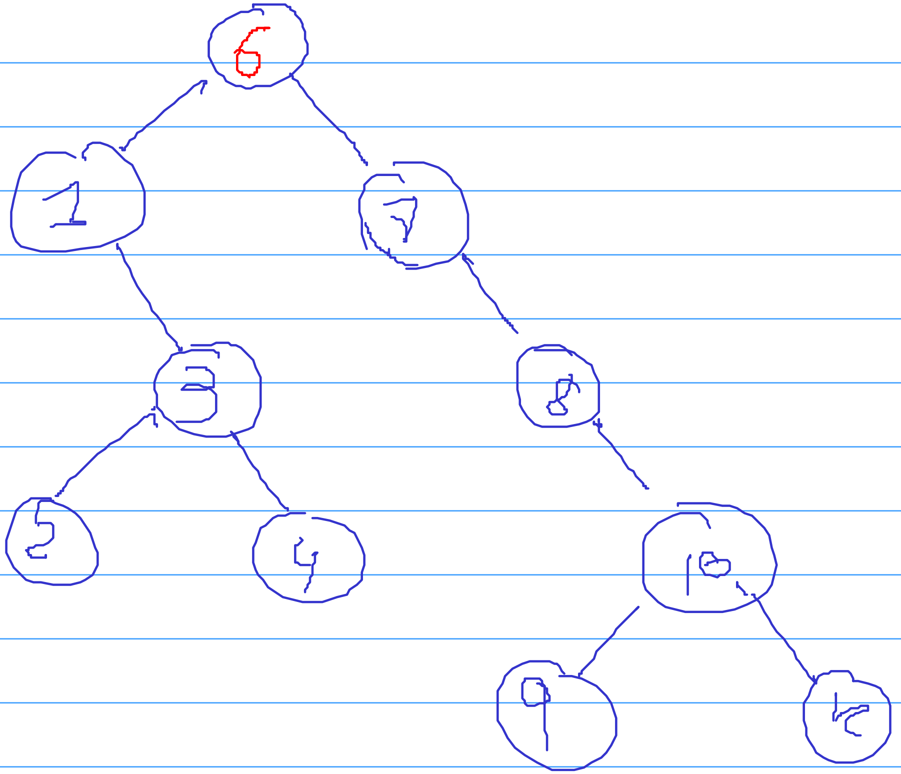


Eliminar (5)

1) succ (5) = 6

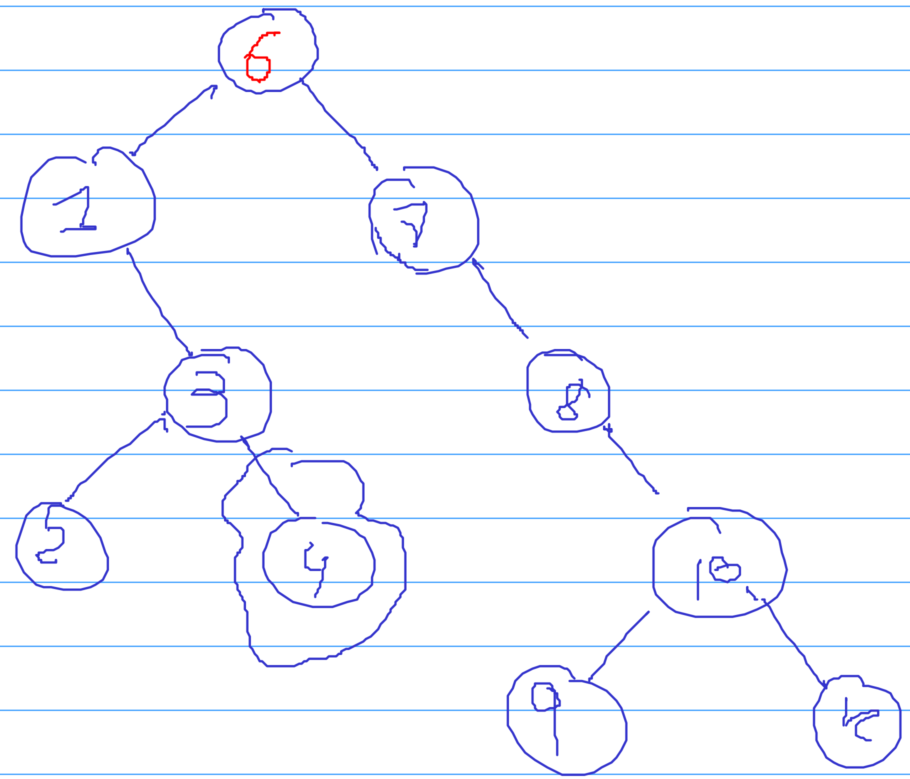


min r 6

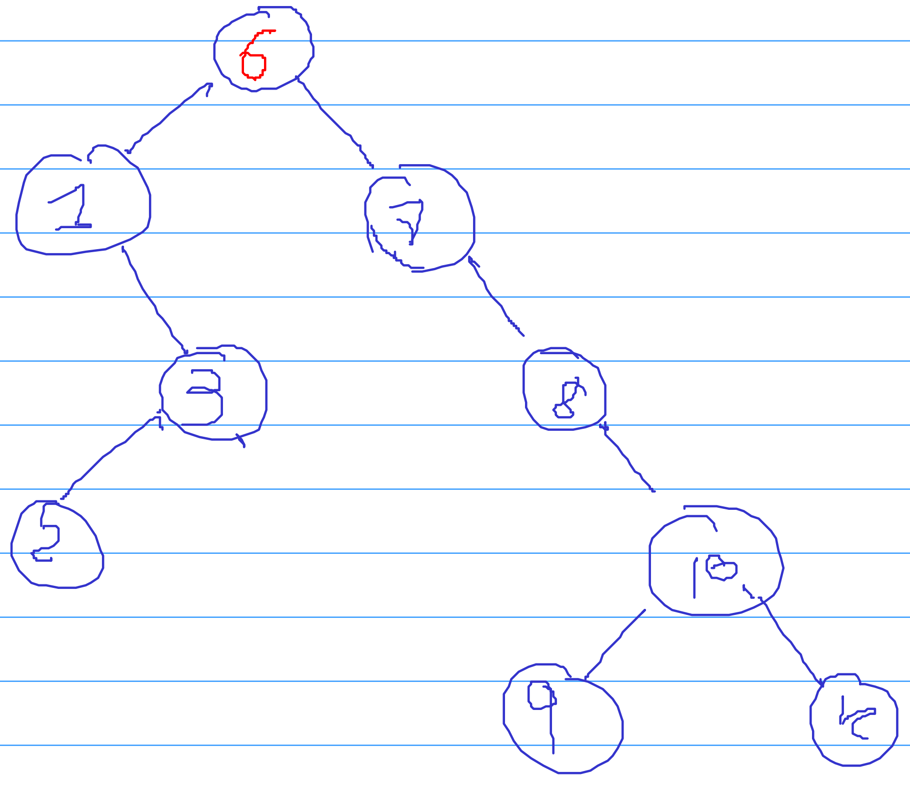


6
1
7
3
4
8
10
9

Elim (4)



Elim (8)



Elim (9)