Análisis y Complejidad de Algoritmos

Arboles Rojinegros

Arturo Díaz Pérez

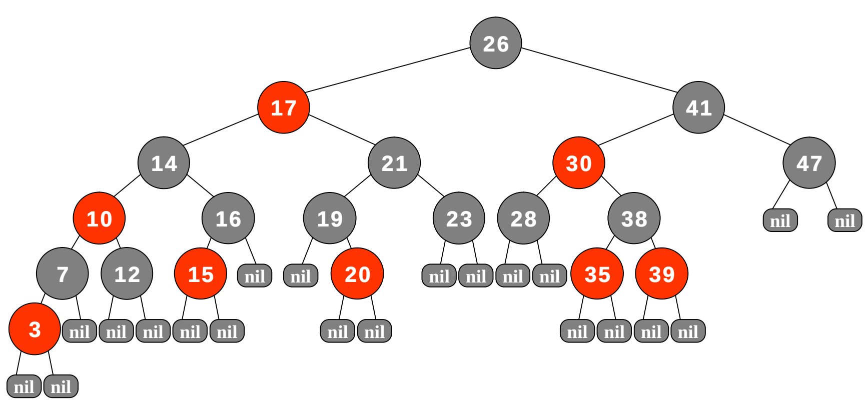
Definición

- Los árboles rojinegros son estructuras basadas en árboles binarios balanceados.
- Un árbol rojinegro es un árbol de búsqueda binaria que satisface las siguientes propiedades:
  - Cada nodo o es rojo o es negro
  - Cada hoja (nil) es negra
  - Si un nodo es rojo, entonces, sus hijos son negros
  - Cada camino de un nodo a cualquier descendiente tiene la misma cantidad de nodos negros

Definición

- ☞ La altura negra de un nodo *x*, *an*(x), es el número de nodos negros que hay en cualquier camino de él a una hoja que desciende de él.
- ☞ La altura negra de un árbol es la altura negra de su raíz.

Ejemplo



Observación

- Sea A un árbol rojinegro con n nodos internos
  - Si x es un nodo en A, entonces, el subárbol con raíz en x tiene a al menos $2^{an(x)}-1$ nodos internos.
- Sea x un nodo en A, por inducción sobre $an(x)$
  - A) Si $an(x) = 0$, entonces, x es hoja y no hay nodos en el subárbol con raíz en él.
  - Por otro lado, $2^{an(x)-1} = 0$.

Análisis y Diseño de Algoritmos *RedBlackTree*-5

Observación

- B) Supongamos que $an(x) > 0$, x es interno y tiene dos hijos. Es inmediato que cualquier hijo y de x cumple con lo siguiente:

  $C(y)$ = rojo, entonces, $an(y) = an(x)$

  $C(y)$ = negro, entonces, $an(y) = an(x) – 1$
- Por hipótesis de inducción, el número de nodos descendientes de x es al menos:

  No descendientes(dere(x)) + 1 + No_descendientes(izq(x)) = 1 + 2($2^{an(x)-1}$ - 1) = $2^{an(x)}$ -1

Análisis y Diseño de Algoritmos *RedBlackTree*-6

Observación

- La altura de A es a lo sumo $2\log_2(n+1)$.

- Sea h la altura del árbol rojinegro.
  - Al menos la mitad de los nodos en un camino de la raíz a una hoja , sin incluir la raíz, deben ser negros.
  - La altura negra del árbol debe ser al menos $h/2$
  - Por lo anterior,
  - $n = 2^{h/2 - 1} \Rightarrow \log_2(n+1) = h/2 \Rightarrow h = \log_2(n+1)$

Análisis y Diseño de Algoritmos *RedBlackTree*-7

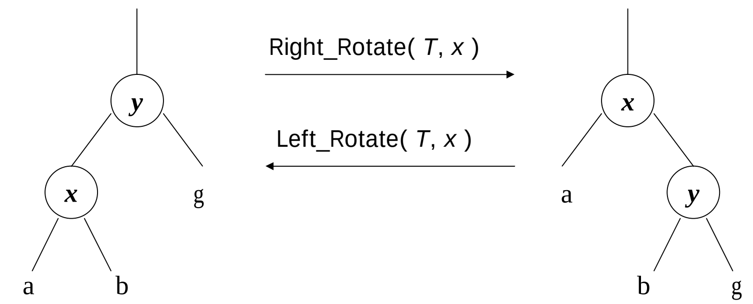
Observaciones

- Los árboles rojinegros son árboles de búsqueda binaria en donde las hojas tienen valores nulos.
- Los algoritmos para árboles de búsqueda binaria se aplican a los árboles rojinegros.
  - Al insertar una nueva llave quedará como padre de dos hojas nulas.
- La operación de búsqueda toma un tiempo $O(\log n)$ en el peor caso.

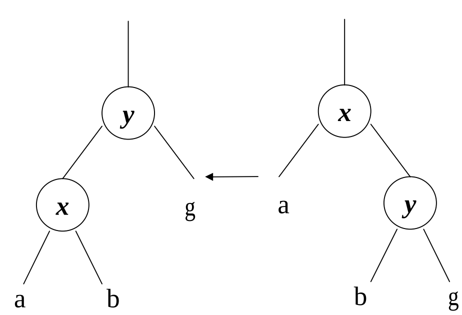
- Las operaciones de `Inserción` y `Supresión` son también $O(\log n)$ pero no necesariamente mantienen estructuras rojinegras.

Análisis y Diseño de Algoritmos *RedBlackTree*-8

Rotaciones

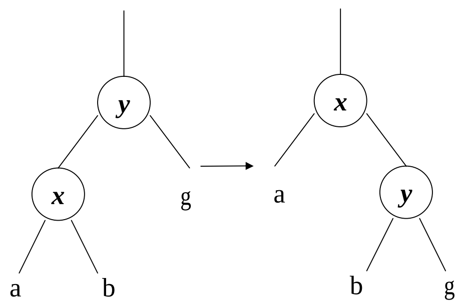


LeftRotate



LeftRotate*(A; x)*
{ *y:=right(x);*
right(x):=left(y);
if** left(y) ? nil **then
p(left(y)) := x;
p(y) := p(x);
if** p(x) = nil **then
root(A) := y
else
if** x := left(p(x)) **then
left(p(x)) := y
else
right(p(x)) := y;
left(y) := x; p(x) := y }

RightRotate




```
RighRotate(A; y)
{   x :=left(y);
    left(y):= right(x);
    if right(y) ? nil then
        p(right(x)) := y;
        p(x) := p(y);
    if p(y) = nil then
        root(A) := x
    else
        if  y := right(p(y)) then
            right(p(y)) := x
        else
            left(p(y)) := x;
    right(x) := y; p(y) := x }
```

Análisis y Diseño de Algoritmos *RedBlackTree-11*

Recorrido EntreOrden

```
RecorridoEntreOrden( x )
begin
    if  x <> nil then begin
        RecorridoEntreOrden( left[x] );
        writeln( x );
        RecorridoEntreOrden( right[x] )
    end
end;
```

Análisis y Diseño de Algoritmos *RedBlackTree-12*

Recorrido PreOrden

```
RecorridoPreOrden( x )
begin
    if  x <> nil then begin
        writeln( x );
        RecorridoPreOrden( left[x] );
        RecorridoPreOrden( right[x] )
    end
end;
```


Recorrido PosOrden

```
RecorridoPosOrden( x )
begin
    if  x <> nil then begin
        RecorridoPosOrden( left[x] );
        RecorridoPosOrden( right[x] )
        writeln( x );
    end
end;
```

Recorrido EntreOrden

- Las rotaciones izquierda y derecha preservan entreórdenes. Esto es, si A es el recorrido entreorden, *x* está antes que *y* en A y B es el recorrido entreorden después de aplicar una rotación en *x*, entonces, *x* está antes que *y* en B.
  - En efecto, consideremos situación derecha de la figura y una rotación izquierda sobre *y*.
  - Se tiene $\alpha = x = \beta = y = \gamma$.
  - Después de aplicar la rotación a la izquierda el orden se preserva.

Inserción

- Al insertar nuevos elementos en árboles rojinegros, en tiempo $O(\log n)$, se debe garantizar que el árbol resultante continúe siendo **rojinegro**
- En los árboles **rojinegros** las hojas son vacías y negras
- Los nuevos elementos se colocan como padres de hojas
- Cada nuevo elemento se coloca como una estructura del árbol binario
- Como las hojas deben ser **negras**, el nodo que contiene la llave a insertarse se colorea como **rojo**
  - No se aumenta la altura negra del árbol
- La única propiedad que puede violarse es la referente al color del padre del nodo que se ha insertado

InserciónRN

```
InserciónRN( A; x )
{   InserciónABB( A; x );
    C(x) := red;
    p := p(x);
    while x ? root(A) ^ C(p) = red do {
        if p = left (p(p(x)) then {
            u := right(p(p(x));
            if C(u) = red then {
1)              C(p) := C(u) := black;
                C(p(p(x)) := red;
                x:= p(p(x)); p := p(x)
            } else {
            . . .
```

Análisis y Diseño de Algoritmos RedBlackTree-17

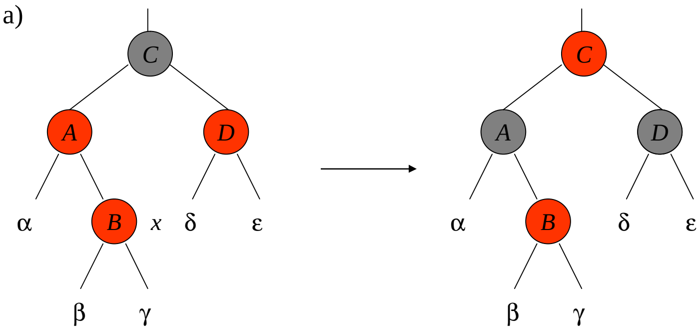
InserciónRN (cont.)

```
                . . .
                    if x = right(p) then {
2)                      x := p;
                        LeftRotate( A; x ); p := p(x)
                    };
3)                  C(p(x)) := black;
                    C(p(p(x)) := red;
                    RightRotate( A; p(p(x)) )
                }
            } else {
                (código simétrico intercambiando "left" y "right" )
            };
        };
        C(root(A)) :=  black
    };
```

Análisis y Diseño de Algoritmos RedBlackTree-18

InserciónRN(*A*, *x*): Caso 1

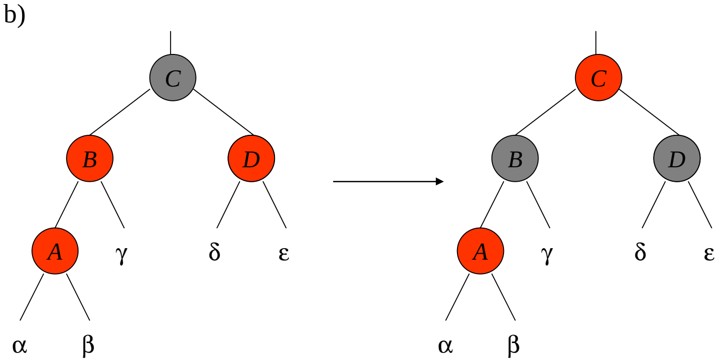
a)



Análisis y Diseño de Algoritmos *RedBlackTree*-19

InserciónRN(*A*, *x*): Caso 1

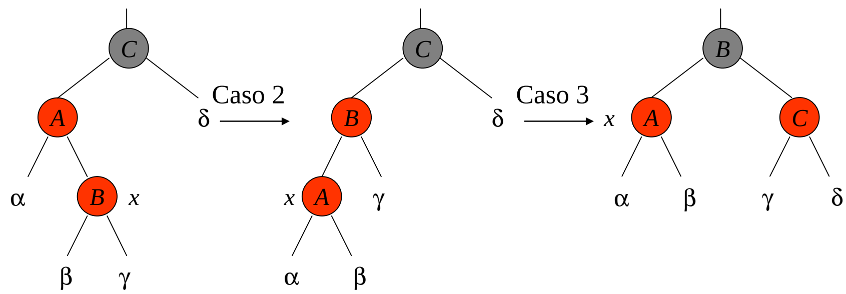
b)



Si *x* rojo y ambos padre y tío rojo, repintamos a estos últimos de negro y al padre de ambos de rojo. Como A es RN, ese abuelo era negro. Al hacer los cambios, alturas negras no se modifican. Trasladamos la dificultad en *x*, a su abuelo *padre(padre)* y reiteramos ciclo principal.

Análisis y Diseño de Algoritmos *RedBlackTree*-20

InserciónRN(*A*, *x*): Casos 2 y 3



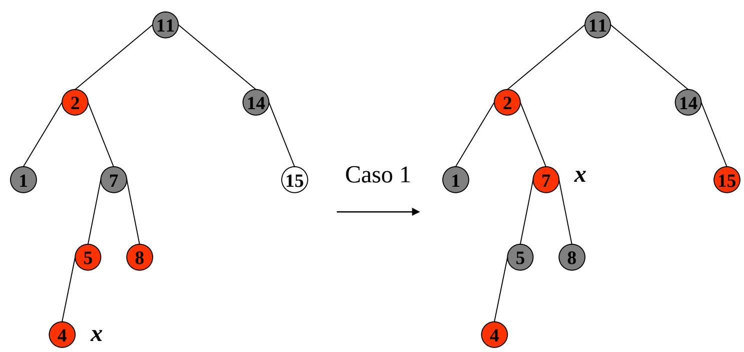
Análisis y Diseño de Algoritmos *RedBlackTree-21*

Casos 2 y 3

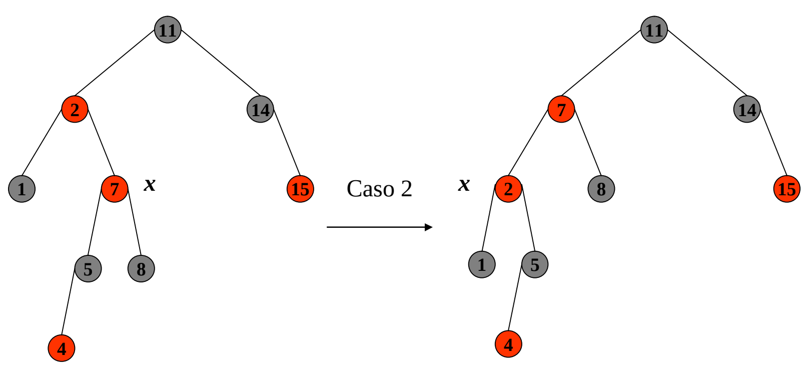
- **Caso 2**: *x* y *padre* rojo mas *tío* negro y, *x* hijo derecho de *padre*.
  - La rotación izquierda sobre *padre* reduce el caso al de 3). No se afecta a las alturas negras.
- **Caso 3**: *x* y *padre* rojo mas tío es negro y, x es hijo izquierdo de *padre*
  - Repintar a *padre* negro, al abuelo rojo y rotación derecha sobre el abuelo.
  - Las alturas negras no se modifican y la posición del abuelo quedaría negra, con ambos hijos rojo.
  - El ciclo principal no se repetirá más.

Análisis y Diseño de Algoritmos *RedBlackTree-22*

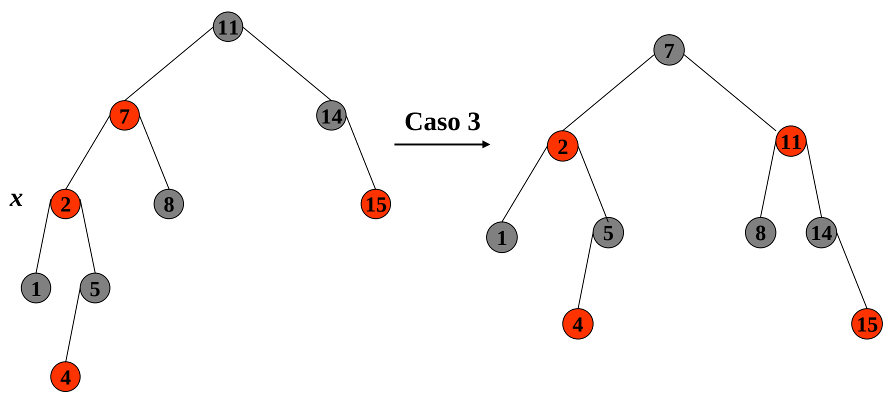
InserciónRN(*A*, *x*)



InserciónRN(*A*, *x*)

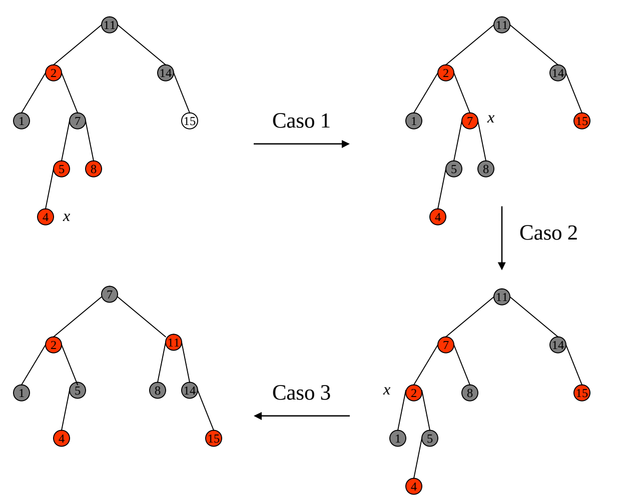


InserciónRN(*A*, *x*)



Análisis y Diseño de Algoritmos *RedBlackTree*-25

InserciónRN(*A*, *x*)



Análisis y Diseño de Algoritmos *RedBlackTree*-26

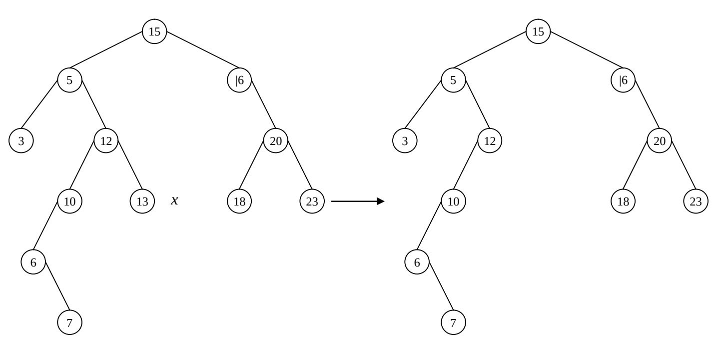
Supresión

- Coincide con la operación de supresión en los árboles de búsqueda binaria
- Sin embargo, es necesario mantener la estructura de los árboles **roji****negros**
- Para evitar repeticiones se introduce vigía, nil(A), el cual representa las hojas del árbol **roji****negro**
- Si el nodo suprimido *y* fuese rojo, entonces, las alturas negras no cambian y, en tal caso, termina
- Si el nodo suprimido *y* fuese negro, entonces, las ramas que pasen por *y* tienen un nodo negro menos, lo que viola la condición de los árboles **roji****negros**. En este caso es necesario ajustar colores.

Análisis y Diseño de Algoritmos *RedBlackTree-27*

SupresiónABB(*A*, *x*): 0 ó 1 hijo

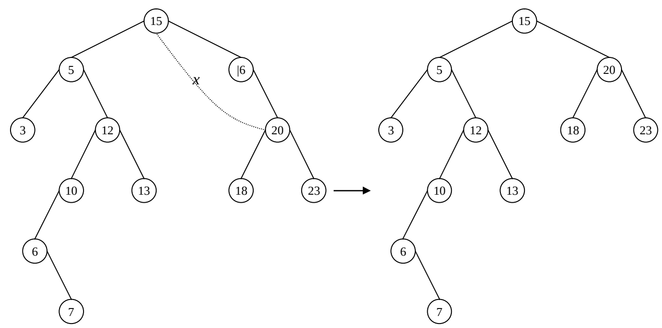
a)



Análisis y Diseño de Algoritmos *RedBlackTree-28*

SupresiónABB(*A*, *x*): 0 ó 1 hijo

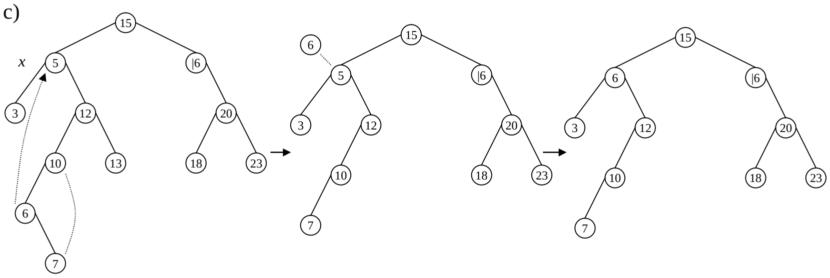
b)



Análisis y Diseño de Algoritmos *RedBlackTree*-29

SupresiónABB(*A*, *x*): 2 hijos

c)



Análisis y Diseño de Algoritmos *RedBlackTree*-30

SupresionRN

```
SupresionRN( A, z )
{
   if (left(z) = nil(A)) or (right(z) = nil(A))
      then   y := z;
      else   y := SucesorABB(z);
   if left(y) ? nil(A)
      then   x := left( y );
      else   x := right( y );
   p(x) := p(y);
   if p(y) = nil
      then   root( A ) := 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 );
   if C(y) = black then AjustarSupresionRN( A, x )
}
```

Análisis y Diseño de Algoritmos RedBlackTree-31

AjustarSupresionRN

```
AjustarSupresionRN( A, x )
{  /* Ciclo principal */
   while x ? root(A) and C(x) = black do {
      if x = left( p(x) ) then {
         w := right( p(x) );
         if C(w) = red then {
1)          C(w) := black;
            C(p(x)) := red;
            LeftRotate( A, p(x) );
            w := right( p(x) )
         };
         if ambos hijos de w son negros then {
2)          C(w) := red;  x := p(x)
         } else {
         . . .
```

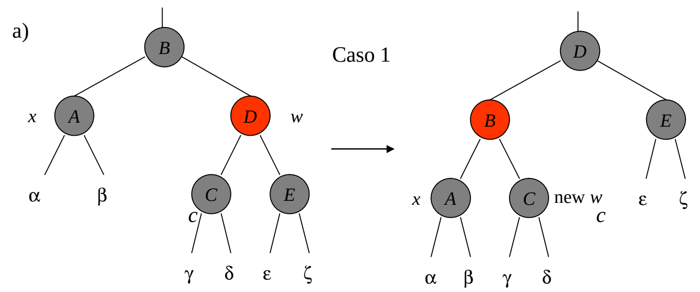
Análisis y Diseño de Algoritmos RedBlackTree-32

AjustarSupresionRN (cont.)

```
                if  C(right(w)) = black then {
3)                  C( left(w) ) := black;   C(w) := red;
                    RightRotate( A, w );
                    w := right( p(x) )
                };
4)              C(w) := C(p(x));  C(p(x) := black;
                C(right(w)) := black;
                LeftRotate( A, p(x) );
                x := root( A )
            }
        } else
            { código simétrico intercambiando "left" y "right" }
    };
    C(x) := black
};
```

Análisis y Diseño de Algoritmos — *RedBlackTree-33*

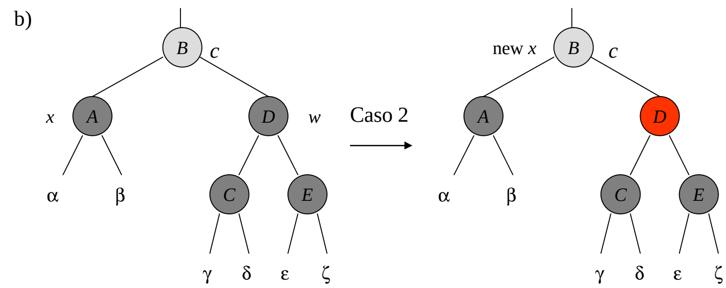
SupresiónRN(*A*, *x*): Caso 1



x **negro y *hermano* rojo. A *hermano* se le pinta negro, a *padre* rojo y rotar a la izquierda. Se obtiene configuración como en 2).**
Al ejecutarse 2) se concluirá el ciclo principal pues x actual roja.

Análisis y Diseño de Algoritmos — *RedBlackTree-34*

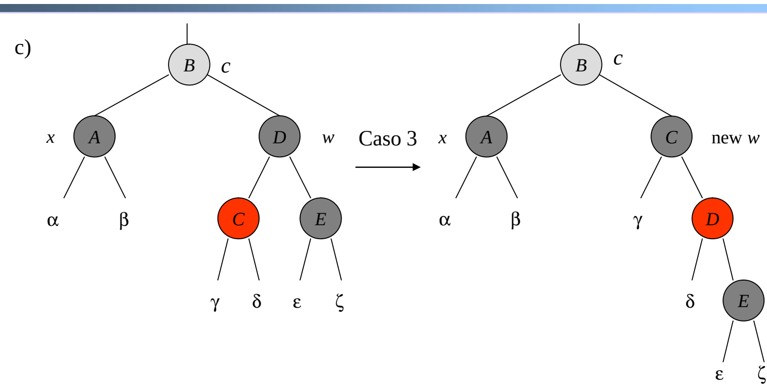
SupresiónRN(*A*, *x*): Caso 2



x y *hermano* negros, así como ambos hijos de *hermano*. A *hermano* se le pinta rojo y se fuerza a padre a llevar marca negra de “acarreo”

Análisis y Diseño de Algoritmos *RedBlackTree*-35

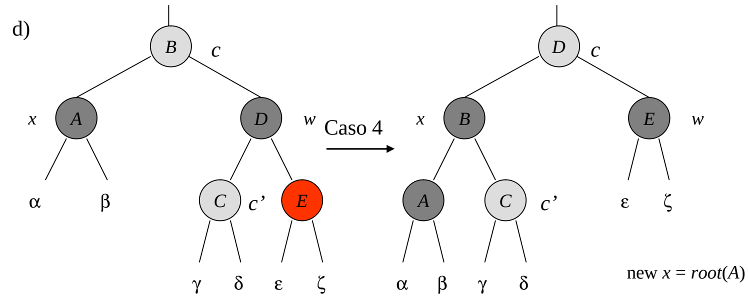
SupresiónRN(*A*, *x*): Caso 3



x y *hermano* negros, el hijo derecho de *hermano* negro y el izquierdo rojo.
Al hijo izquierdo de *hermano* se le pinta negro y a hermano de rojo.
Rotar a la derecha para obtener la configuración en 4).

Análisis y Diseño de Algoritmos *RedBlackTree*-36

SupresiónRN(*A*, *x*): Caso 4



x y *hermano* negros, el hijo derecho de hermano rojo y el izquierdo negro.
A *padre* se le pinta negro, a hermano del que tenía padre y al hijo derecho de hermano también negro.
Rotar a la izquierda para equilibrar alturas negras.

Análisis y Diseño de Algoritmos — *RedBlackTree-37*

Conclusiones

- Las operaciones de inserción y supresión recorren un árbol **rojinegro** a lo más dos veces.
- Por tanto, su complejidad en tiempo para el peor caso está acotada por *O*(log *n*)
- ¿Cómo se podría plantear el análisis del caso promedio?
- Cómo se puede analizar el mejor caso?

Análisis y Diseño de Algoritmos — *RedBlackTree-38*