Estructura de Datos

Maria C. Torres

Maria C. Torres

Ing. Electrónica (UNAL)
M.E. Ing. Eléctrica (UPRM)
Ph.D. Ciencias e Ingeniería de la Computación y la Información (UPRM)
Profesora asociada
Dpto. Ciencias de la Computación y la Decisión

mctorresm@unal.edu.co

HORARIO DE ATENCIÓN: Martes 10:00 am a 12:00 m – Oficina 313 M8A

Contenido del Curso

- ❑ Introducción: revisión fundamentos y POO
- ❑ Análisis de complejidad
- ❑ Arreglos
- ❑ Listas enlazadas
- ❑ Pilas y colas
- ❑ Heap
- ❑ **Arboles binarios**
- ❑ Tablas hash
- ❑ Grafos

Árbol Binario de Búsqueda

- Arboles
- Arboles binarios
- Arboles binarios de búsqueda
- **Arboles balanceados**

Arboles balanceados

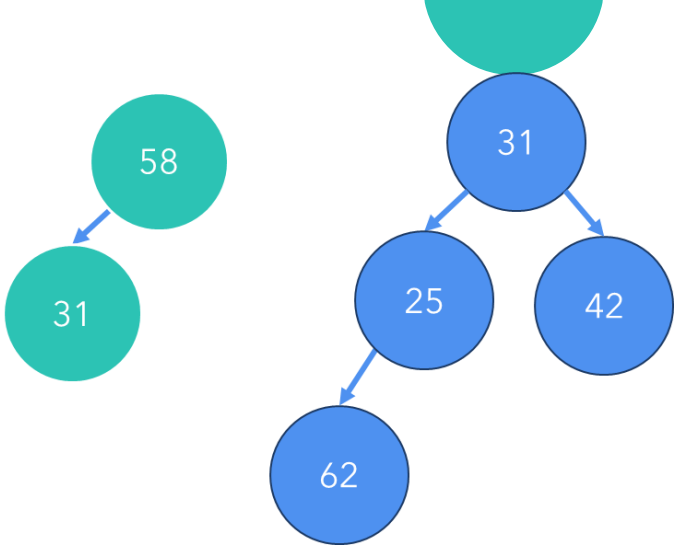
Definición:

Un árbol binario es balaceado si la altura del árbol es $O(\log n)$.

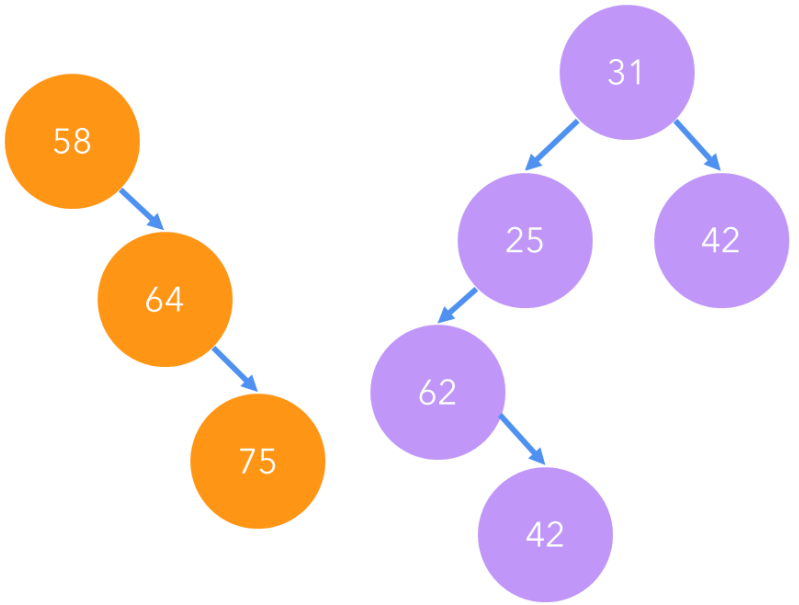
Existen diferentes tipos de árboles balanceado, los cuales emplean diferentes estrategias para garantizar que la altura del árbol. Por ejemplo:

- ❑AVL
- ❑Red-Black-Tree
- ❑B-tress

Arboles balanceados



Arboles no balanceados

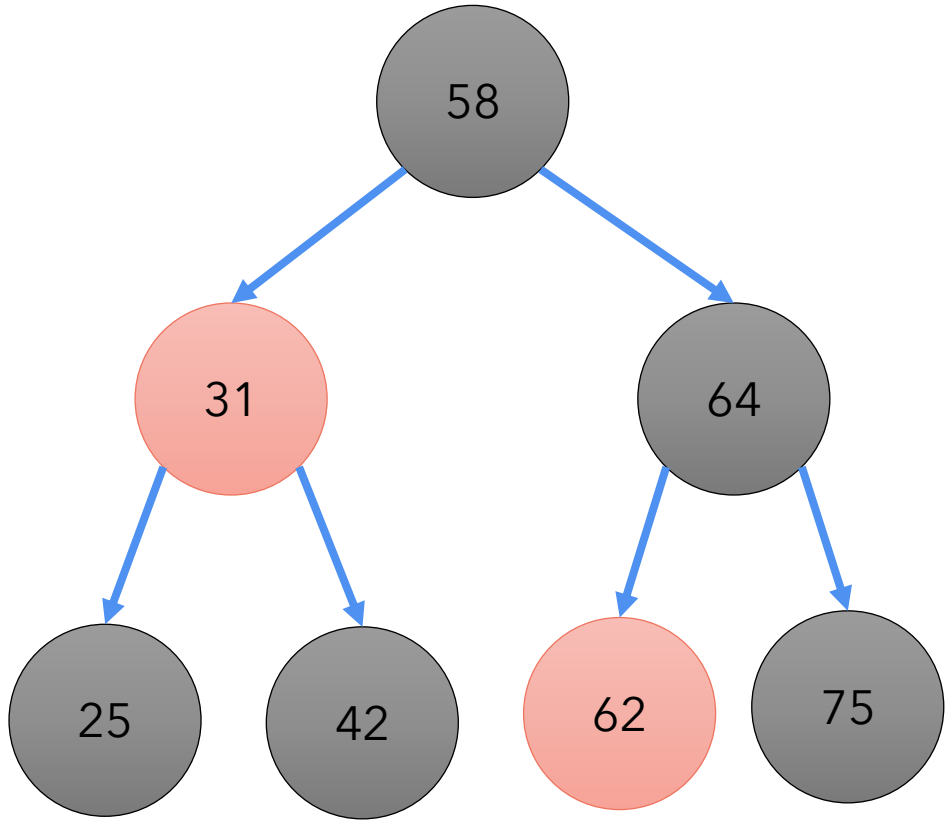


Arboles balanceados - RBT

RED-BLACK-TREE

Definición:

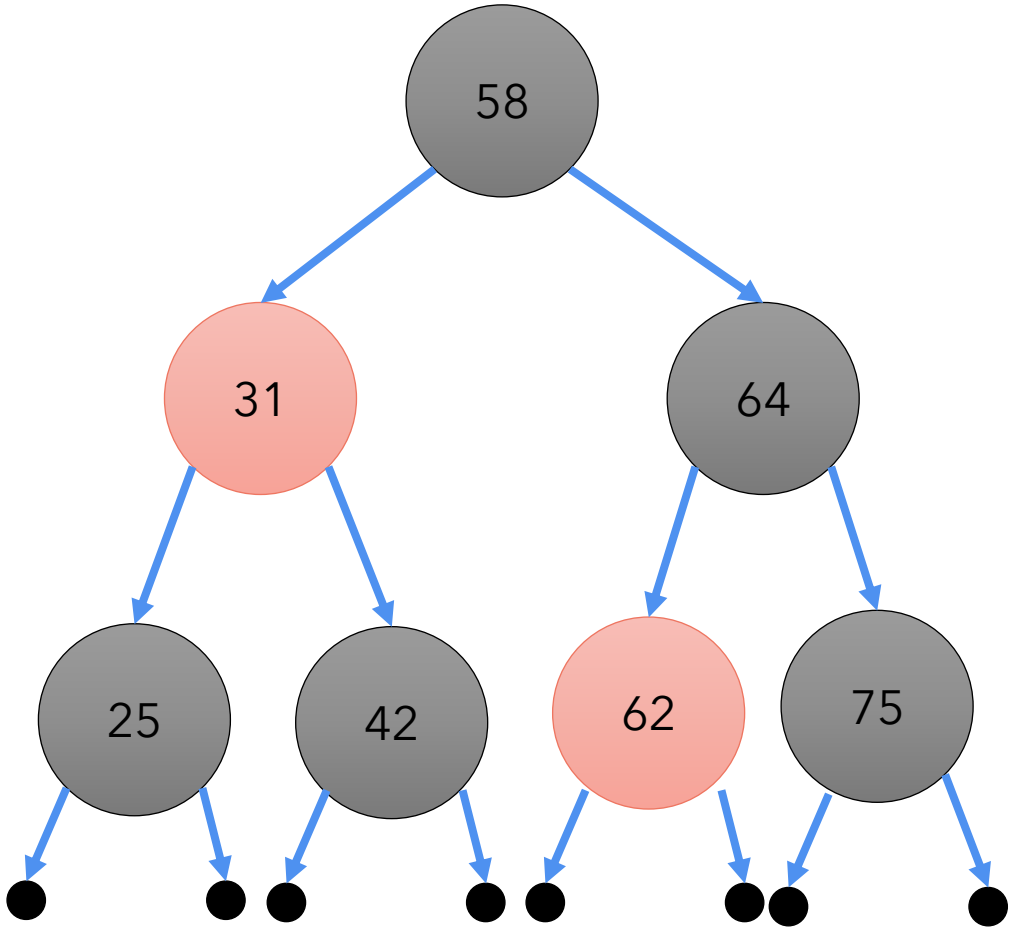
- Es un árbol binario de búsqueda ABB con un atributo adicional por nodo: su color
- Cada nodo puede ser RED (rojo) o BLACK (negro)
- Limitando el número de nodos de un color en cada rama del árbol, los árboles RED-BLACK buscan aproximar un árbol balanceado
- Es decir, es una aproximación a un árbol completo



Arboles balanceados - RBT

Para la construcción se agregan como nodos externos **centinelas**, que no almacenan datos.

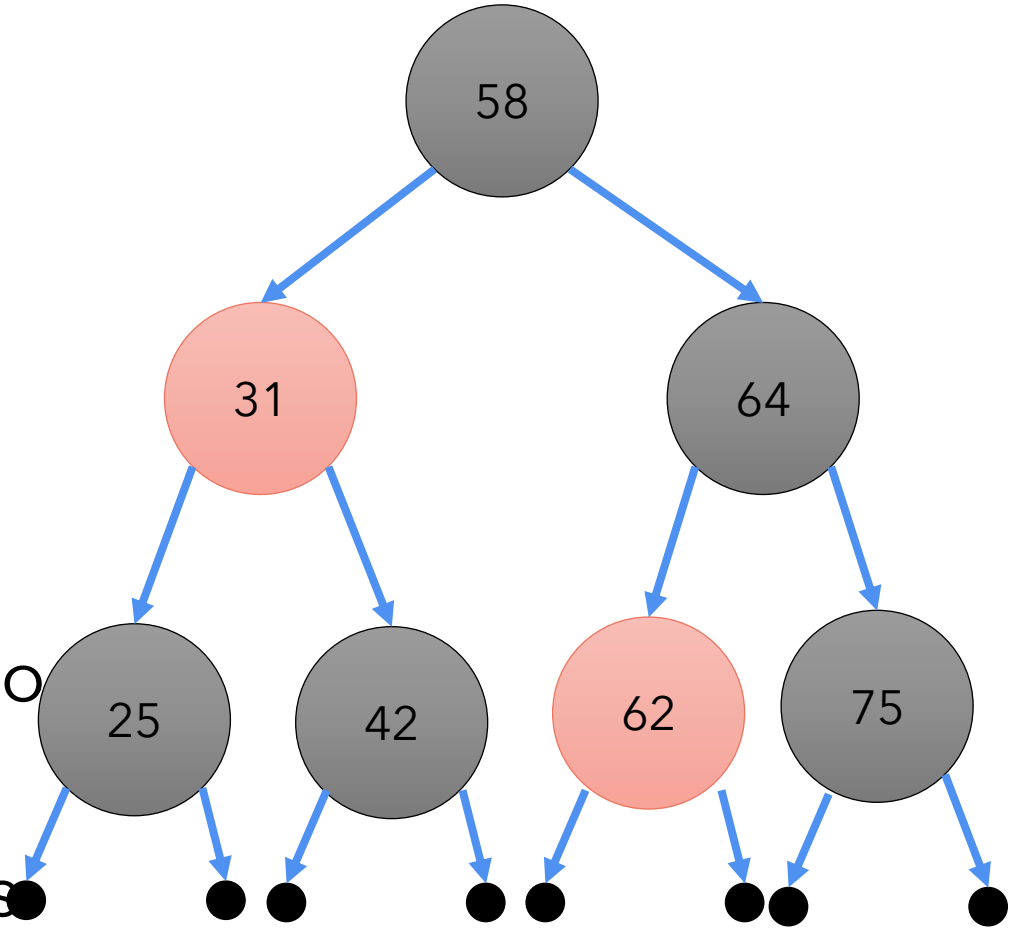
- Todos los datos se encuentran en los nodos internos.



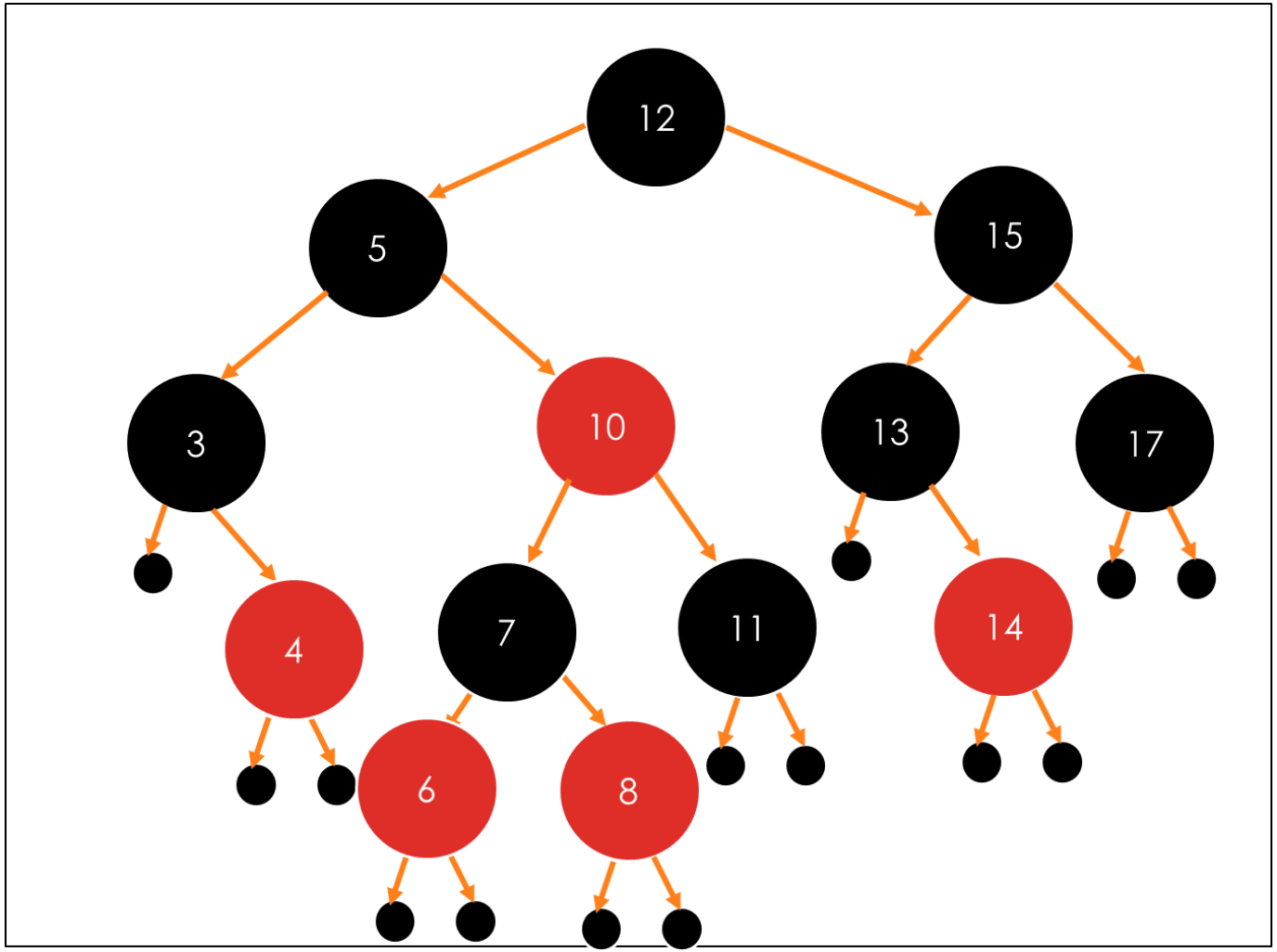
Arboles balanceados - RBT

Propriedades

- Propiedad de la raíz: la raíz es negra
- Propiedad de las hojas: los nodos externos son negros
- Propiedad de los nodos internos: Los hijos de un nodo rojo son negros
- Propiedad de la profundidad: Todos lo nodos externos tienen la misma profundidad negra (Black Depth), definida como el número de ancestros negros menos uno

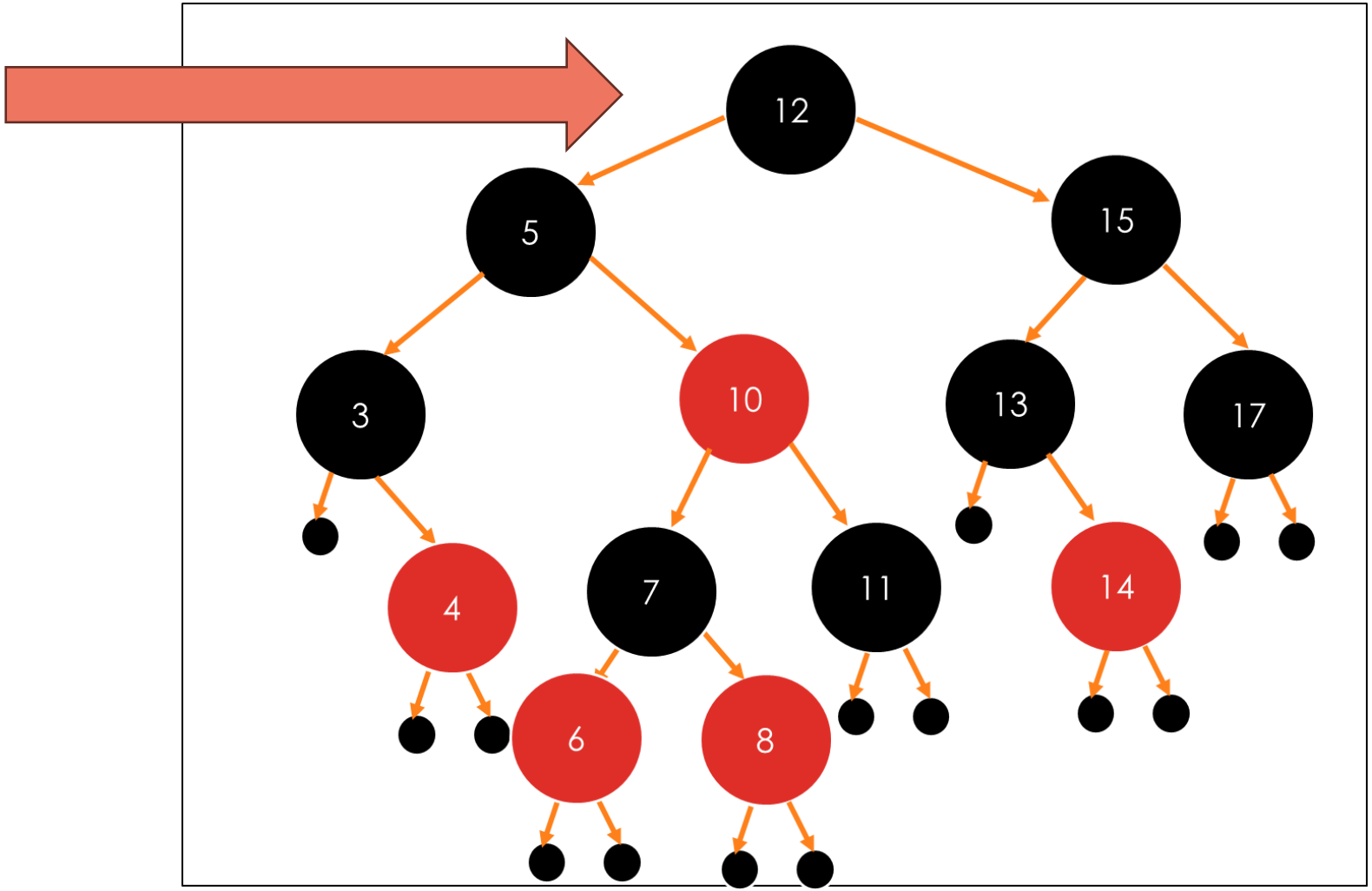


Arboles balanceados - RBT



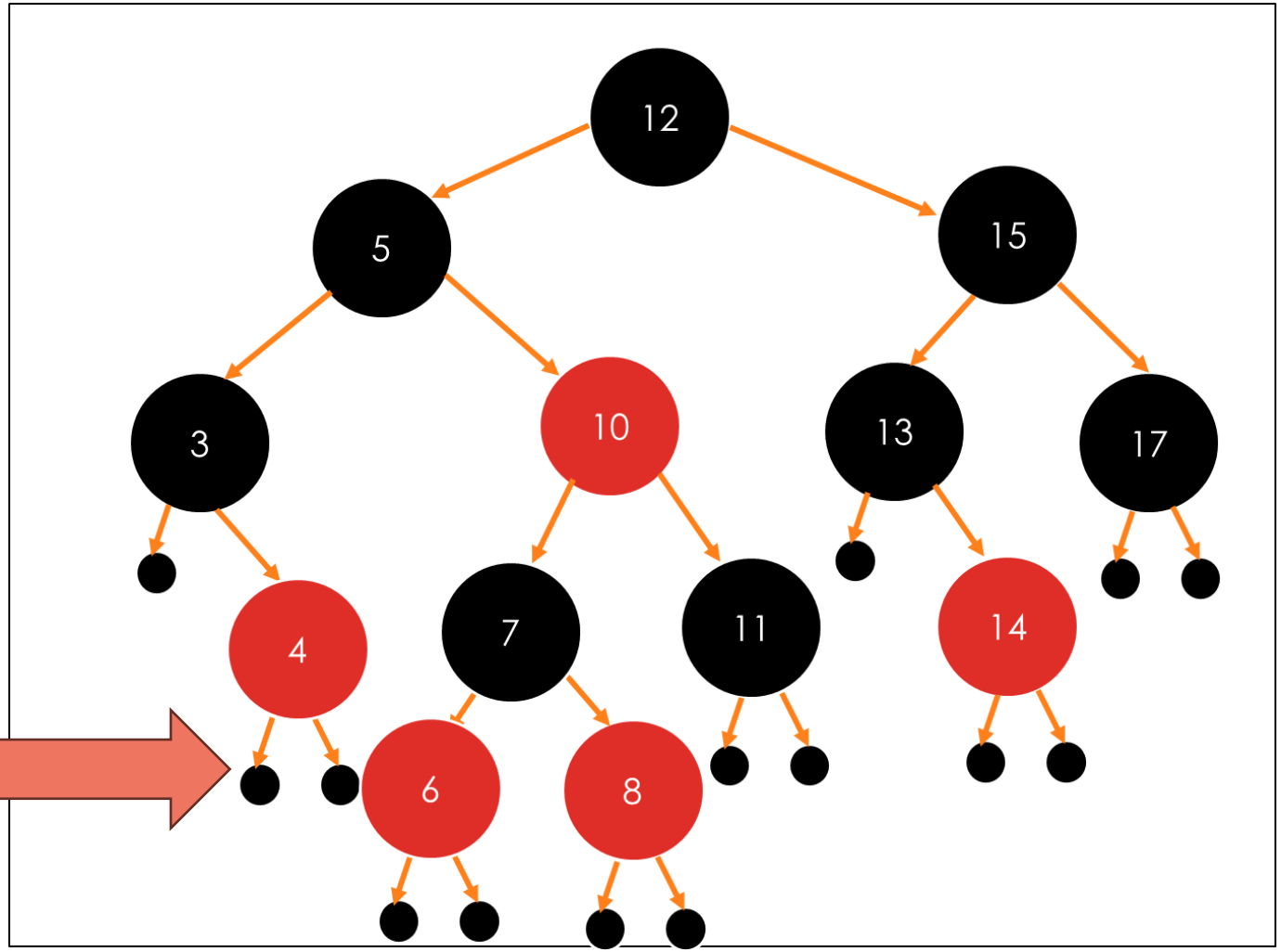
Arboles balanceados - RBT

- Propiedad de la raíz:
  la raíz es negra



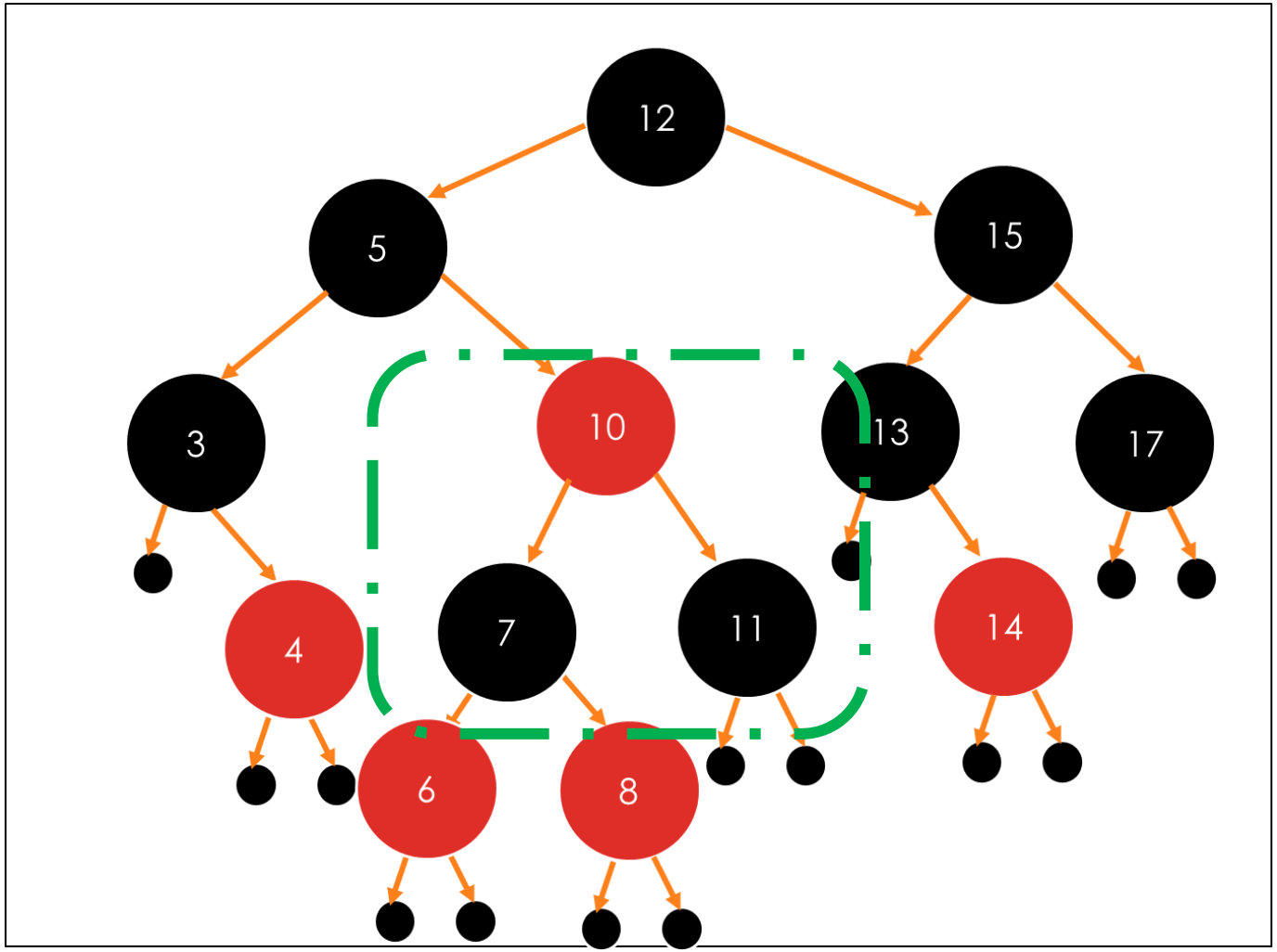
Arboles balanceados - RBT

- Propiedad de las hojas: los nodos externos son negros



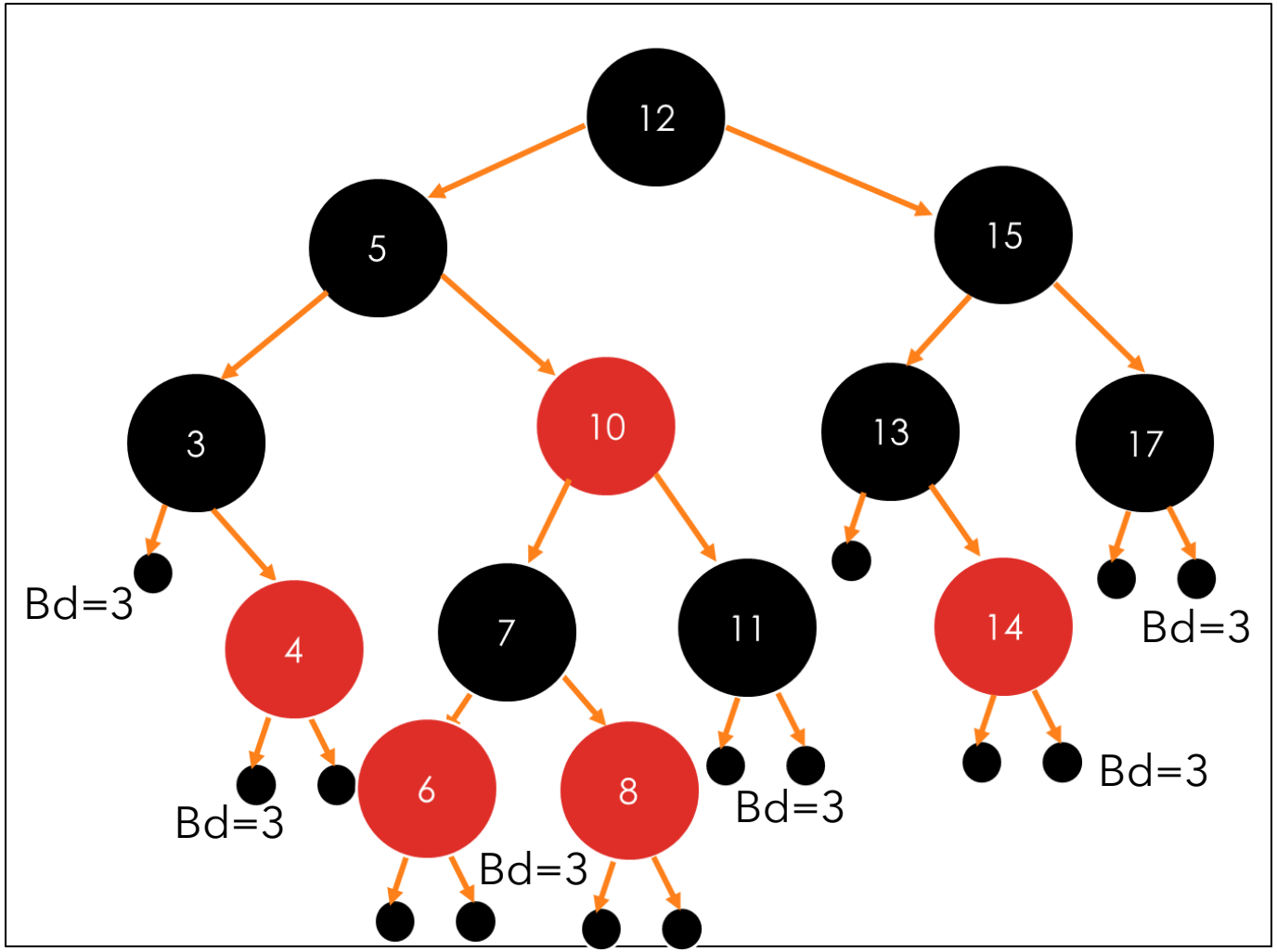
Arboles balanceados - RBT

- Propiedad de los nodos internos: Los hijos de un nodo rojo son negros



Arboles balanceados - RBT

- Propiedad de la profundidad: Todos los nodos externos tienen la misma profundidad negra (Black Depth), definida como el número de ancestros negros menos uno



Arboles balanceados - RBT

Teorema:

Un RBT con n nodos internos tiene una altura menor o igual a $2\ln(n+1)$.

- Es decir, la altura del arboles es $h(T) = O(\ln n))$

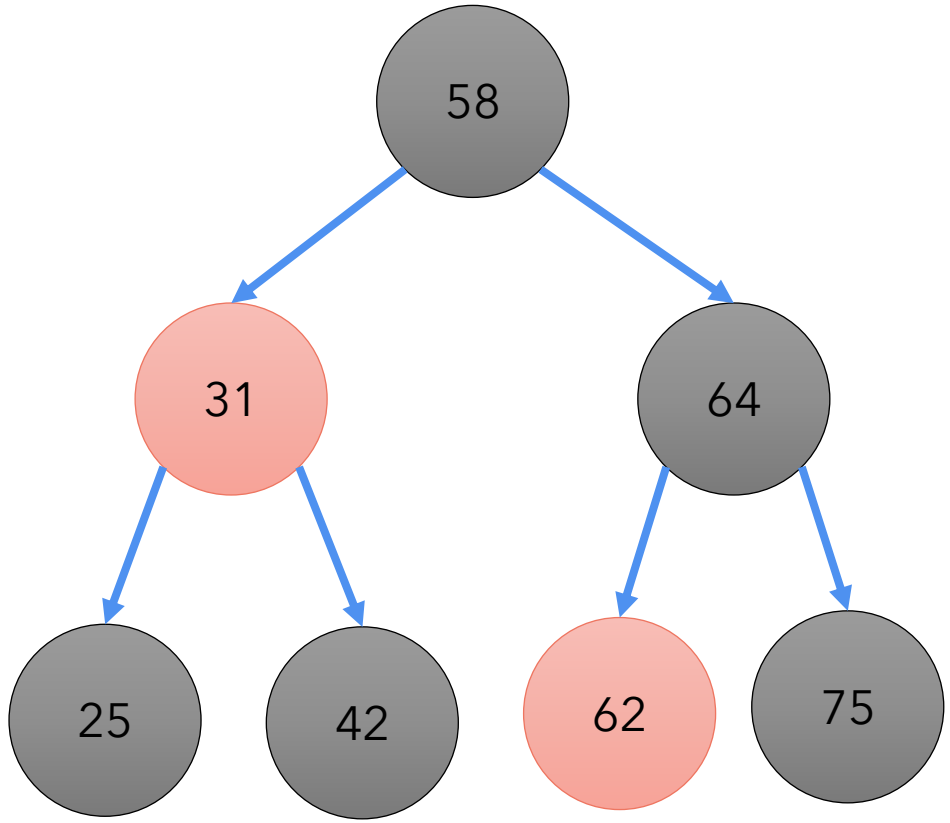
Operaciones ABB

Operación	Complejidad
parent()	O(n)
depth()	$O(n^2)$ Árbol completo: O(n lg n)
height()	O(n) Árbol completo: O(lg n)
addRoot()	Θ(1)
insertLeft()	Θ(1)
insertRight()	Θ(1)
remove()	O(n) Árbol completo: O(lg n)
Find()	Árbol completo: O(h) Árbol completo: O(lg n)
Insert()	Árbol completo: O(h) Árbol completo: O(lg n)

Arboles balanceados - RBT

OPERACIONES

- Rotar: mueve los nodos para mantener las propiedades del RBT
  - Rotación a la izquierda
  - Rotación a la derecha
- Insertar: agrega un nuevo elemento al RBT manteniendo las propiedades – hace uso de las rotaciones
- Eliminar: remueve un nodo del RBT manteniendo las propiedades – hace uso de las rotaciones



Arboles balanceados - RBT

Implementación

- Nodo de RBT: almacena un objeto, una clave y el color del nodo
  - El color se representa por una variable booleana (TRUE = ROJO, FALSE=NEGRO)
- Para manejar los datos creamos BSTEntry

RBEntry
#data: Object #k:int # color: Boolean
+RBEntry(Object d, int k) +getData(): Object +getKey(): int +getColor(): Boolean +setData(Object d) +setKey(int k) +setColor(Boolen c)

```
BSTEntry(Object e, int k)
  data = e
  this.k = k
  color = FALSE
```

```
setData(Object d)
  data = d

setKey(int k)
  this.k = k
```

```
getData()
  return data

getKey()
  return key
```

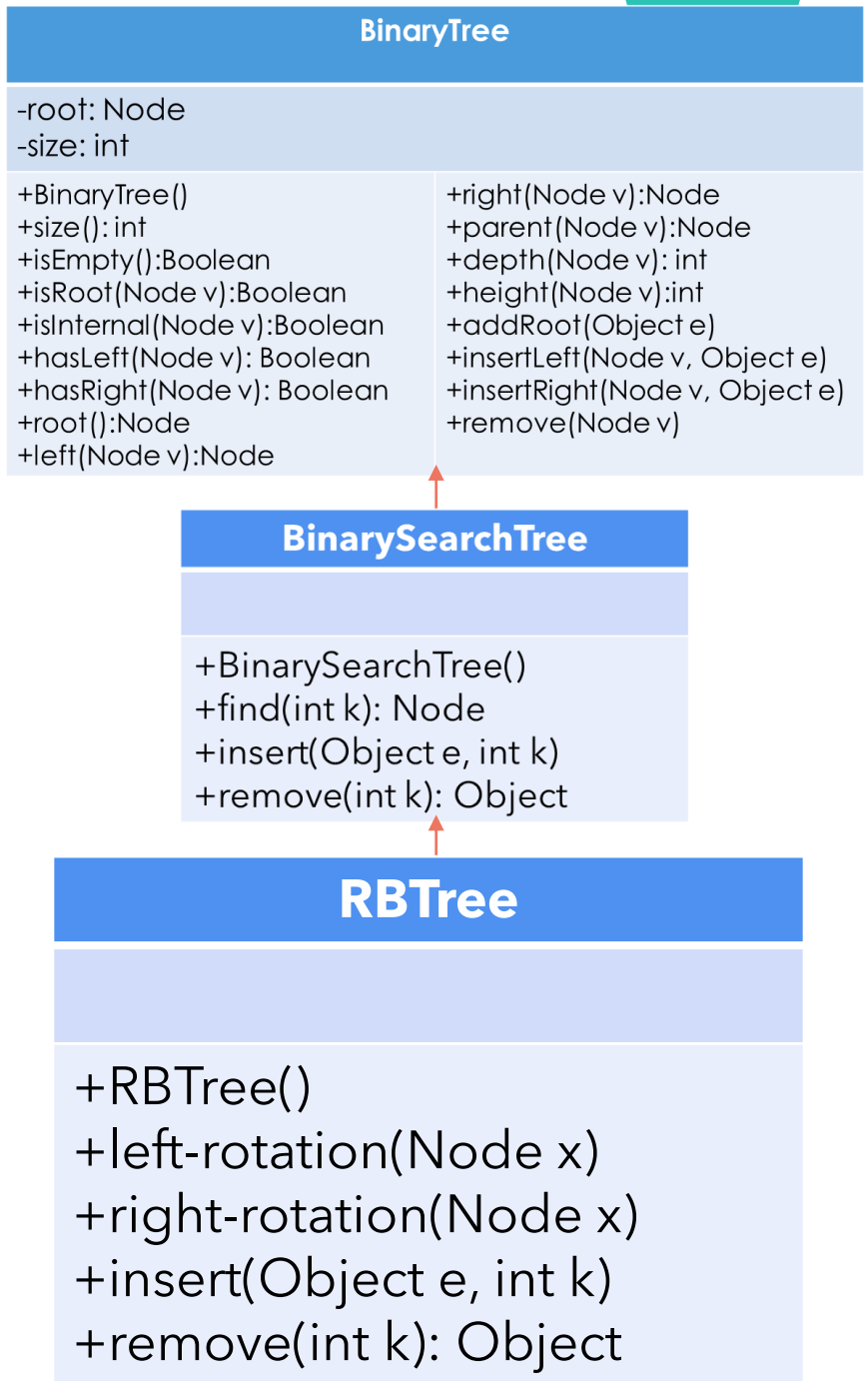
```
getColor()
  return color

setKey(Boolean c)
  color = c
```

Arboles balanceados - RBT

Implementación

- La clase RBTree puede ser una extensión de nuestras clases BinaryTree y BinarySearchTree
- Se debe agregar las operaciones de rotaciones y actualizar las operaciones para agregar y eliminar un dato



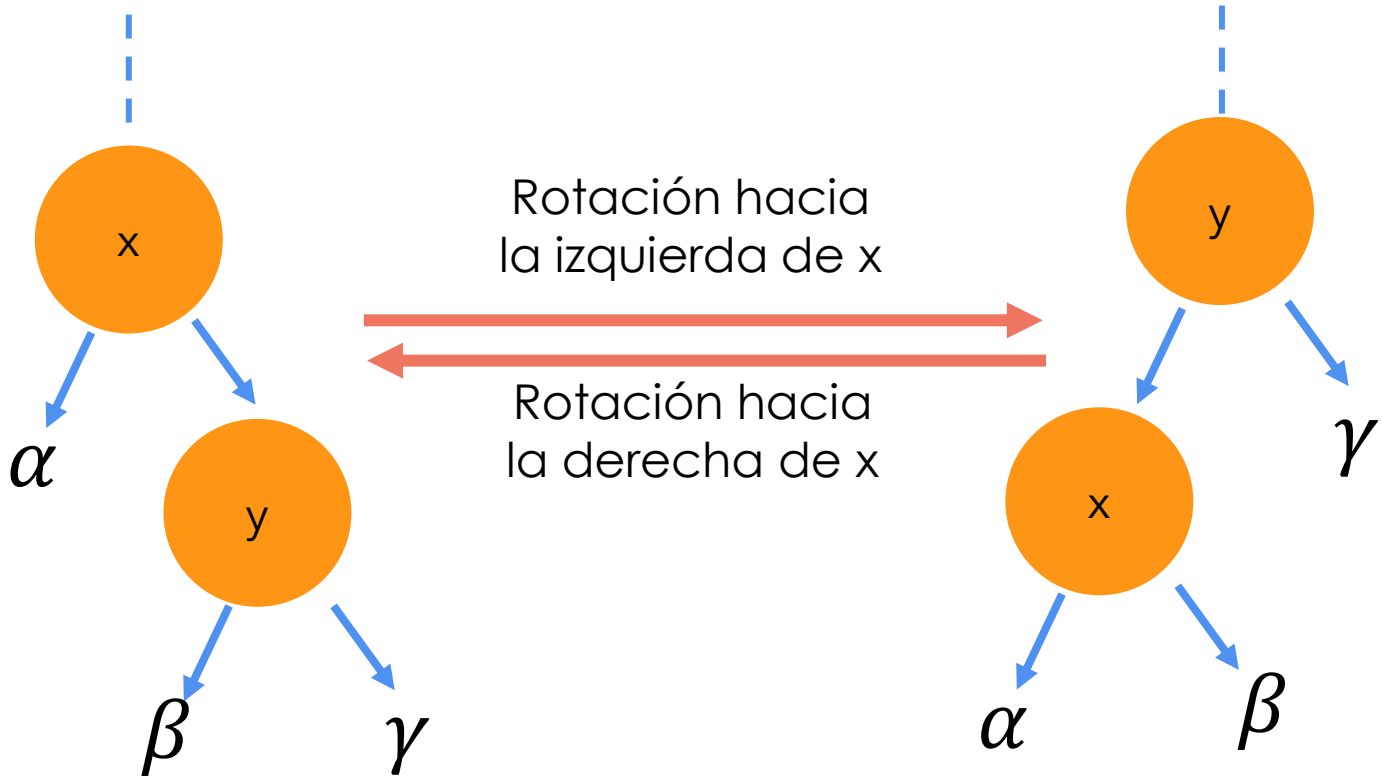
Arboles balanceados - RBT

ROTACIONES

- Cuando se realizan las operaciones de insertar y eliminar, el resultado puede no cumplir con las propiedades de un árbol RED-BLACK
- Para restaurar estas propiedades, debemos cambiar el color de uno o varios nodos, y debemos cambiar algunas conexiones en la estructura del árbol
- Para realizar los cambios de conexiones se realizan operaciones de rotación
- La rotación es una operación local que preserva la propiedad de un árbol binario de búsqueda
- Existen dos tipos de rotaciones
  - Rotación hacia la izquierda
  - Rotación hacia la derecha

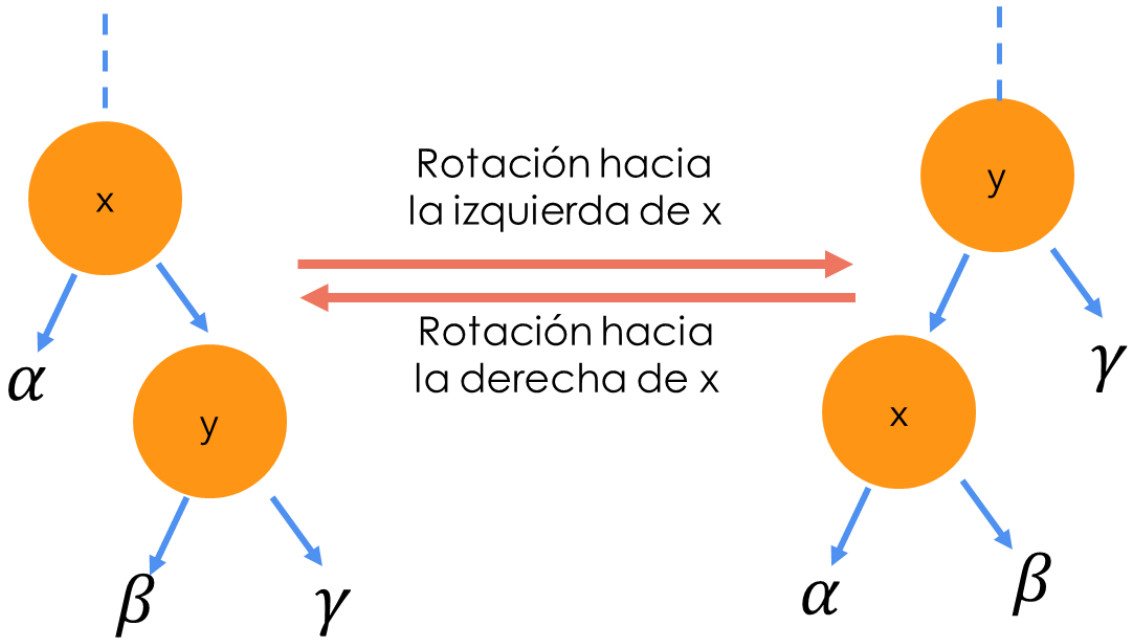
Arboles balanceados - RBT

ROTACIONES



Arboles balanceados - RBT

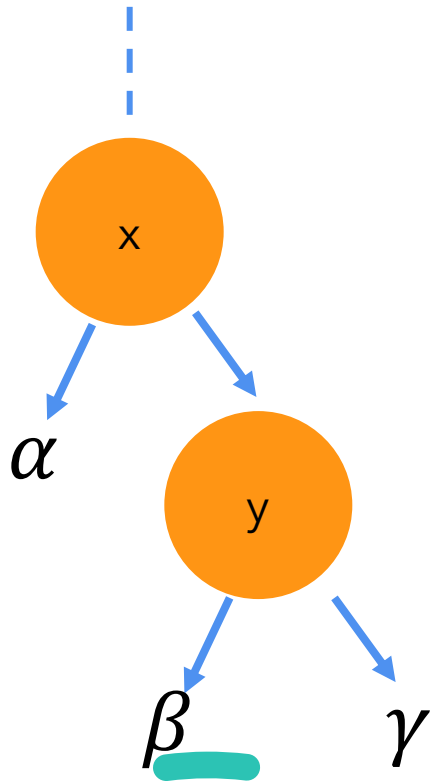
ROTACIONES




```
LEFT-ROTATION(T,x)
1.  y = T.right(x)
2.  T.insertRight(x,T.left(y))
3.  IF T.parent(x)==null
4.      root = y
5.  ELSEIF x==T.left(T.parent(x))
6.      T.insertLeft(T.parent(x),y)
7.  ELSE
8.      T.insertRight(T.parent(x),y)
9.  T.insertLeft(y,x)
```

Arboles balanceados - RBT

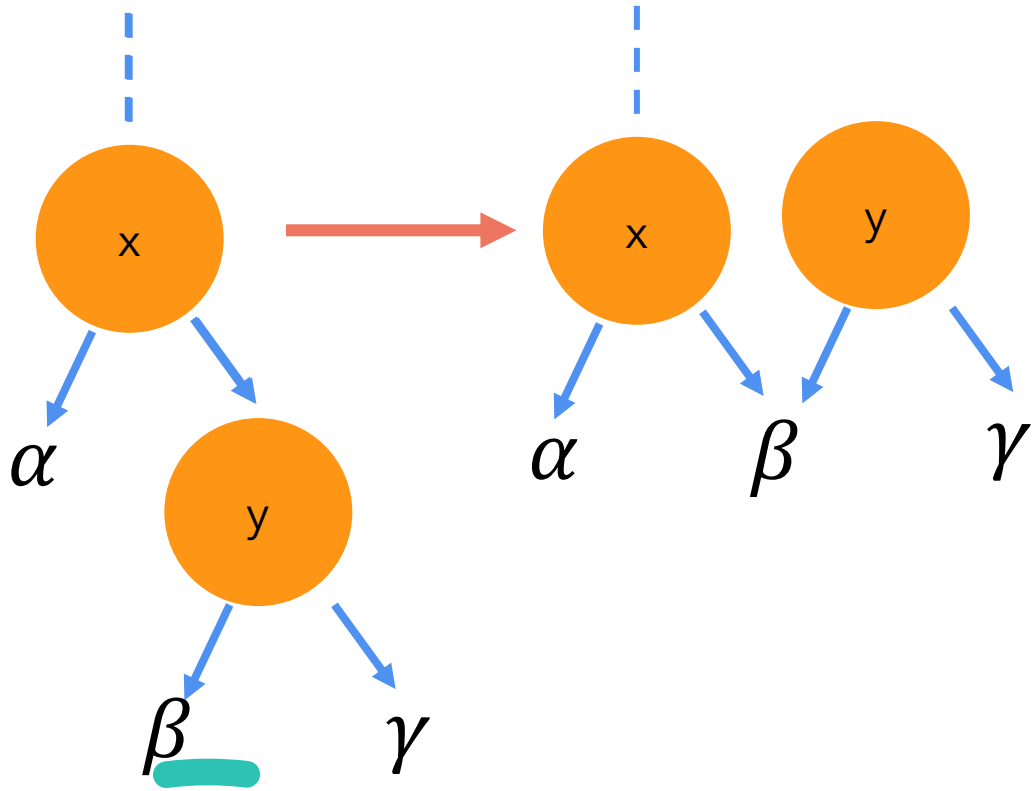
ROTACIONES




```
LEFT-ROTATION(T,x)
1.  y = T.right(x)
2.  T.insertRight(x,T.left(y))
3.  IF T.parent(x)==null
4.      root = y
5.  ELSEIF x==T.left(T.parent(x))
6.      T.insertLeft(T.parent(x),y)
7.  ELSE
8.      T.insertRight(T.parent(x),y)
9.  T.insertLeft(y,x)
```

Arboles balanceados - RBT

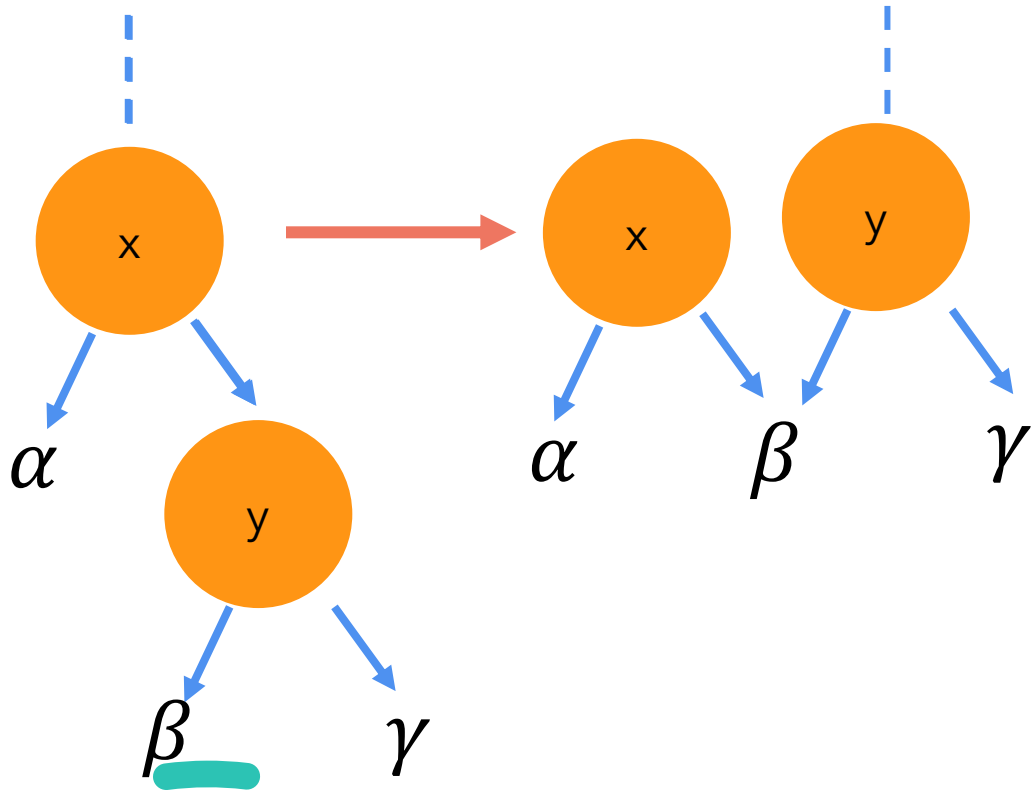
ROTACIONES




```
LEFT-ROTATION(T,x)
1.  y = T.right(x)
2.  T.insertRight(x,T.left(y))
3.  IF T.parent(x)==null
4.      root = y
5.  ELSEIF x==T.left(T.parent(x))
6.      T.insertLeft(T.parent(x),y)
7.  ELSE
8.      T.insertRight(T.parent(x),y)
9.  T.insertLeft(y,x)
```

Arboles balanceados - RBT

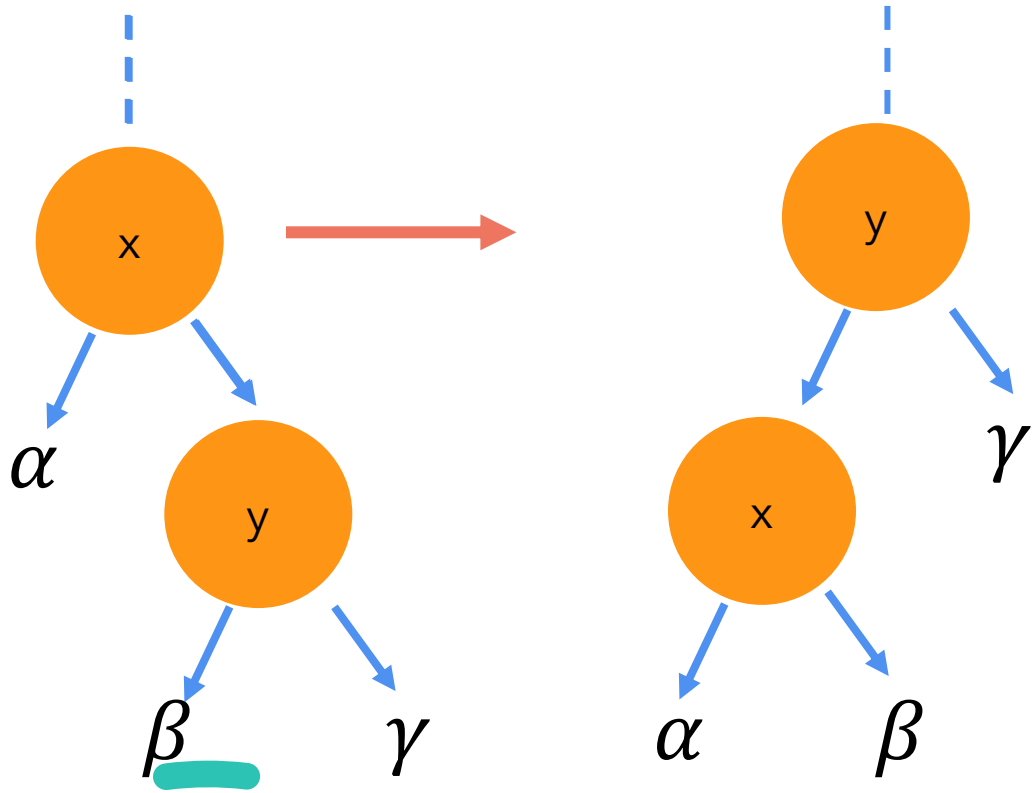
ROTACIONES




```
LEFT-ROTATION(T,x)
1.  y = T.right(x)
2.  T.insertRight(x,T.left(y))
3.  IF T.parent(x)==null
4.     root = y
5.  ELSEIF x==T.left(T.parent(x))
6.     T.insertLeft(T.parent(x),y)
7.  ELSE
8.     T.insertRight(T.parent(x),y)
9.  T.insertLeft(y,x)
```

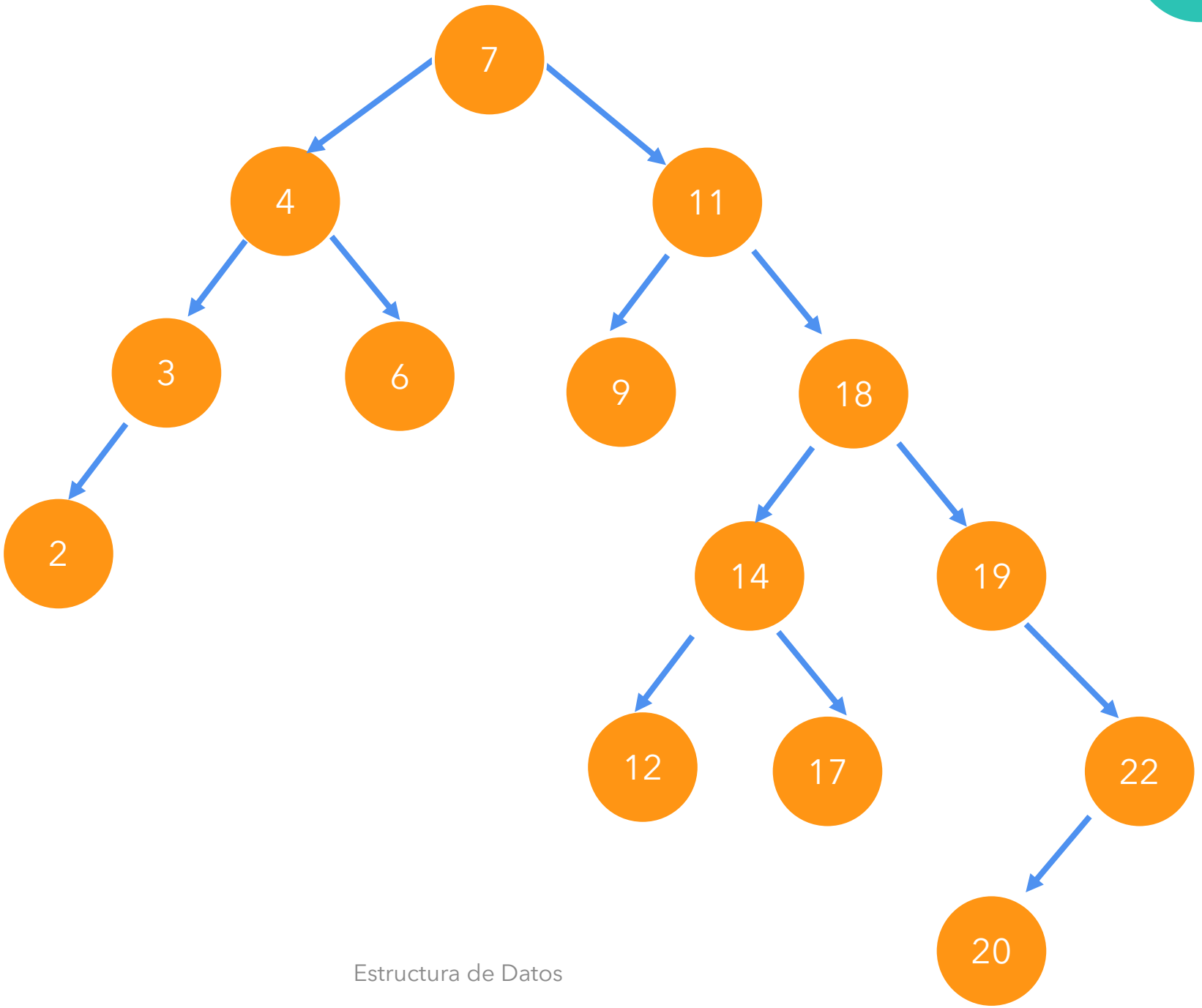
Arboles balanceados - RBT

ROTACIONES

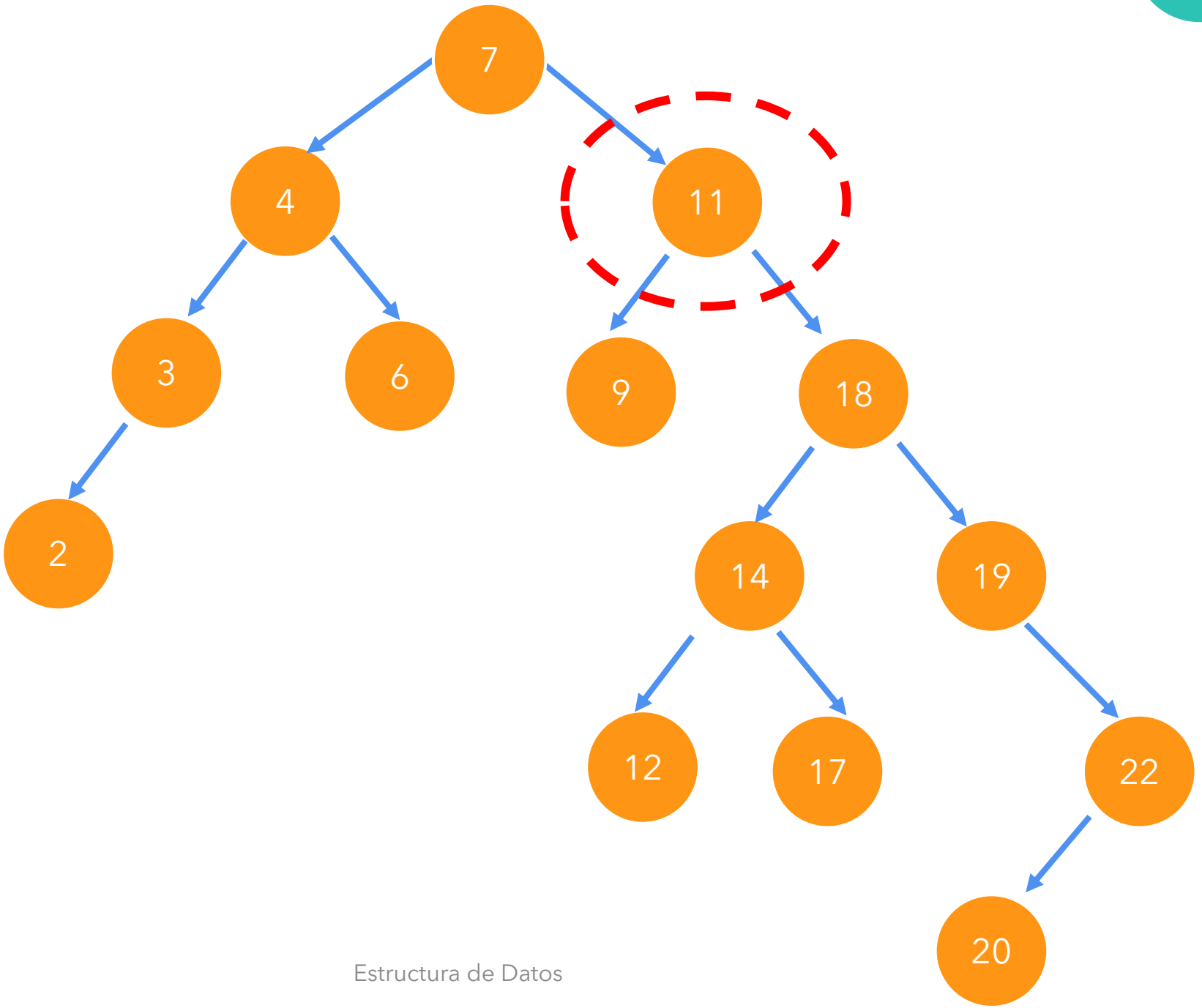



```
LEFT-ROTATION(T,x)
1.  y = T.right(x)
2.  T.insertRight(x,T.left(y))
3.  IF T.parent(x)==null
4.      root = y
5.  ELSEIF x==T.left(T.parent(x))
6.      T.insertLeft(T.parent(x),y)
7.  ELSE
8.      T.insertRight(T.parent(x),y)
9.  T.insertLeft(y,x)
```

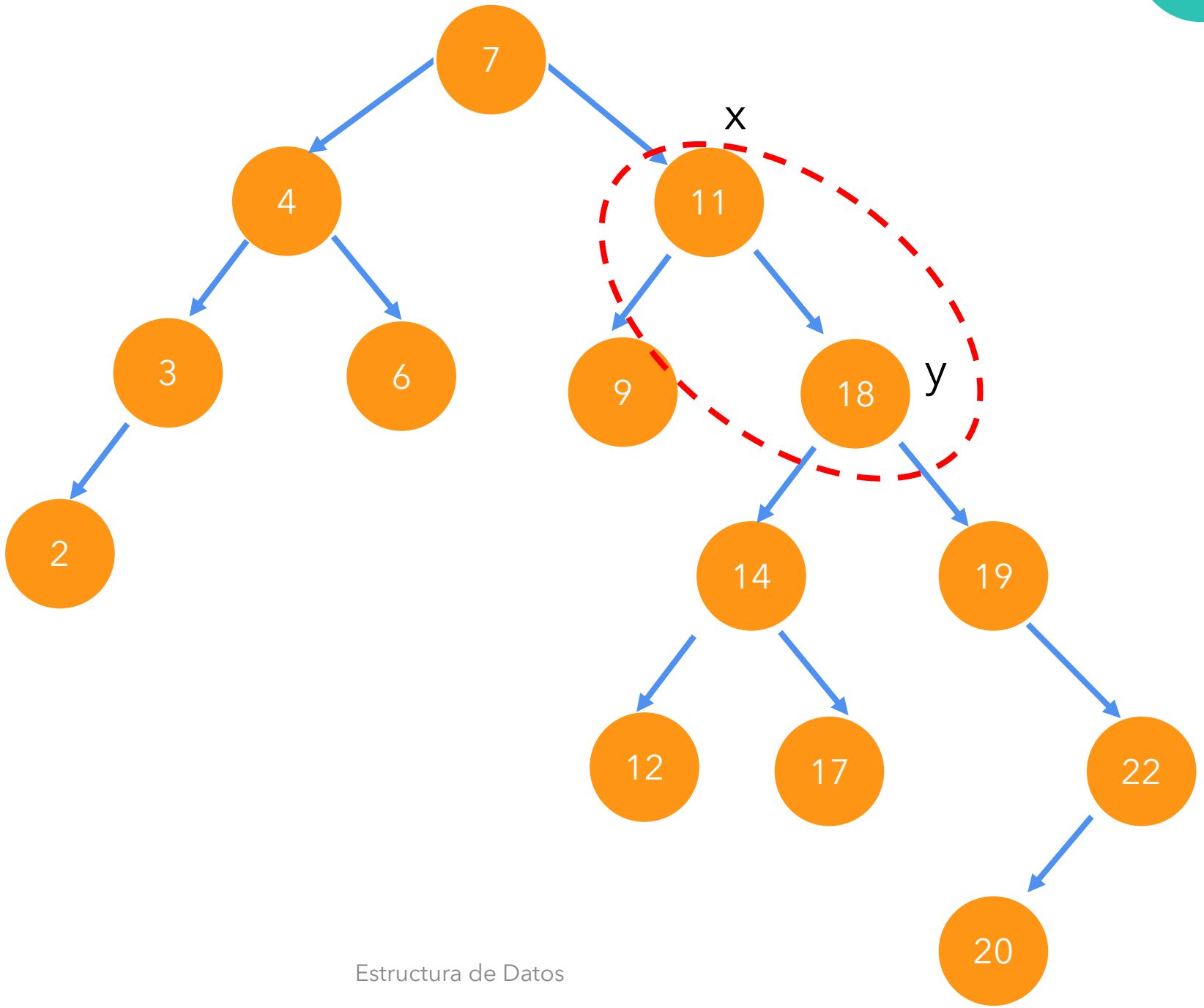
Ejemplo



Ejemplo



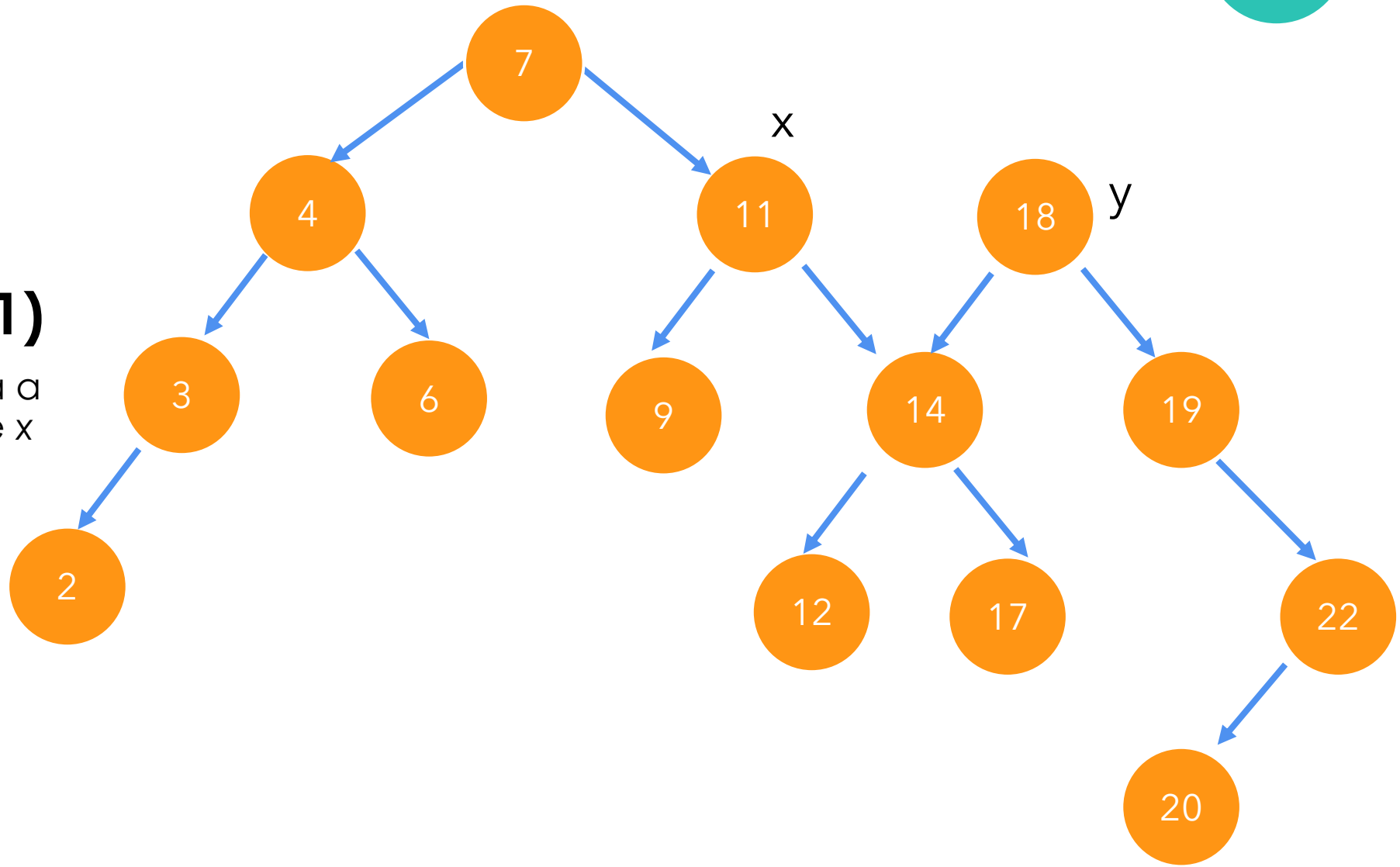
Ejemplo



Ejemplo

LEFT-ROTATION(T,11)

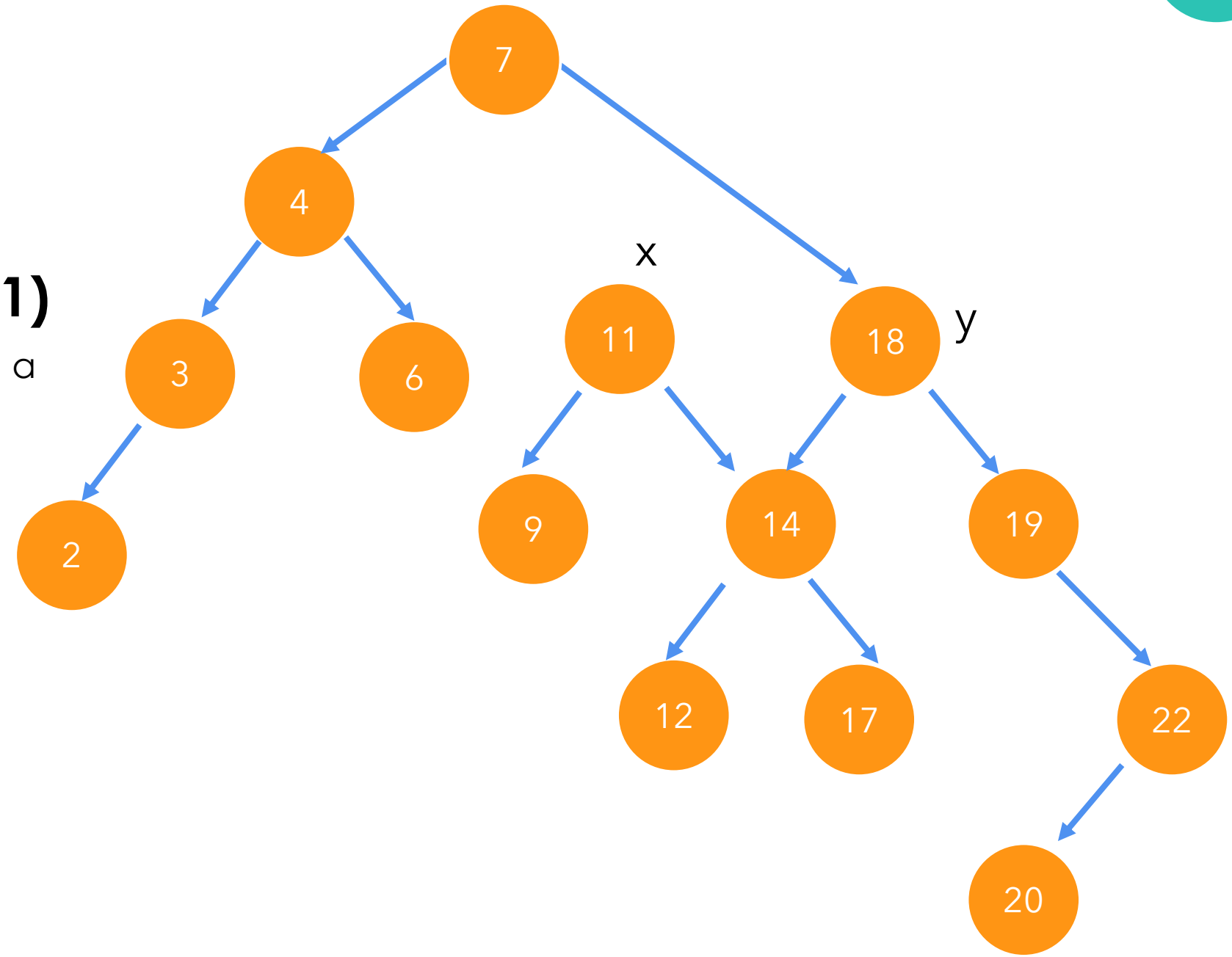
1. El hijo izq de y pasa a ser el hijo derecho de x



Ejemplo

LEFT-ROTATION(T,11)

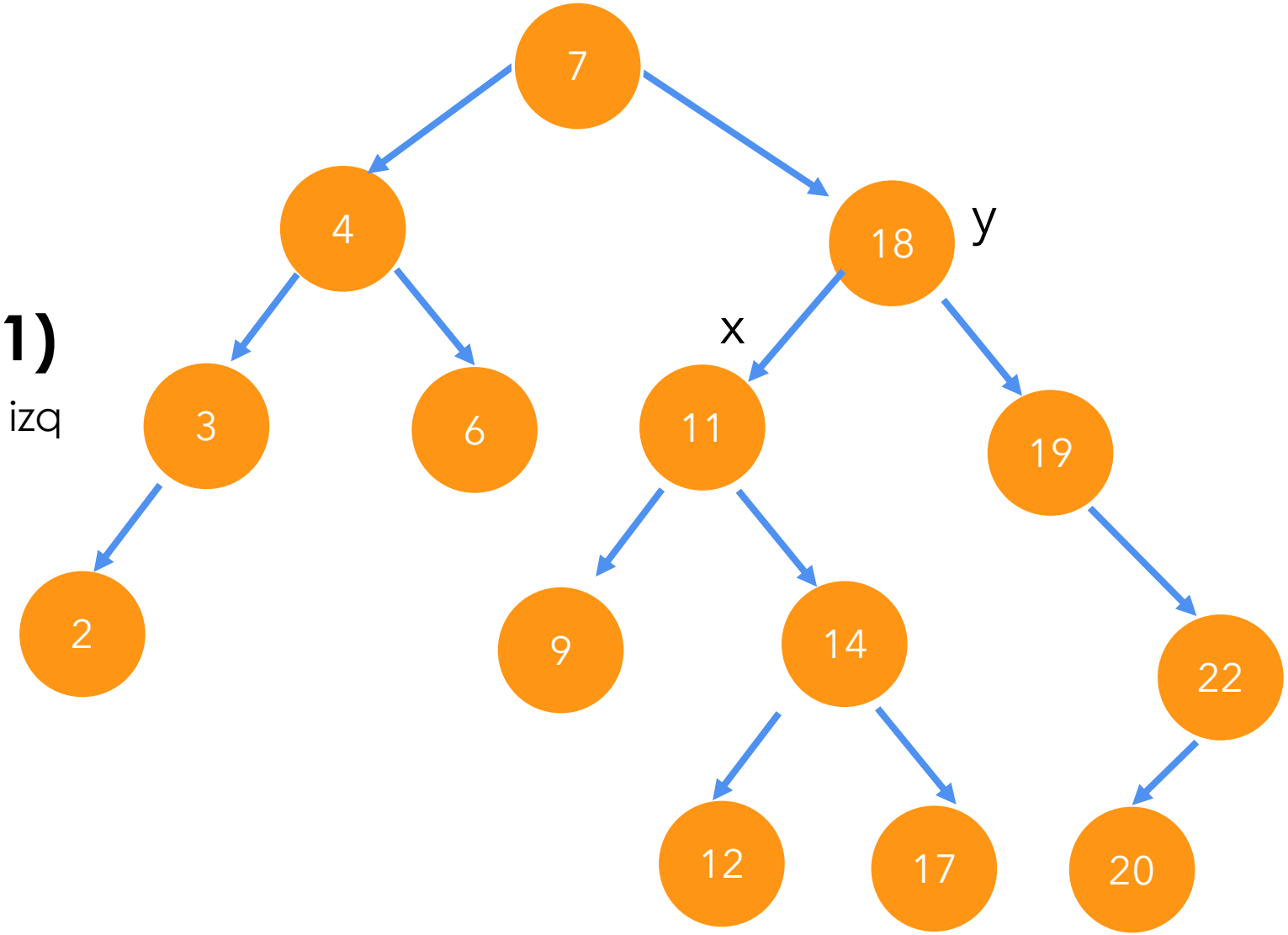
2. El padre de x pasa a ser el padre de y



Ejemplo

LEFT-ROTATION(T,11)

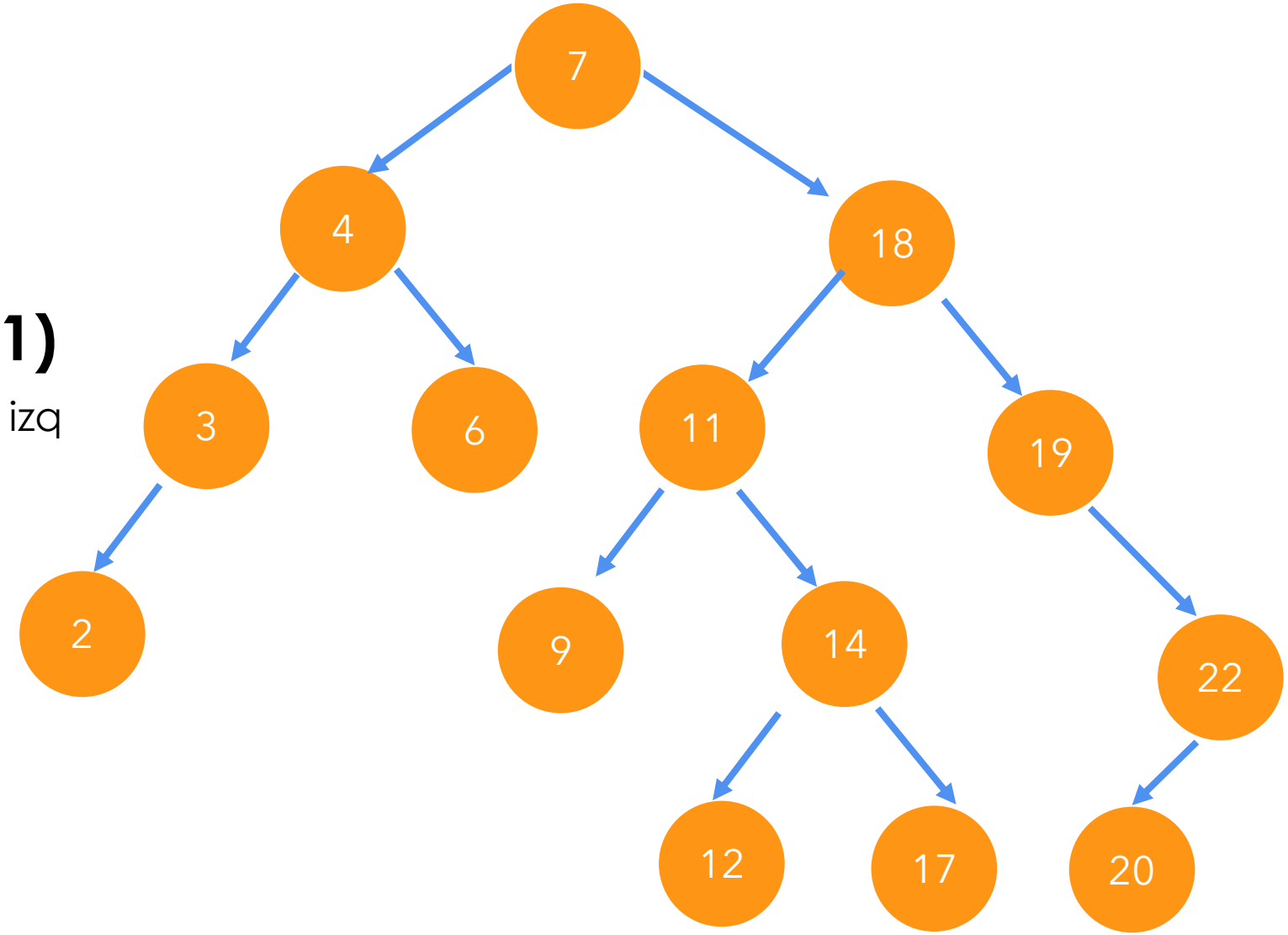
3. X pasa a ser el hijo izq de y



Ejemplo

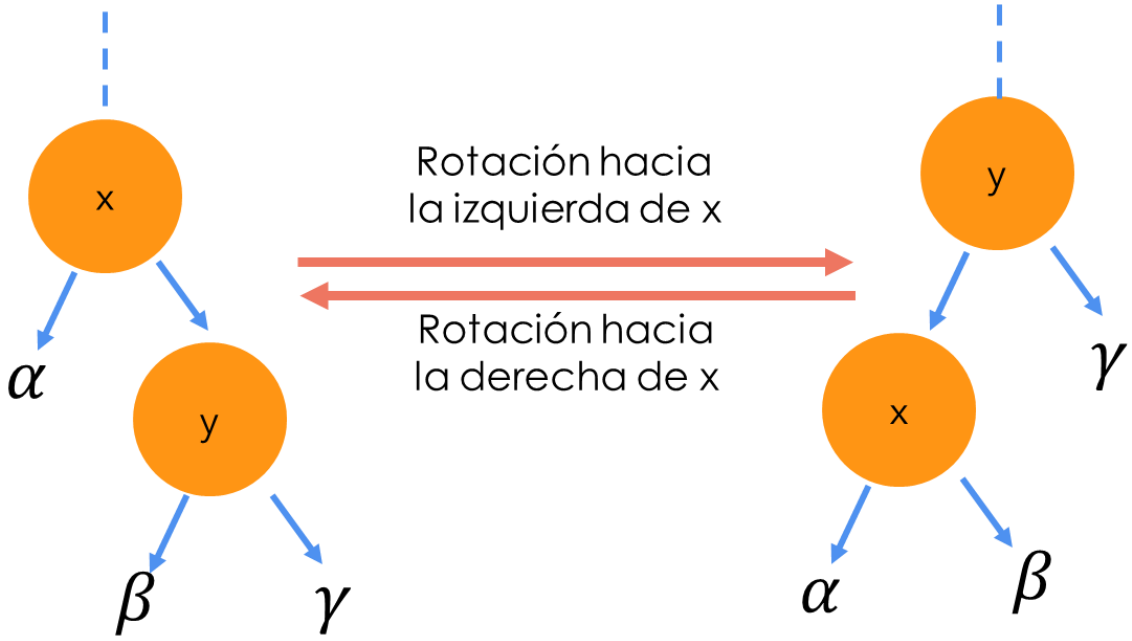
LEFT-ROTATION(T,11)

3. X pasa a ser el hijo izq de y



Arboles balanceados - RBT

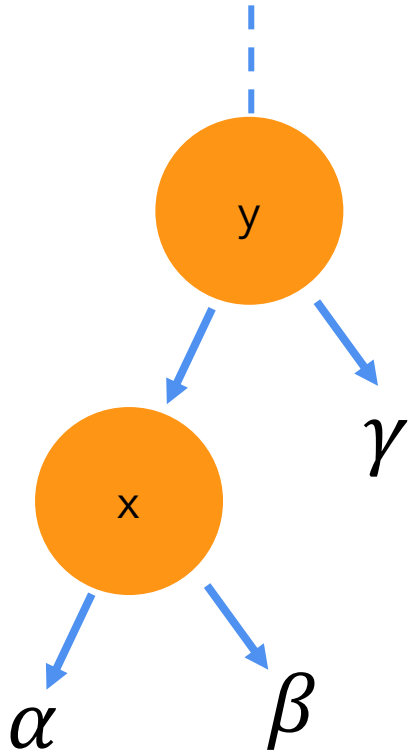
ROTACIONES




```
RIGHT-ROTATION(T,y)
1. x = T.left(y)
2. T.insertLeft(y,T.rigth(x))
3. IF T.parent(y)==null
4.     root = x
5. ELSEIF y==T.left(T.parent(y))
6.     T.insertLeft(T.parent(y),x)
7. ELSE
8.     T.insertRight(T.parent(y),x)
9. T.insertRight(x,y)
```

Arboles balanceados - RBT

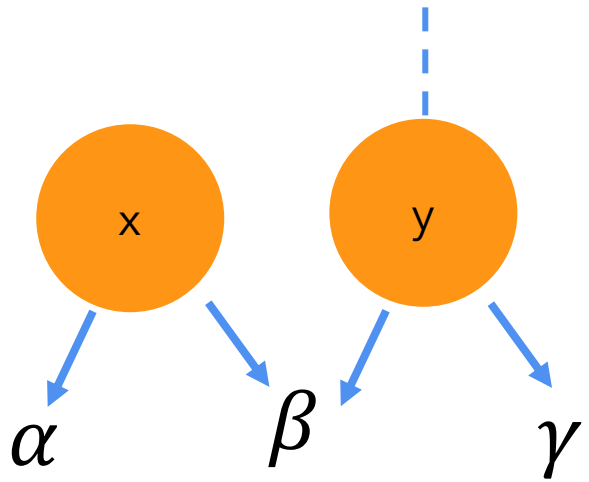
ROTACIONES




```
RIGHT-ROTATION(T,y)
1. x = T.left(y)
2. T.insertLeft(y,T.rigth(x))
3. IF T.parent(y)==null
4.     root = x
5. ELSEIF y==T.left(T.parent(y))
6.     T.insertLeft(T.parent(y),x)
7. ELSE
8.     T.insertRight(T.parent(y),x)
9. T.insertRight(x,y)
```

Arboles balanceados - RBT

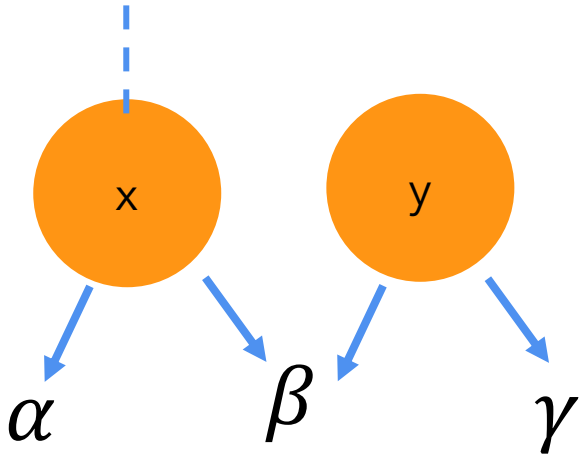
ROTACIONES




```
RIGHT-ROTATION(T,y)
1. x = T.left(y)
2. T.insertLeft(y,T.rigth(x))
3. IF T.parent(y)==null
4.     root = x
5. ELSEIF y==T.left(T.parent(y))
6.     T.insertLeft(T.parent(y),x)
7. ELSE
8.     T.insertRight(T.parent(y),x)
9. T.insertRight(x,y)
```

Arboles balanceados - RBT

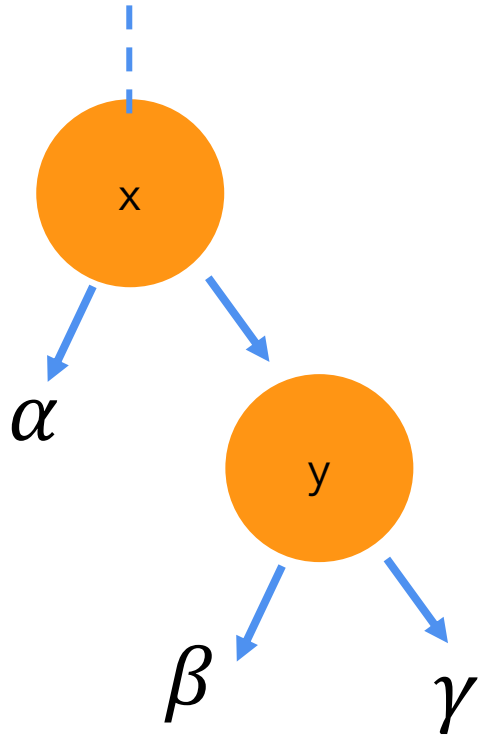
ROTACIONES




```
RIGHT-ROTATION(T,y)
1. x = T.left(y)
2. T.insertLeft(y,T.rigth(x))
3. IF T.parent(y)==null
4.     root = x
5. ELSEIF y==T.left(T.parent(y))
6.     T.insertLeft(T.parent(y),x)
7. ELSE
8.     T.insertRight(T.parent(y),x)
9. T.insertRight(x,y)
```

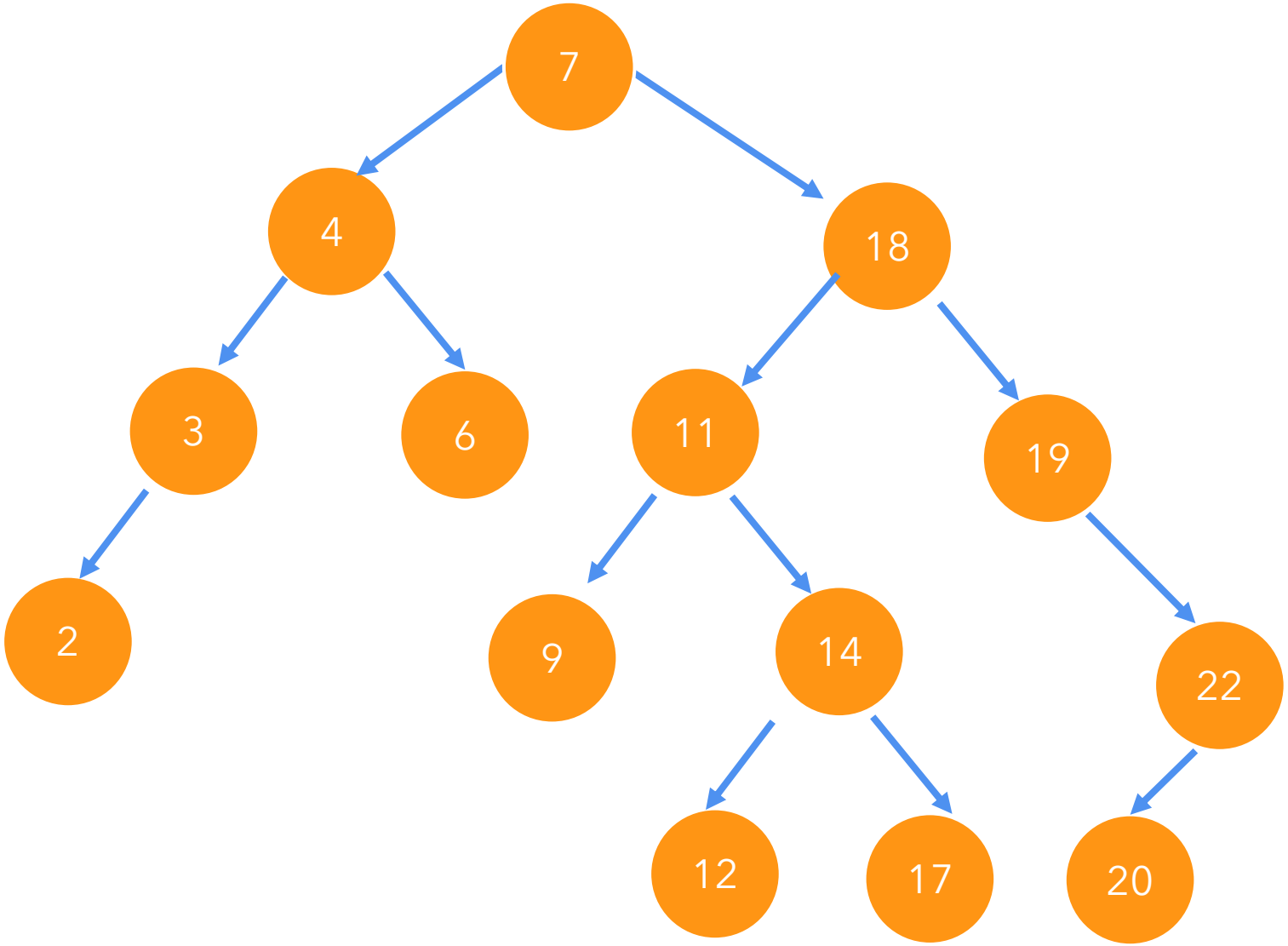
Arboles balanceados - RBT

ROTACIONES

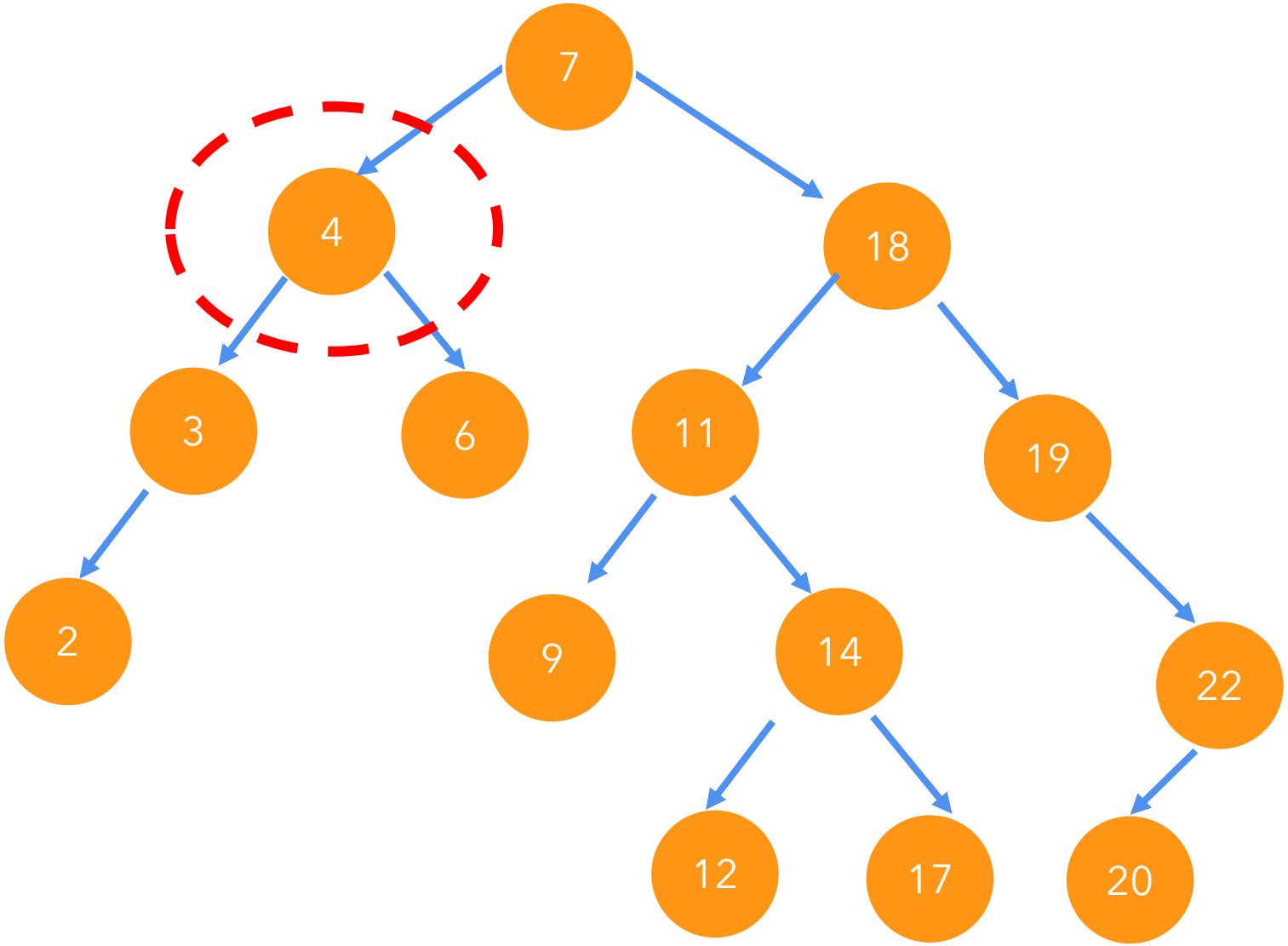



```
RIGHT-ROTATION(T,y)
1. x = T.left(y)
2. T.insertLeft(y,T.rigth(x))
3. IF T.parent(y)==null
4.     root = x
5. ELSEIF y==T.left(T.parent(y))
6.     T.insertLeft(T.parent(y),x)
7. ELSE
8.     T.insertRight(T.parent(y),x)
9. T.insertRight(x,y)
```

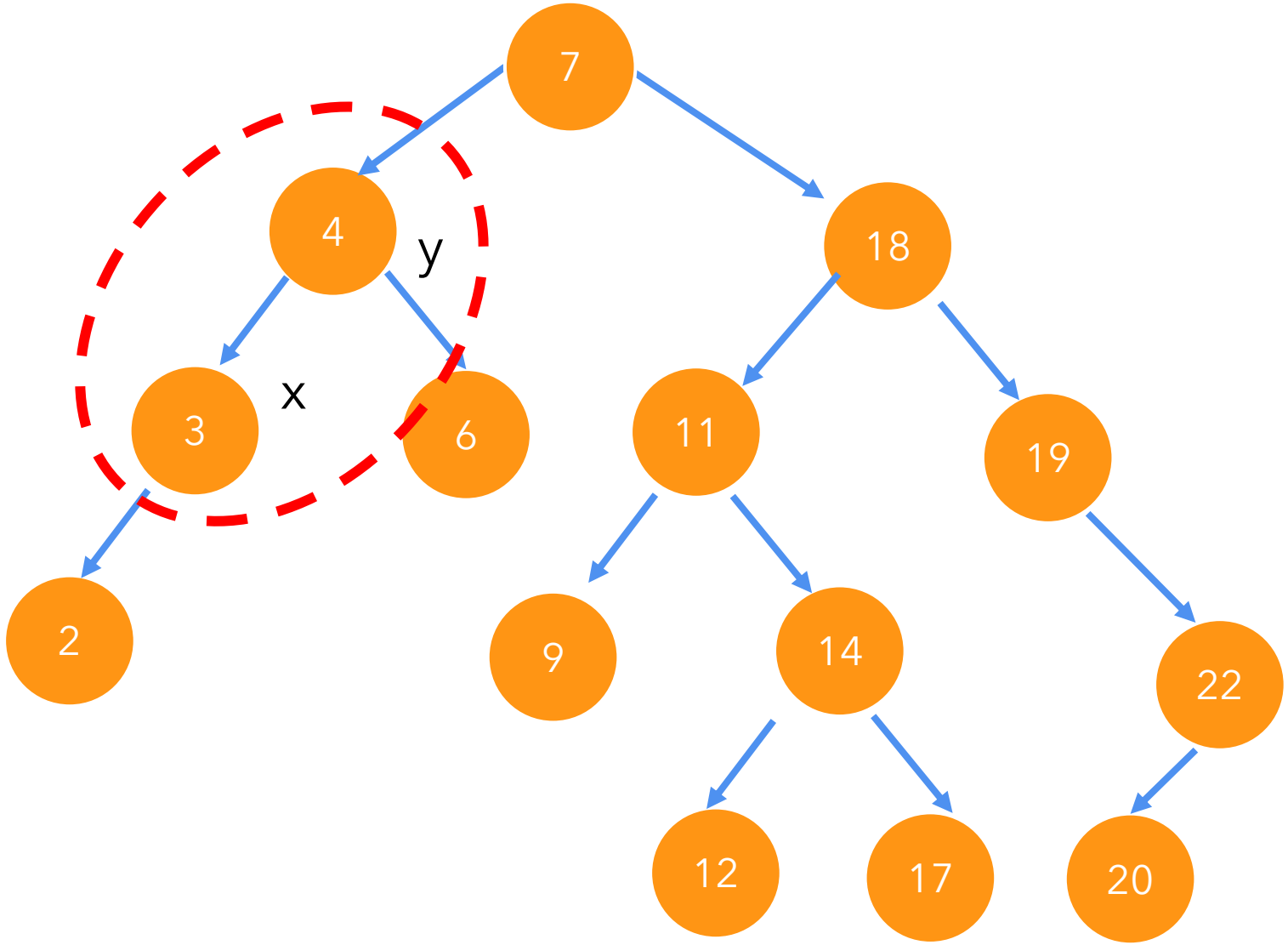
Ejemplo



Ejemplo



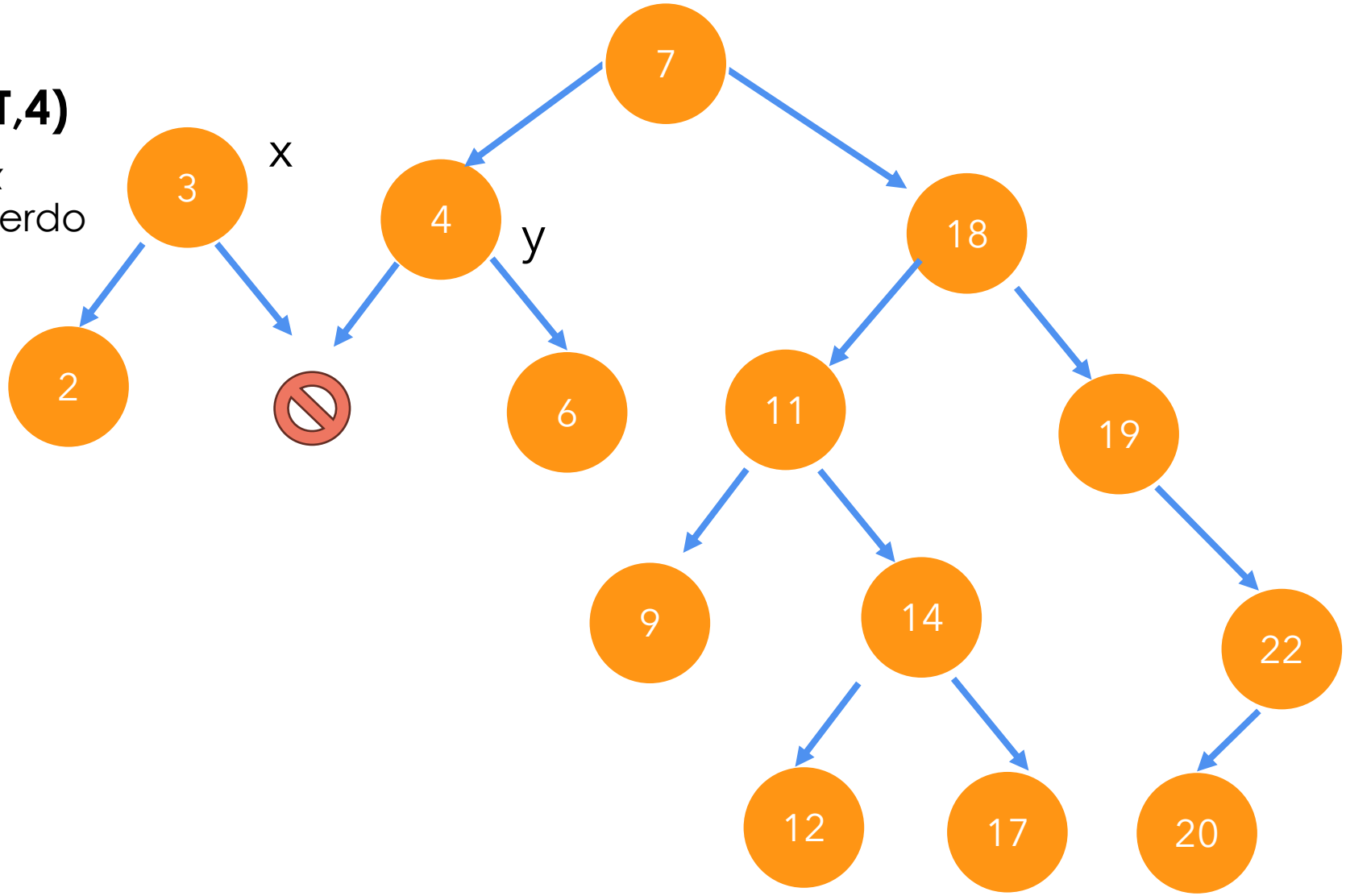
Ejemplo



Ejemplo

RIGHT-ROTATION(T,4)

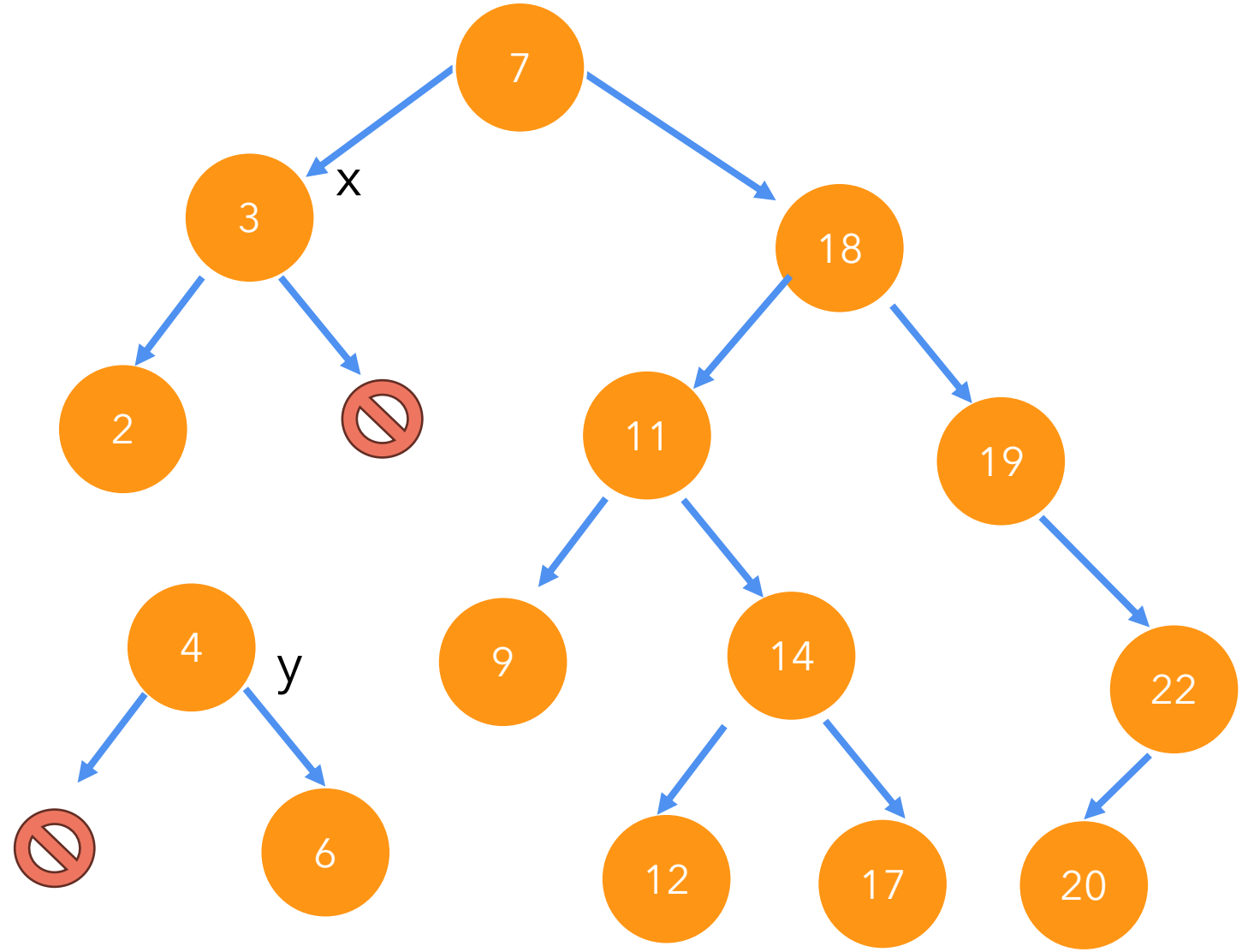
1. El hijo derecho de x pasa a ser el hijo izquierdo de x



Ejemplo

RIGHT-ROTATION(T,4)

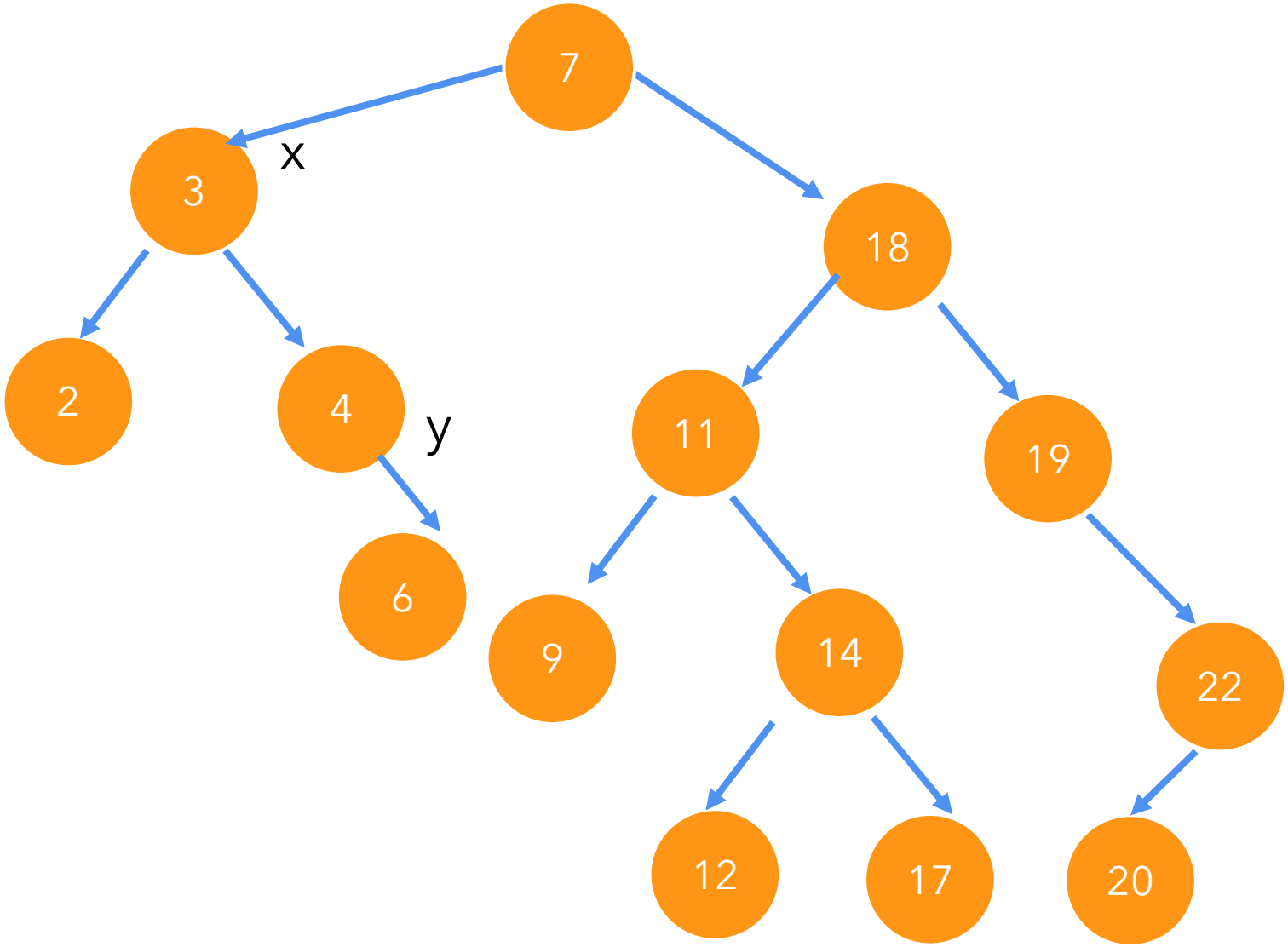
2. El padre de y pasa a ser el padre de x



Ejemplo

RIGHT-ROTATION(T,4)

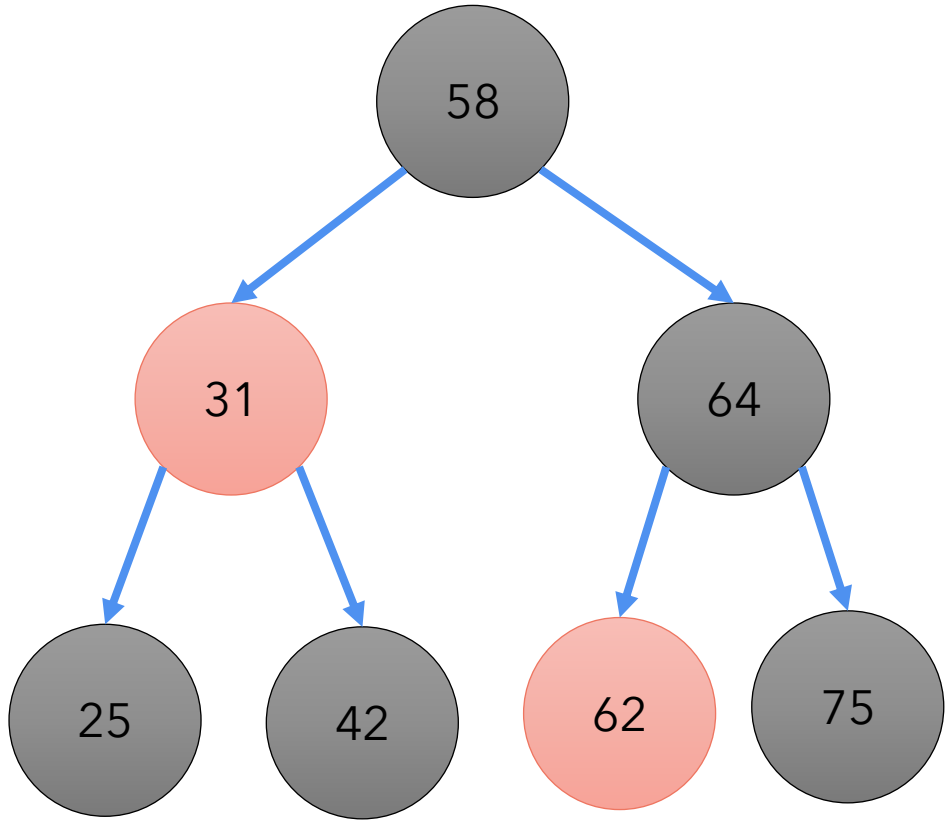
3. Y pasa a ser el hijo derecho de x



Arboles balanceados - RBT

OPERACIONES

- Rotar: mueve los nodos para mantener las propiedades del RBT
  - Rotación a la izquierda
  - Rotación a la derecha
- Insertar: agrega un nuevo elemento al RBT manteniendo las propiedades – hace uso de las rotaciones
- Eliminar: remueve un nodo del RBT manteniendo las propiedades – hace uso de las rotaciones



Arboles balanceados - RBT

INSERTAR

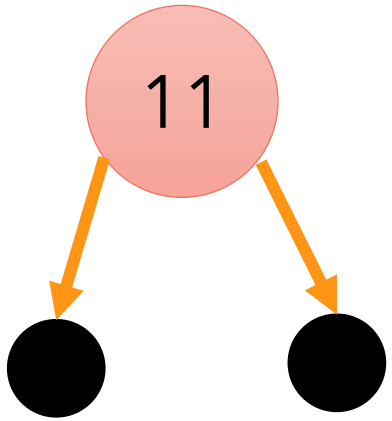
- Similar al algoritmo de insertar un nodo en un ABB
- Primero, se inserta el dato como en un árbol binario de búsqueda, y se le asigna el color rojo al nuevo nodo
- Como el nuevo nodo es una hoja se agregan los nodos centinelas: nodos sin dato, pero de color negro
- Para garantizar que las propiedades del árbol RED-BLACK se cumplen, se llama un algoritmo auxiliar RB-INSERT-FIXUP que realiza la asignación de colores y rotaciones

Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol está vacío

```
RB-Insert(RBTree T, Object o, int k)
1. RBEntry e = RBEntry(o,k)
2. IF T.isEmpty()
3.     T.addRoot(e)
4.     newNodo = T.root()
```



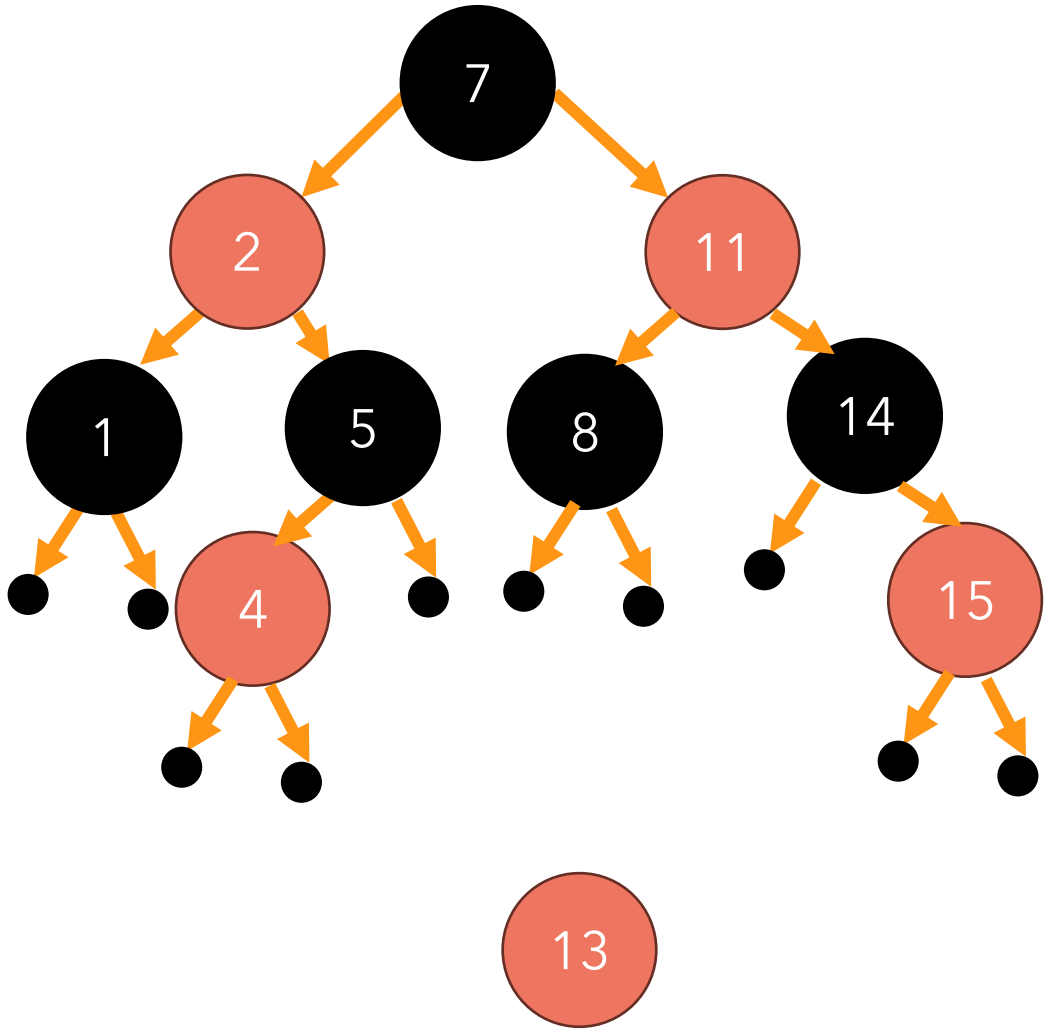
Resultado antes de aplicar el método RB-INSERT_FIXUP()

Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

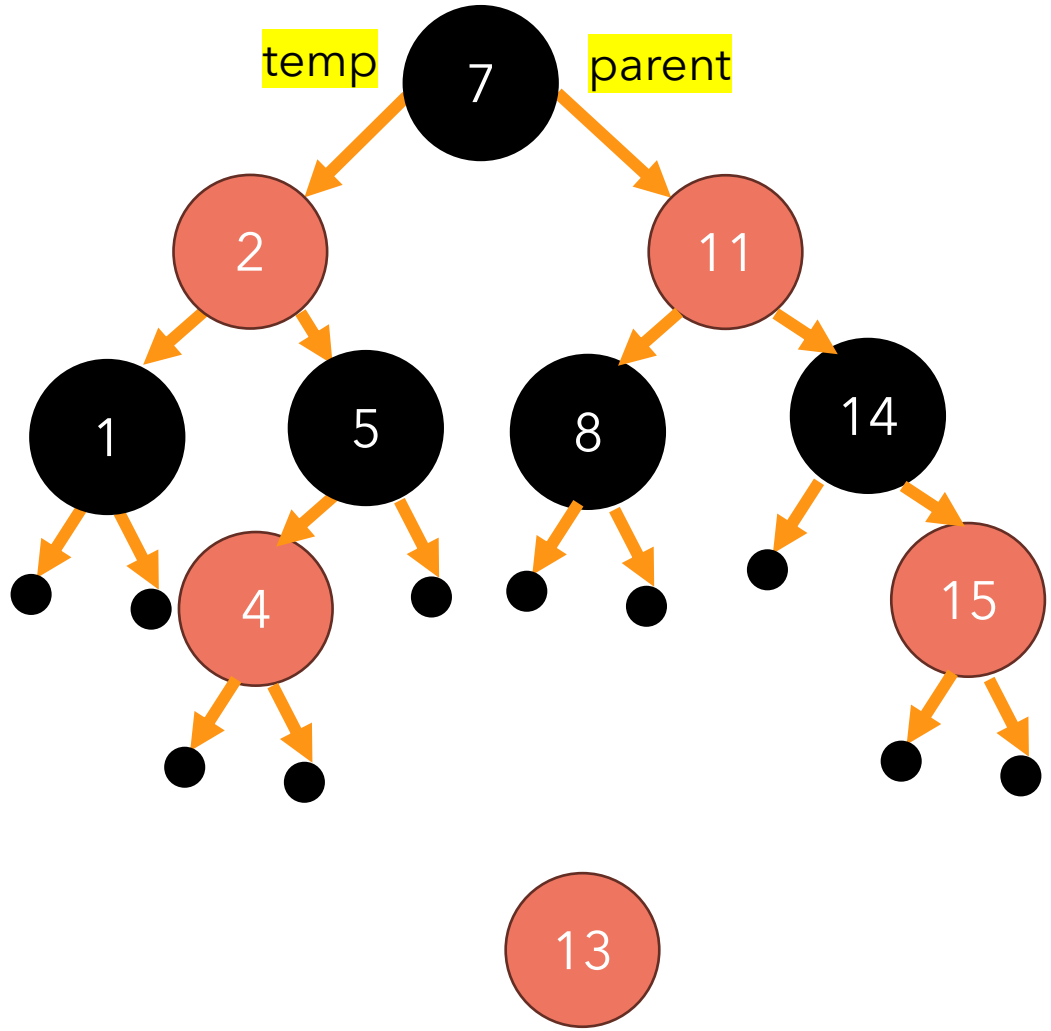


Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```



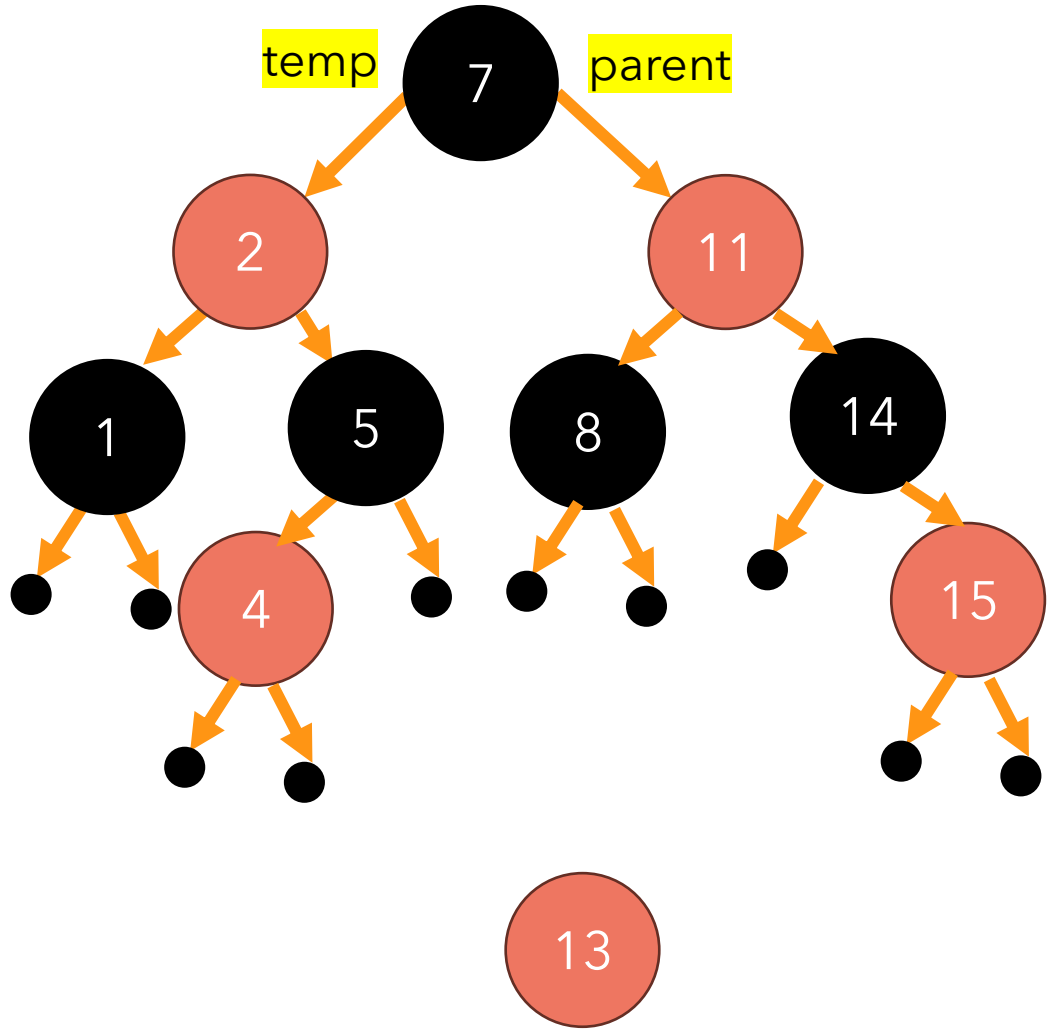
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



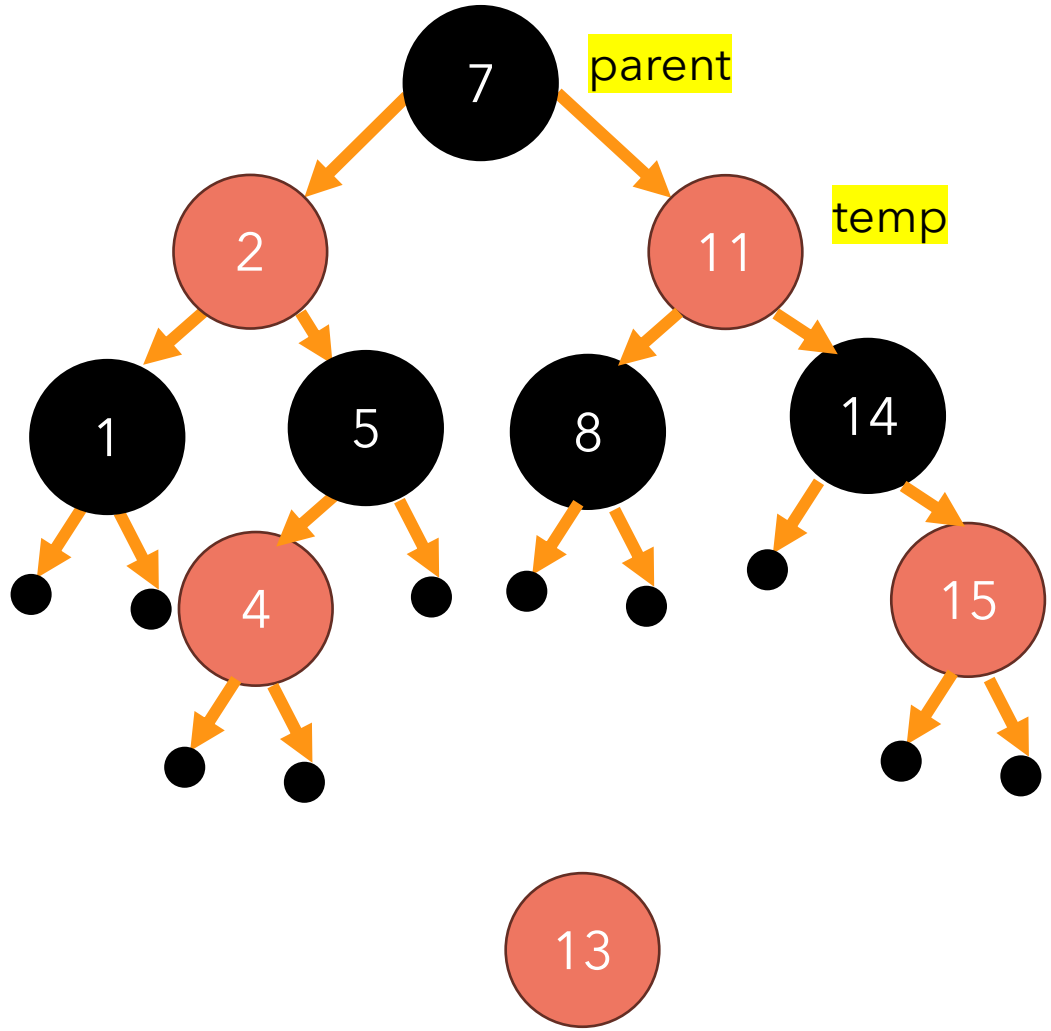
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



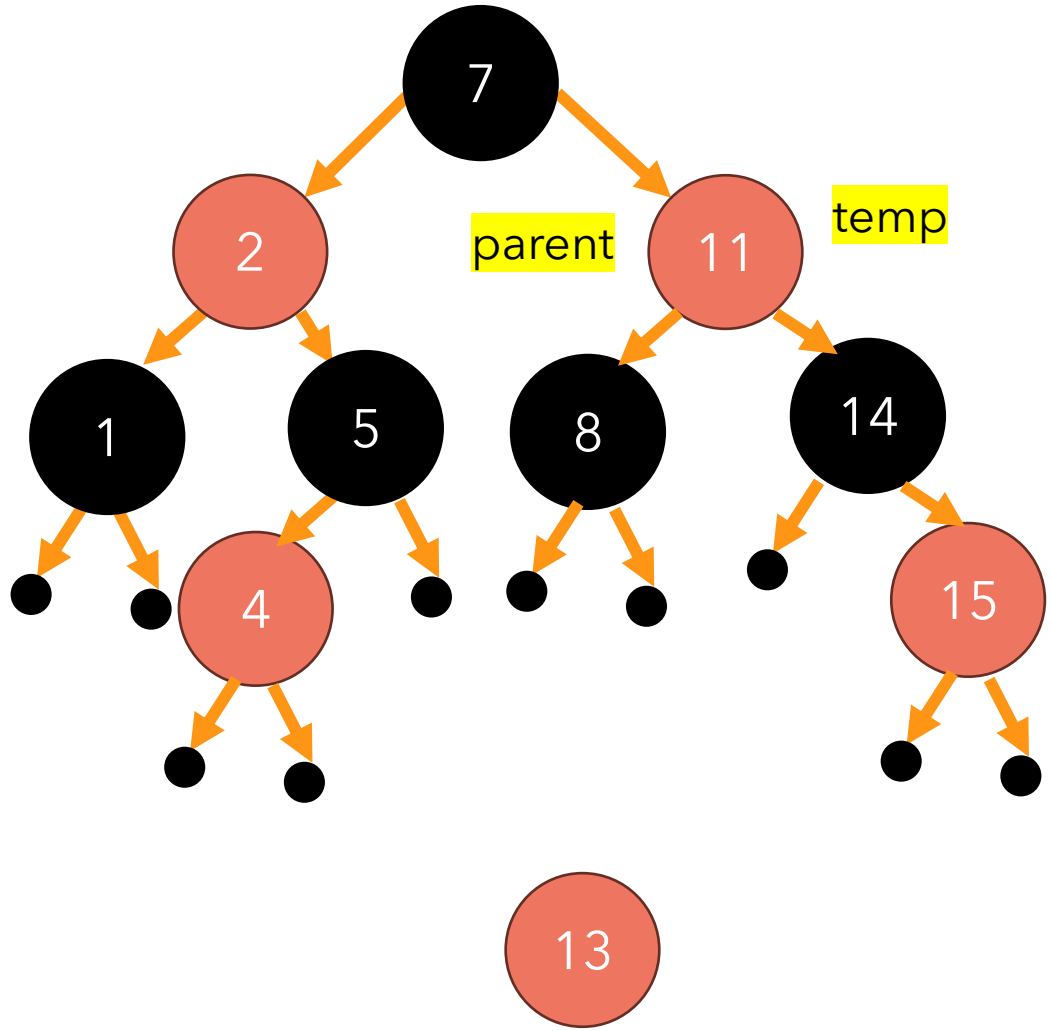
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



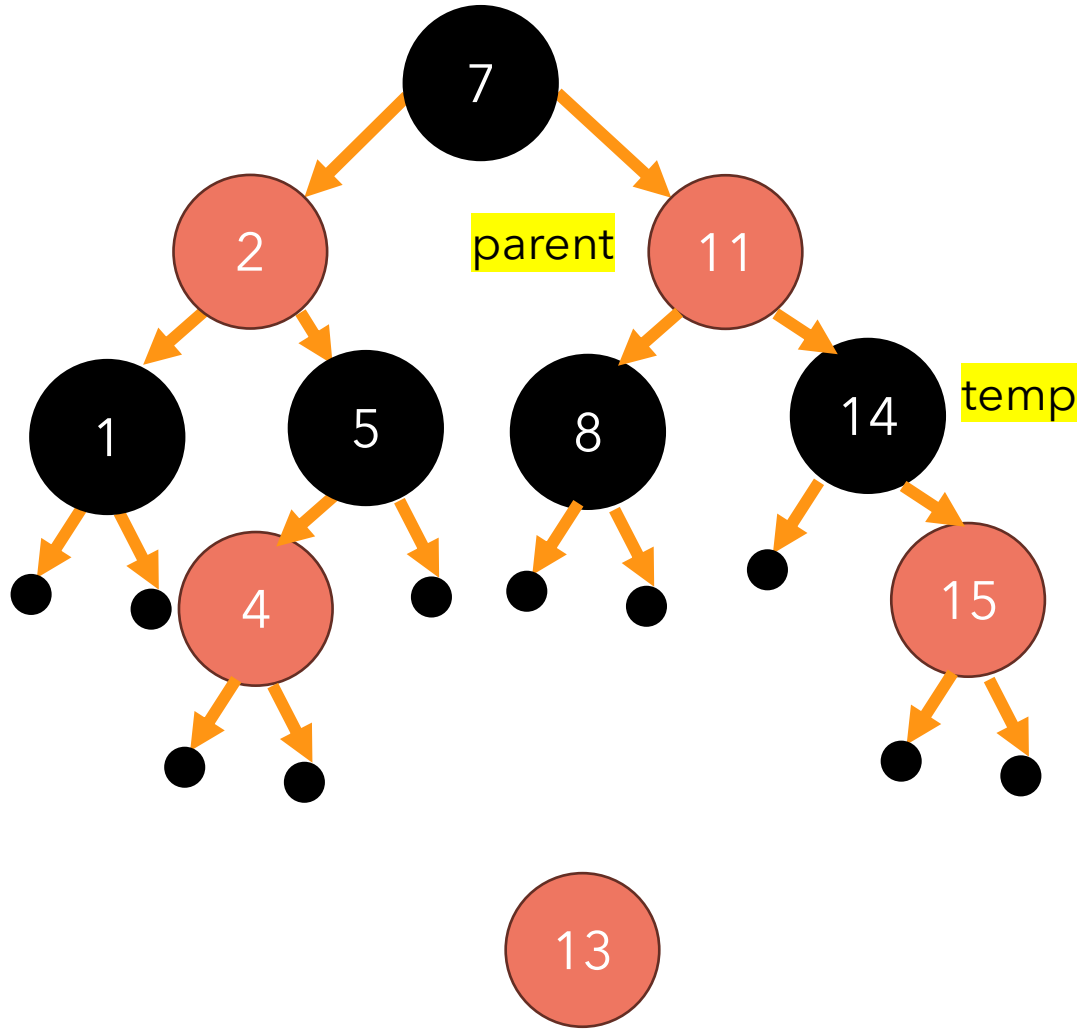
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



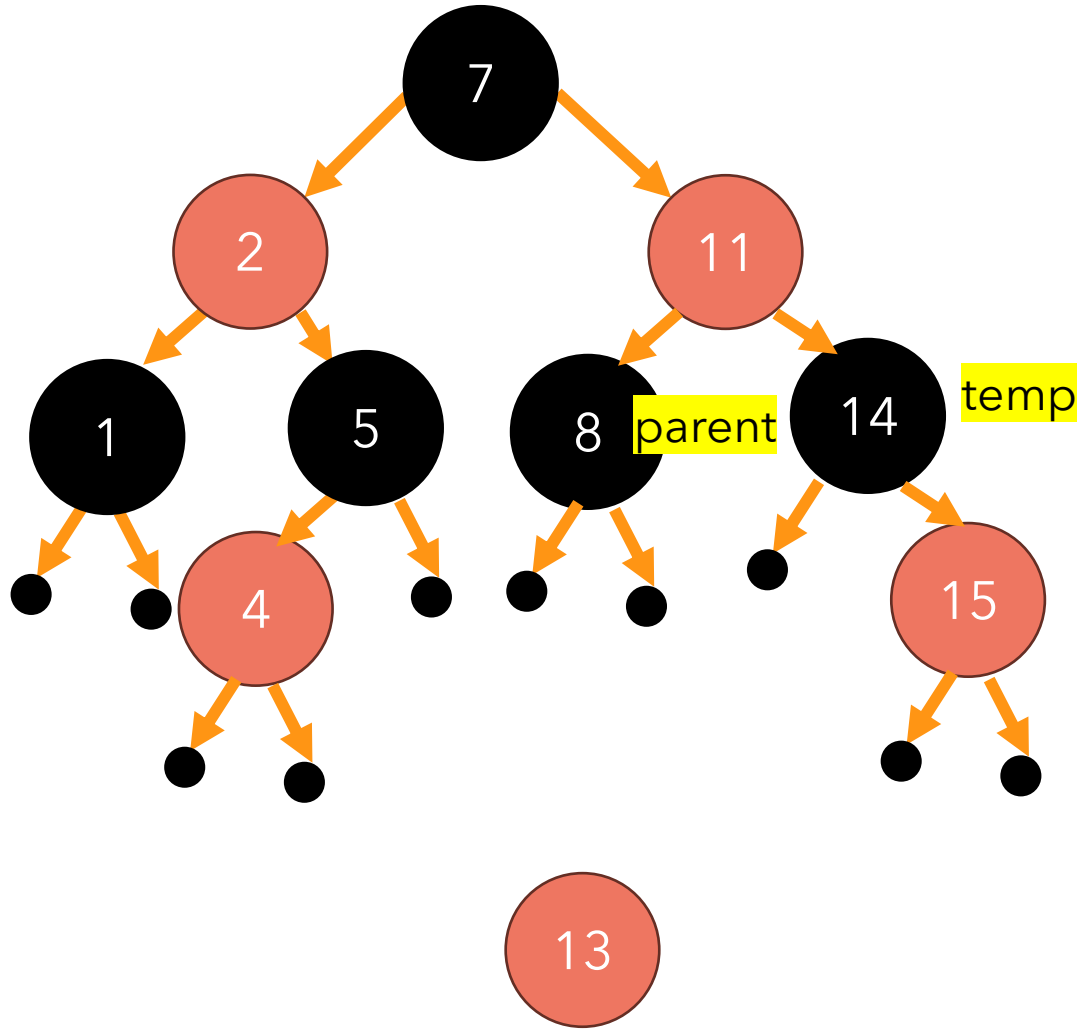
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



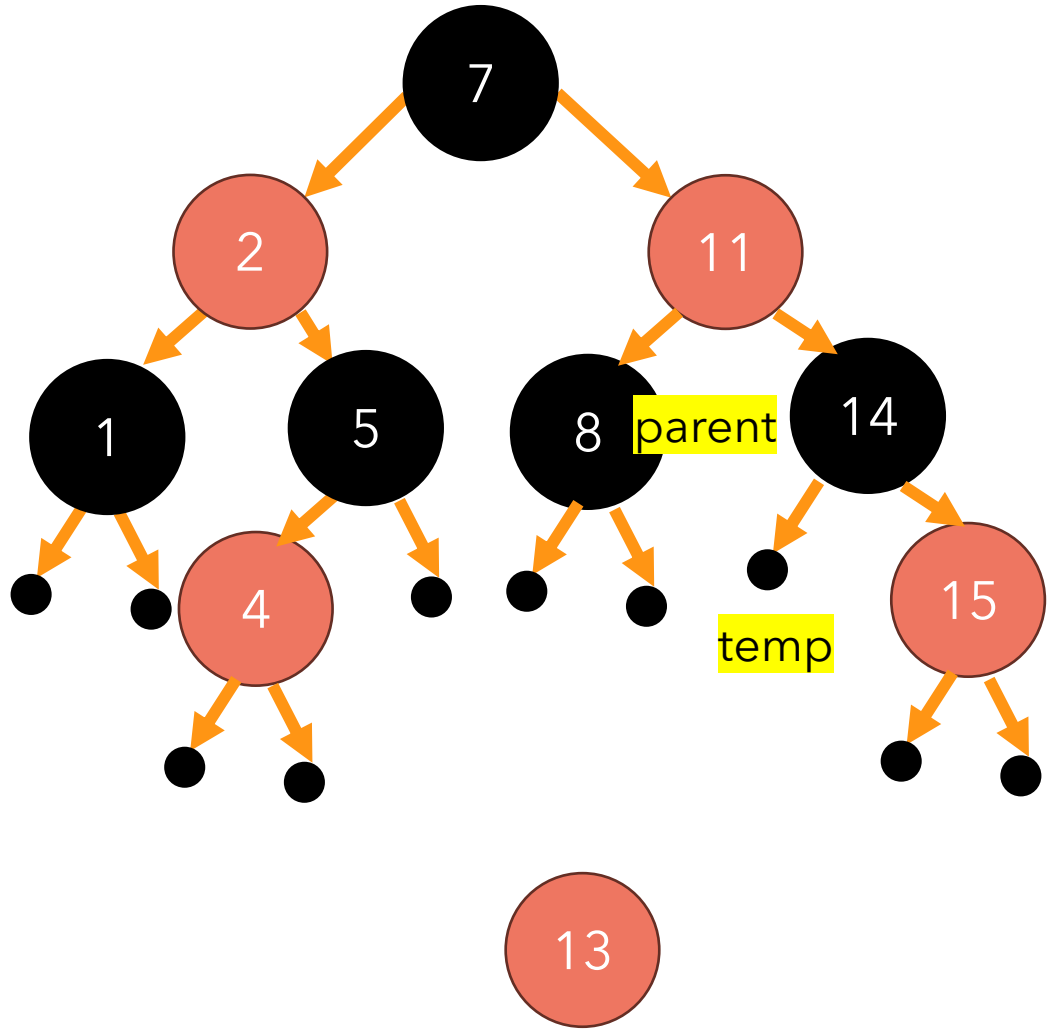
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.         temp=T.left(temp)
11.      ELSE
12.         temp = T.right(temp)
```

- Busca la posición donde se va insertar



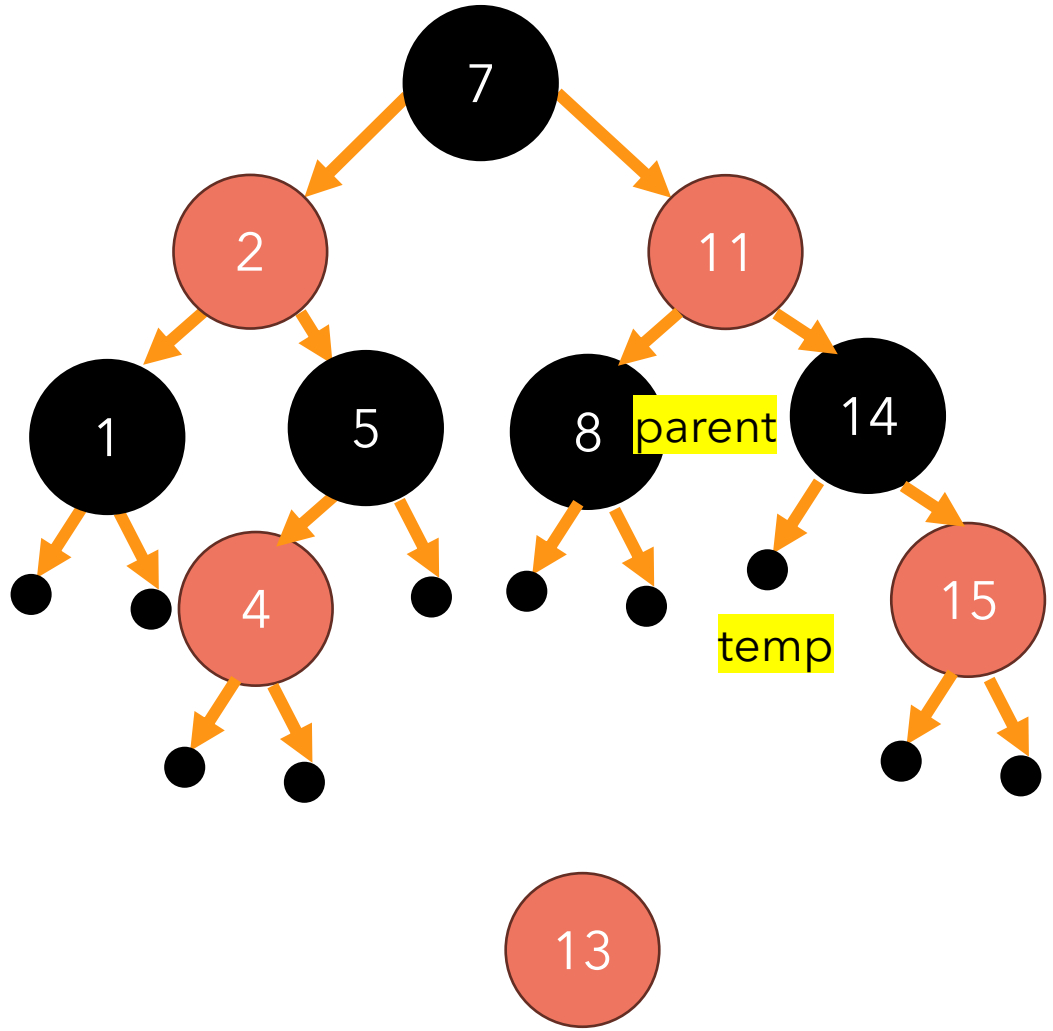
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
13.   IF k<=parent.getData().getKey()
14.      T.insertLeft(parent,e)
15.      newNodo = T.left(parent)
16.   ELSE
17.      T.insertRight(parent,e)
18.      newNodo = T.right(parent)
19.   T.size++
```

- Realiza las conexiones



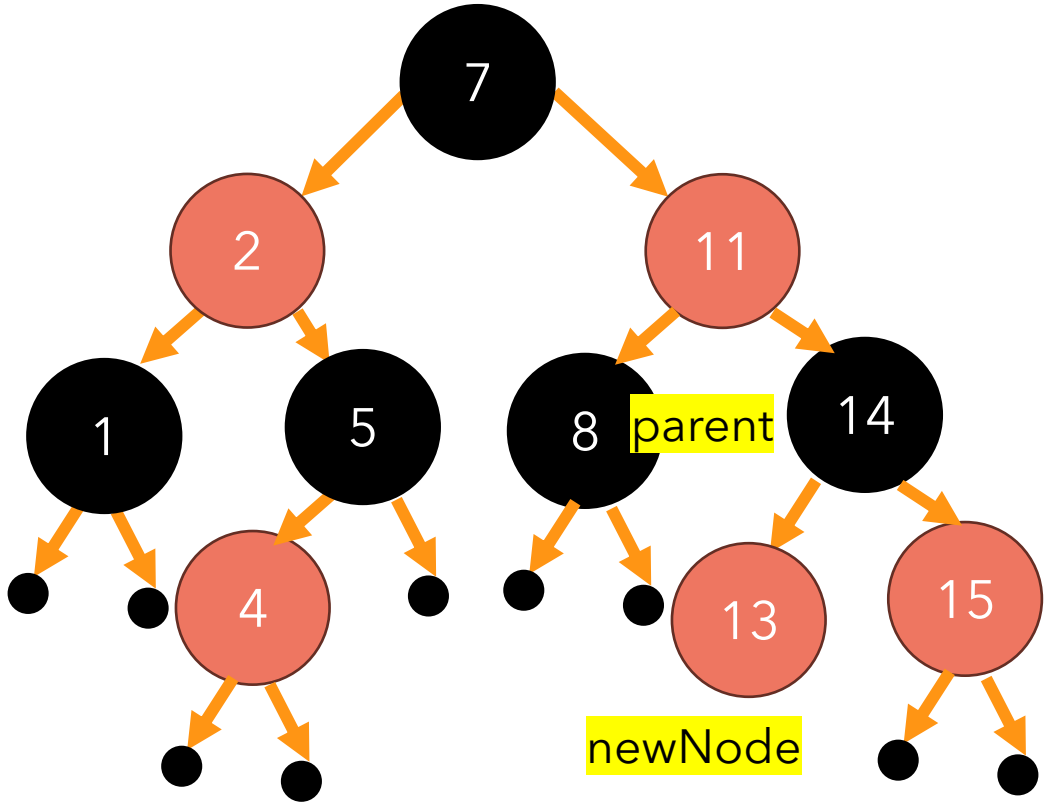
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
13.   IF k<=parent.getData().getKey()
14.      T.insertLeft(parent,e)
15.      newNodo = T.left(parent)
16.   ELSE
17.      T.insertRight(parent,e)
18.      newNodo = T.right(parent)
19.   T.size++
```

- Realiza las conexiones



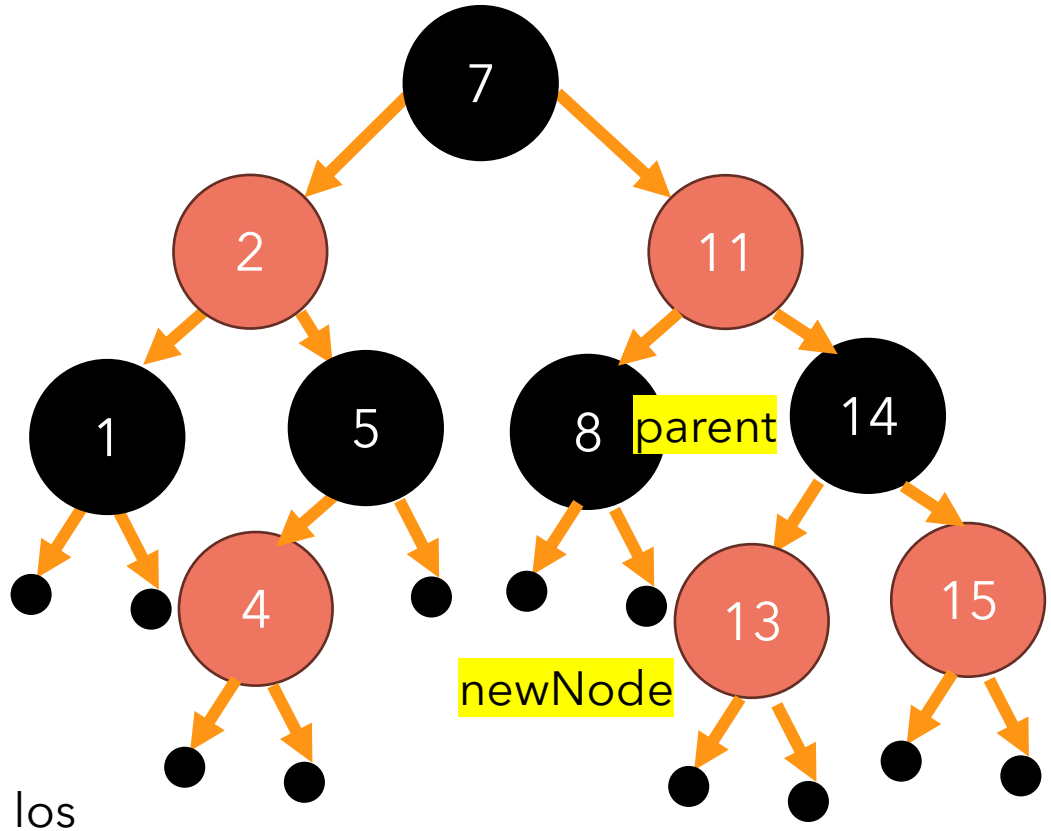
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
20. I.insertRight(newNodo, (new Node()).setColor=TRUE)
21. I.insertLeft(newNodo, (new Node()).setcolor=TRUE)
```

- Crea los nodos centinela



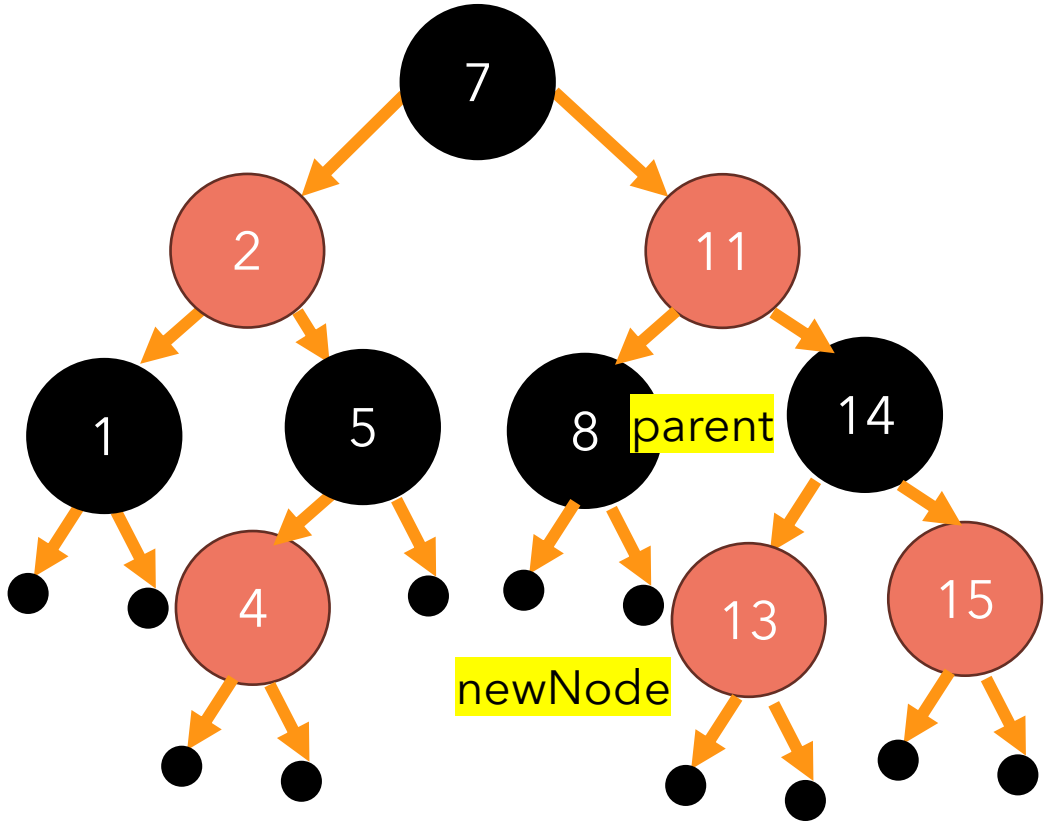
Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
22. RB-INSERT_FIXUP(T,newNodo)
```

- Realiza rotaciones y cambios de color



- Hasta este punto es el mismo algoritmo de insertar que un ABB

Arboles balanceados - RBT

INSERTAR

1. Caso: el árbol tiene datos

```
RB-Insert(RBTree T, Object o, int k)
1. RBEntry e = RBEntry(o,k)
2. IF T.isEmpty()
3.     T.addRoot(e)
4.     newNodo = T.root()
5. ELSE
6.     temp = T.root()
7.     WHILE(temp!=null)
8.       parent = temp
9.       IF k<=temp.getData().getKey()
10.        temp=T.left(temp)
11.      ELSE
12.        temp = T.right(temp)
13.    IF k<=parent.getData().getKey()
14.       T.insertLeft(parent,e)
15.       newNodo = T.left(parent)
16.    ELSE
17.       T.insertRight(parent,e)
18.       newNodo = T.right(parent)
19.    T.size++
20. I.insertRight(newNodo, (new Node()).setColor=TRUE)
21. I.insertLeft(newNodo, (new Node()).setcolor=TRUE)
22. RB-INSERT_FIXUP(T,newNodo)
```

Arboles balanceados - RBT

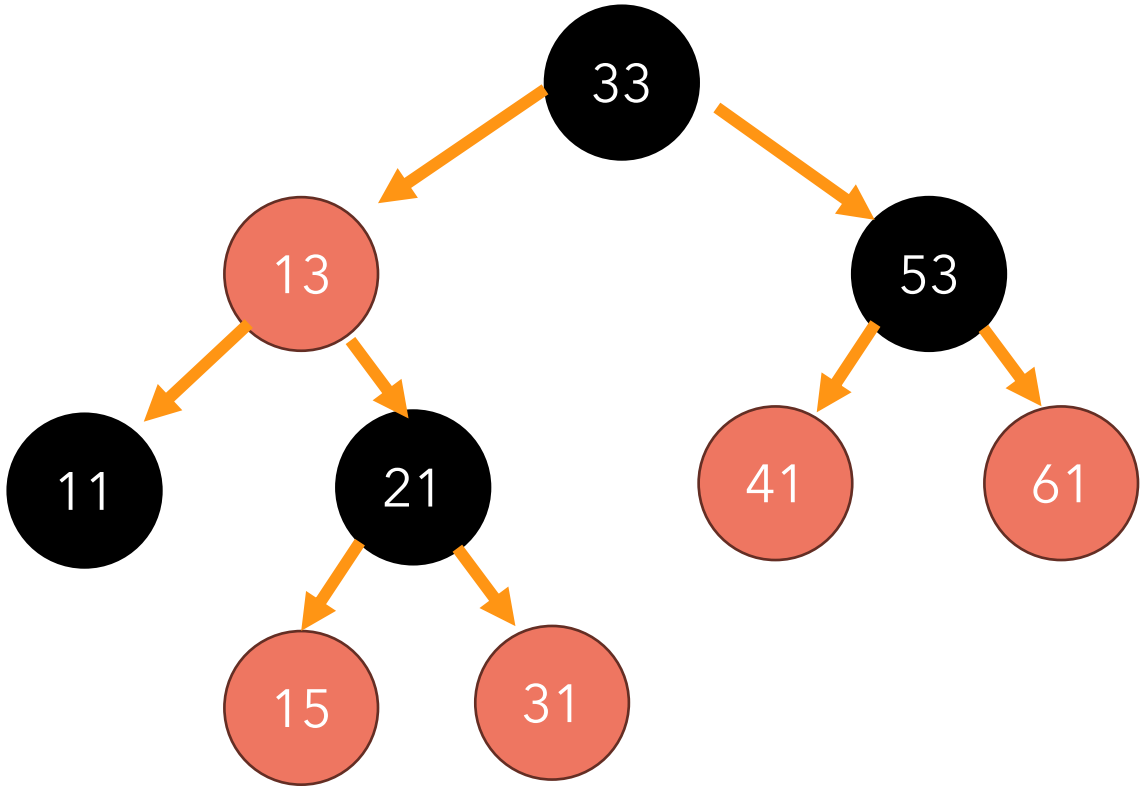
RB-INSERT_FIXUP

- Para garantizar que las propiedades del árbol RED-BLACK se cumplen, se llama un algoritmo auxiliar RB-INSERT-FIXUP que realiza la asignación de colores y rotaciones
- En el siguiente algoritmo:
  - El nodo **current** se refiere al nodo que se está analizando, inicialmente es igual a newNode
  - El nodo **p** se refiere al padre de current
  - El nodo **gp** se refiere al abuelo de current, es decir el padre del padre del current
- Vamos a estudiar el algoritmo mediante un ejemplo

Arboles balanceados - RBT

RB-INSERT_FIXUP

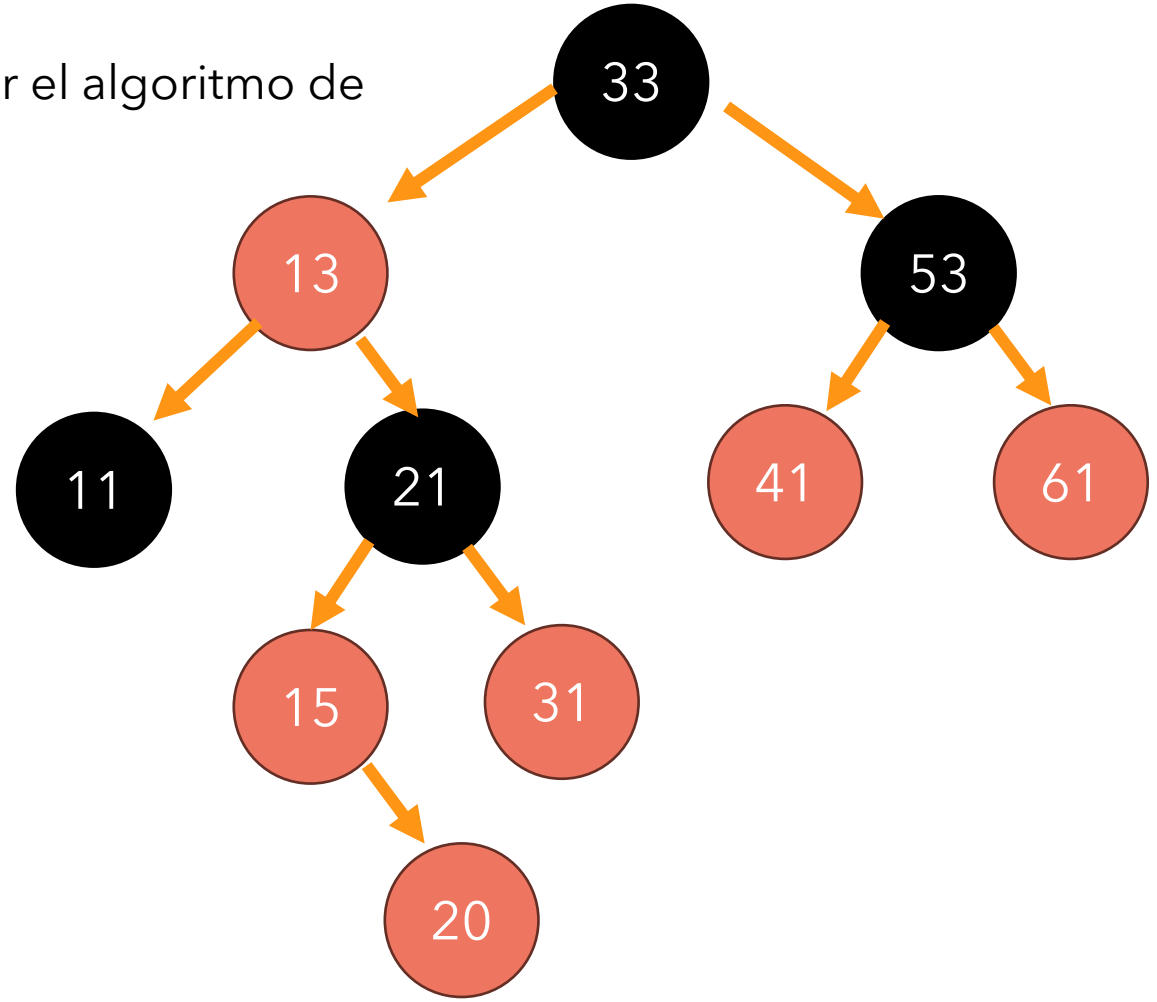
Vamos a insertar un nodo con clave = 20



Arboles balanceados - RBT

RB-INSERT_FIXUP

Resultado obtenido al aplicar el algoritmo de insertar de la línea 1-21



Arboles balanceados - RBT

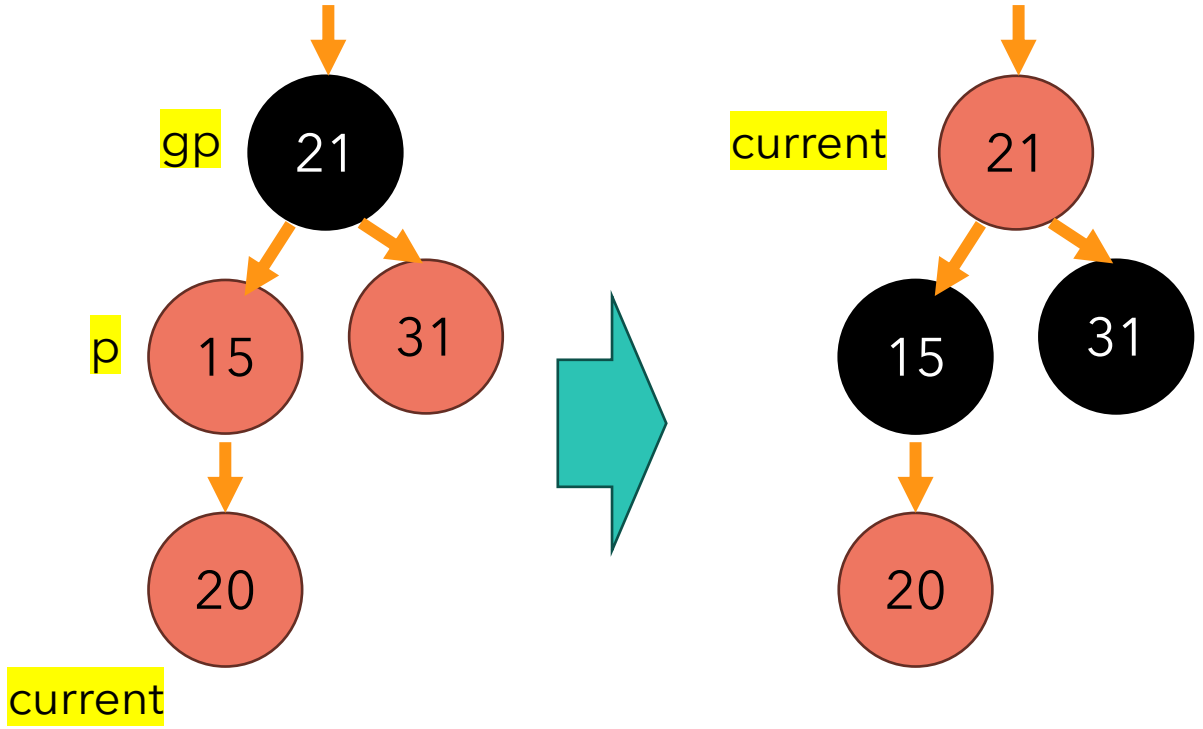
RB-INSERT_FIXUP

Para el algoritmo se corrige las propiedades del RBT:

- Repetir mientras p es RED
  - ❑ Si p es el hijo izquierdo de gp:

    CASO 1:

    1. Si el hijo derecho de gp es RED, configure los colores de ambos hijos de gp como BLACK y el color de gp como RED
    2. Asigne gp como el nuevo current



Arboles balanceados - RBT

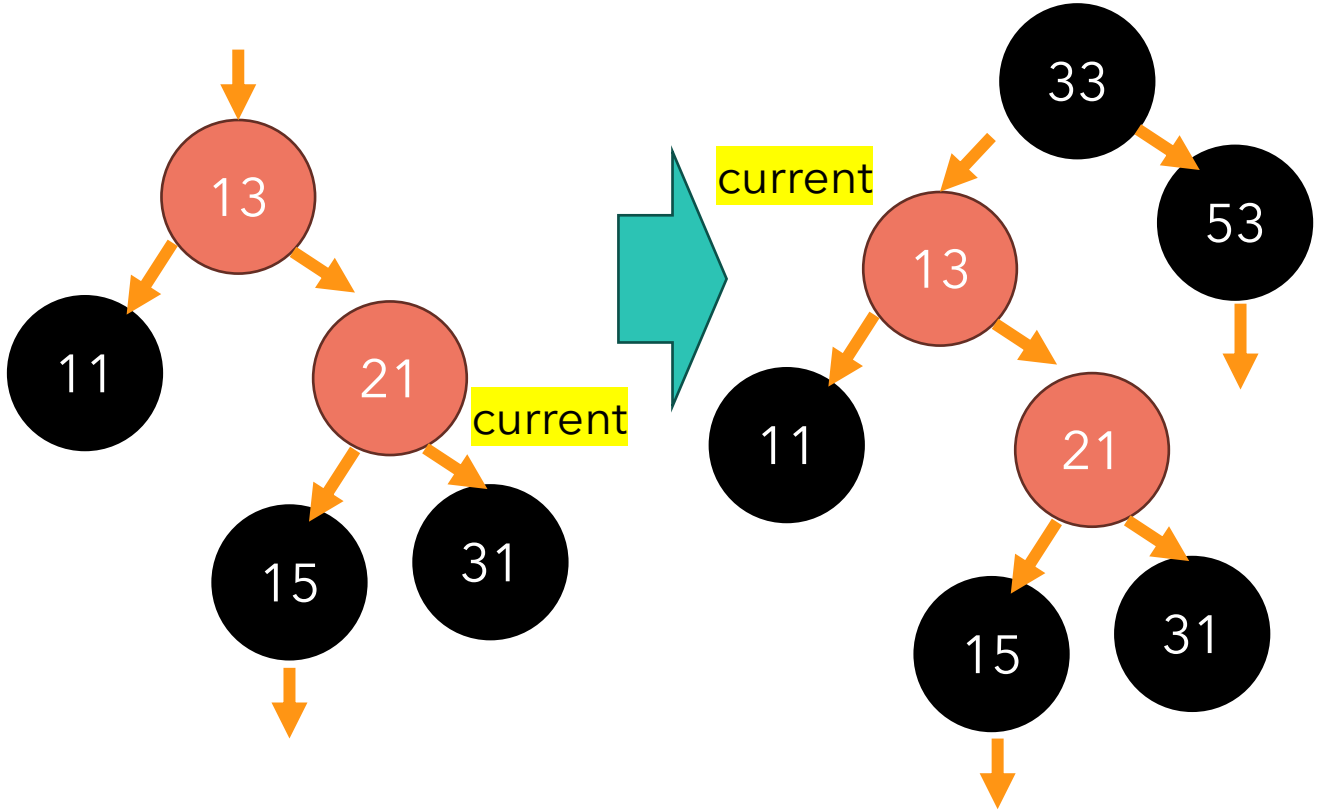
RB-INSERT_FIXUP

Para el algoritmo se corrige las propiedades del RBT:

- Repetir mientras p es RED
  - ❑ Si p es el hijo izquierdo de gp:

    CASO 2:

    1. En caso contrario, si current es el hijo derecho de p, asigne p como current
    2. Realice una rotación a la izquierda en current



Arboles balanceados - RBT

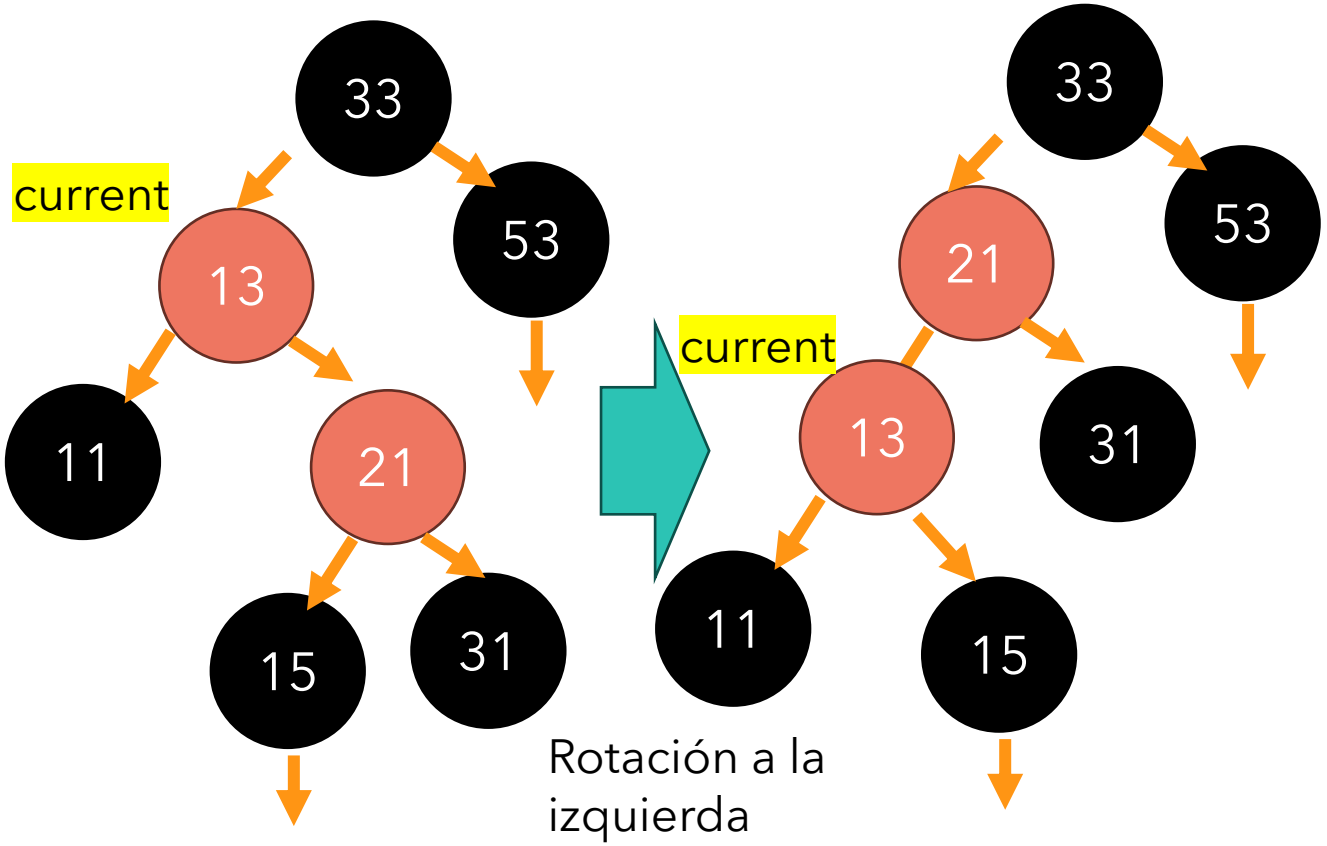
RB-INSERT_FIXUP

Para el algoritmo se corrige las propiedades del RBT:

- Repetir mientras p es RED
  - ❑ Si p es el hijo izquierdo de gp:

    CASO 2:

    1. En caso contrario, si current es el hijo derecho de p, asigne p como current
    2. Realice una rotación a la izquierda en current



Arboles balanceados - RBT

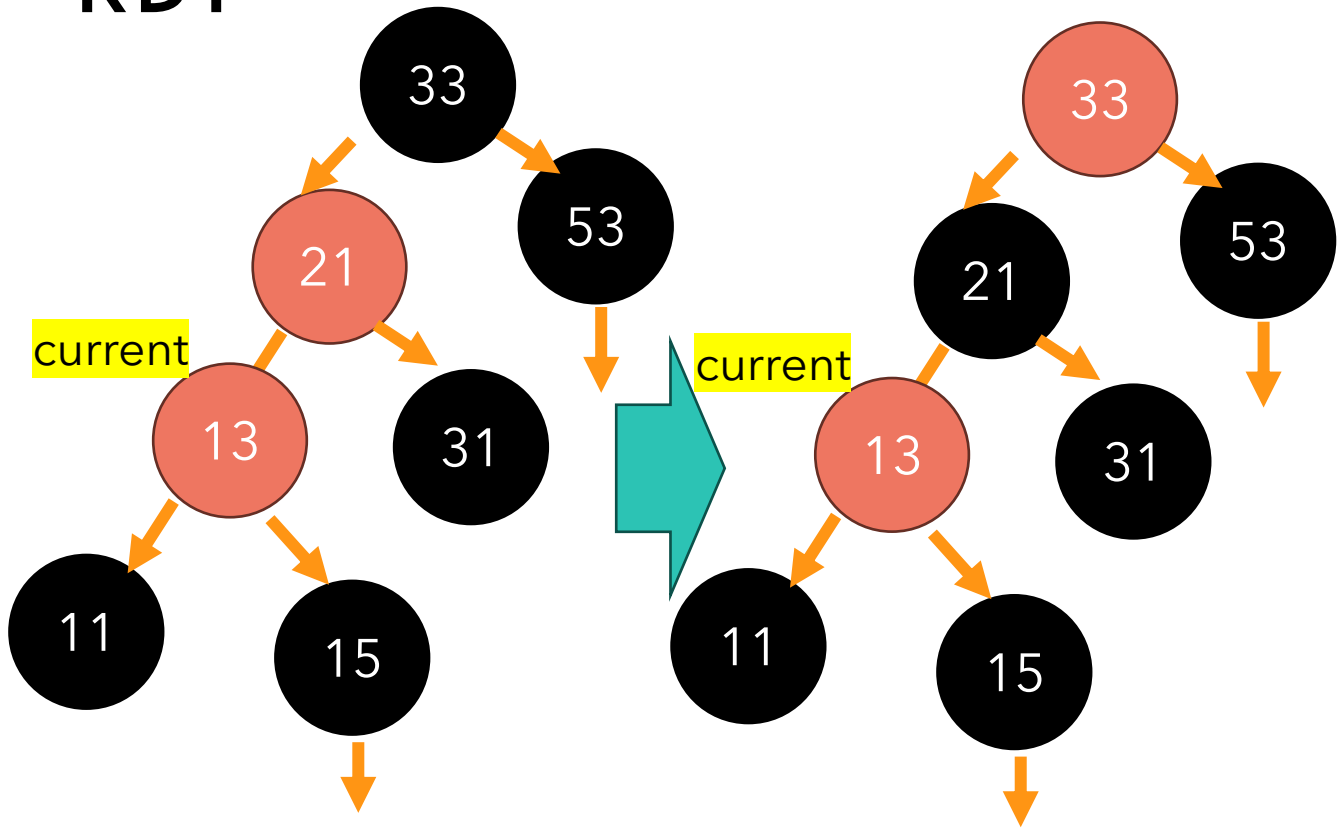
RB-INSERT_FIXUP

Para el algoritmo se corrige las propiedades del RBT:

- Repetir mientras p es RED
  - ❑ Si p es el hijo izquierdo de gp:

    CASO 3:

    1. Configure el color de p como BLACK y de gp como RED
    2. Realice una rotación a la derecha en gp



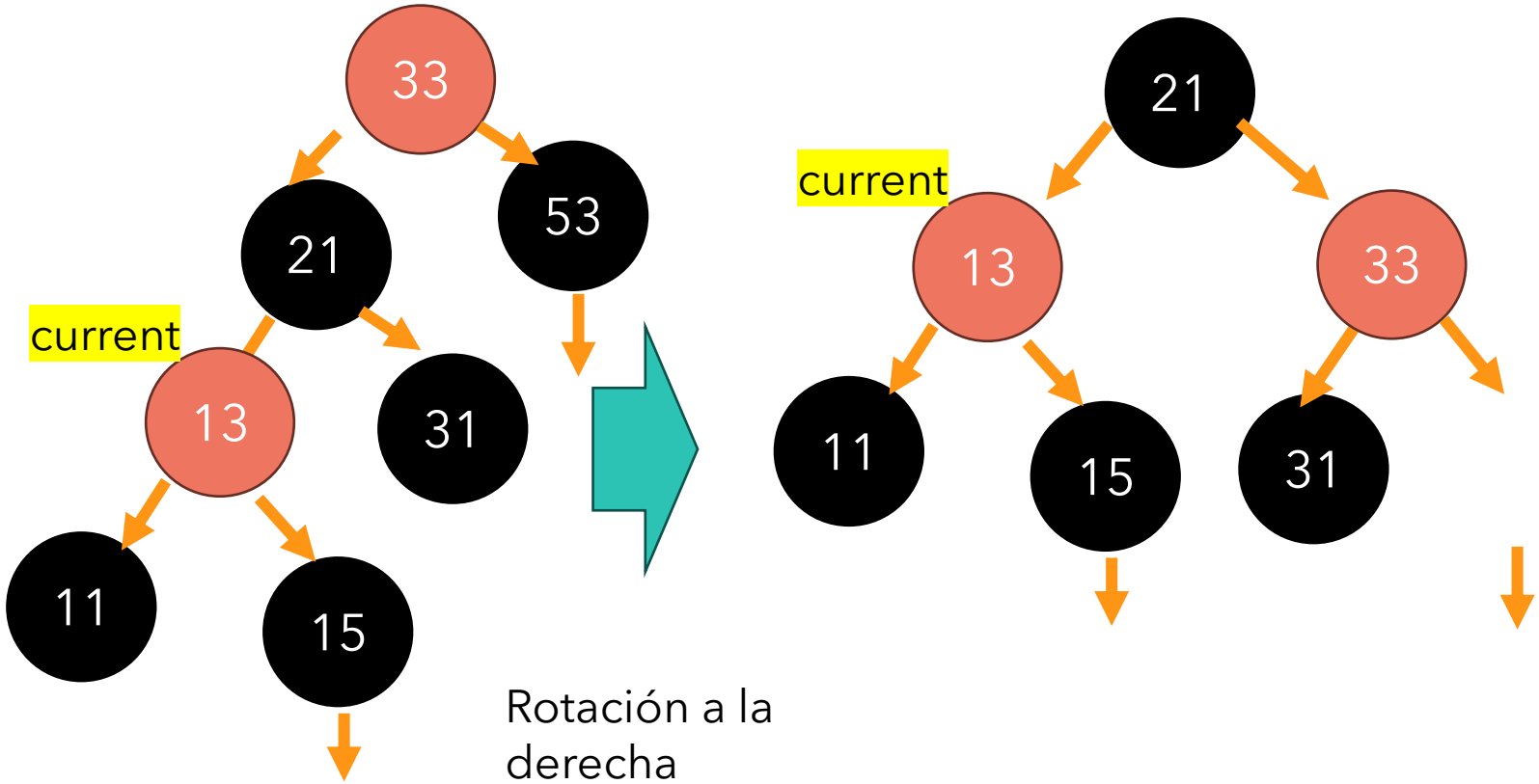
Arboles balanceados - RBT

RB-INSERT_FIXUP

Para el algoritmo se corrige las propiedades del RBT:

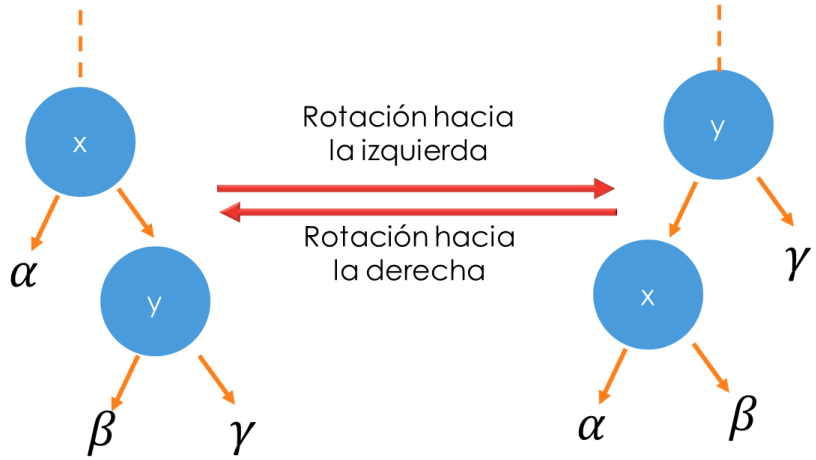
- Repetir mientras p es RED
  - ❑ Si p es el hijo izquierdo de gp:

  CASO 3:
  1. Configure el color de p como BLACK y de gp como RED
  2. Realice una rotación a la derecha en gp



Arboles balanceados - RBT

RB-INSERT_FIXUP



Para el algoritmo se corrige las propiedades del RBT:

- Repetir mientras p es RED
  - ❑ Si p es el hijo derecho de gp, de forma simetrica

  CASO 1:
  1. Si el hijo izquierdo de gp es RED, configure los colores de ambos hijos de gp como BLACK y el color de gp como RED
  2. Asigne gp como el nuevo current

  CASO 2:
  1. En caso contrario, si current es el hijo izquierdo de p, asigne p como current
  2. Realice una rotación a la derecha en current

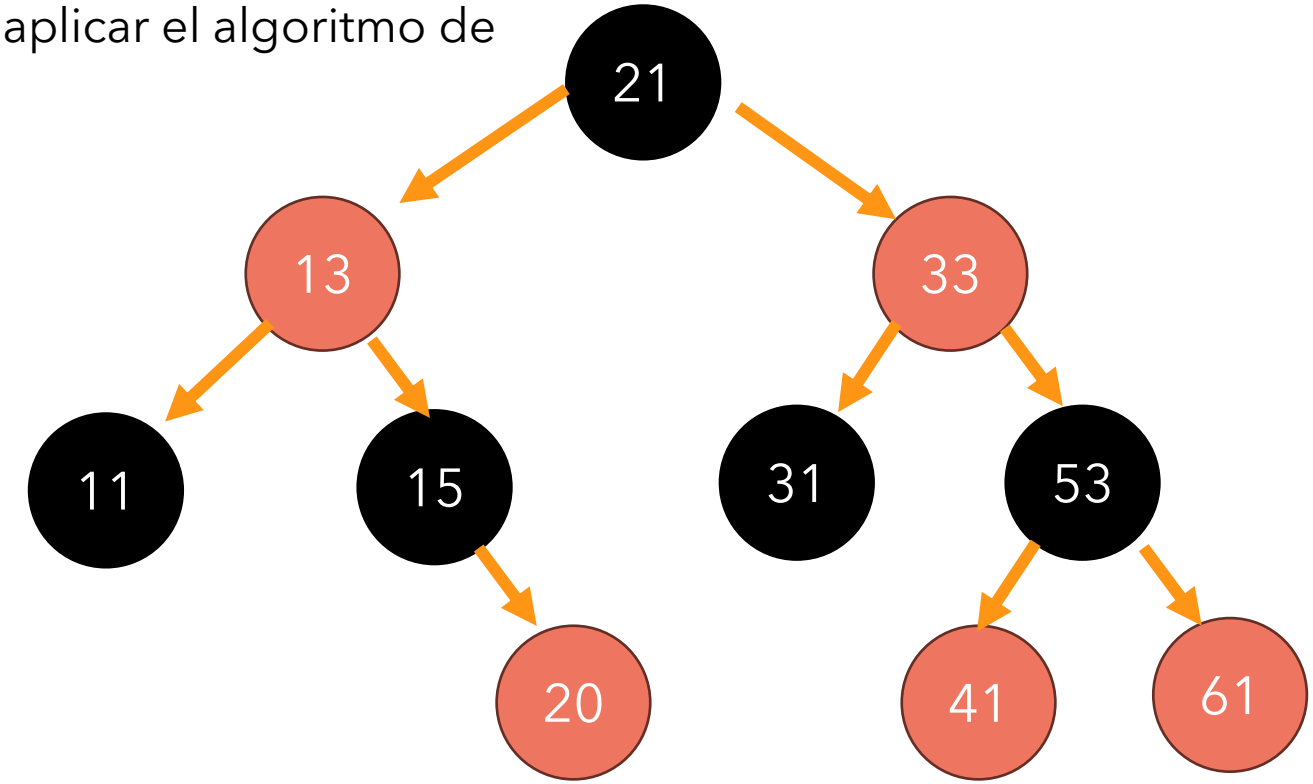
  CASO 3:
  1. Configure el color de p como BLACK y de gp como RED
  2. Realice una rotación a la izquierda en gp

  Al finalizar el procedimiento configure la raíz del árbol en BLACK

Arboles balanceados - RBT

RB-INSERT_FIXUP

Resultado obtenido al aplicar el algoritmo de insertar



Arboles balanceados - RBT

Aplicaciones

Para el algoritmo se corrige las propiedades del RBT:

- ❑ Indexación en base de datos, permitiendo una rápida búsqueda y consulta
- ❑ Implementación eficiente de algoritmos en grafos, como el camino más corto de Dijkstra
- ❑ Se emplean en Inteligencia Artificial para la implementación de arboles de decisión
- ❑ Algoritmos de criptografía
- ❑ Algoritmos de procesamiento de señales como las transformadas Wavelet

Desventajas

- ❑ Los arboles RED-BLACK son complejos de manejar para un alto numero de nodos
- ❑ Insertar datos puede ser lento comparado con otras estructuras de datos
- ❑ No es apropiado para conjuntos de datos muy grande (e.g. Big Data)
- ❑ Las operaciones de insertar y eliminar requieren un costo adicional para mantener el árbol balanceado