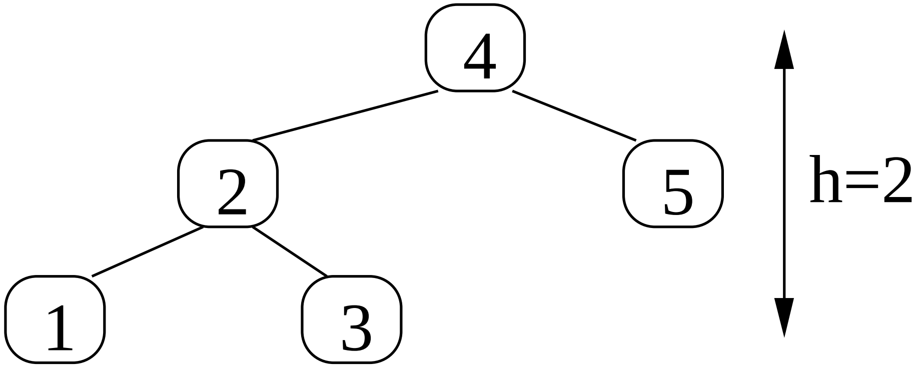
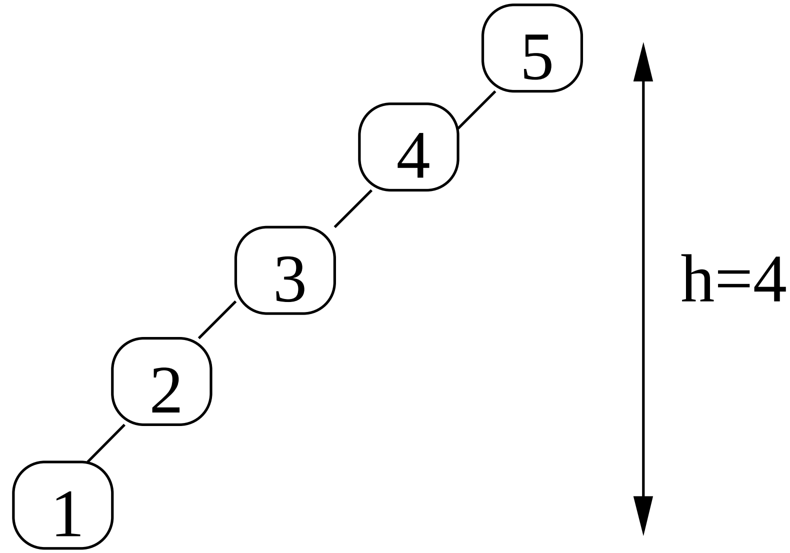
Binary Search Trees (BST)

- T – drzewo BST
- *root*$[T]$ – wskaźnik na korzeń (*root*$[T] = NIL$ – drzewo puste)
- *key*$[x]$ – klucz w węźle x
- *p*$[x]$ – wskaźnik na rodzica x (dla korzenia: NIL)
- *left*$[x]$ – wskaźnik na lewego syna (korzeń lewego poddrzewa) x (*left*$[x] = NIL$ – brak lewego syna)
- *right*$[x]$ – wskaźnik na prawego syna (korzeń prawego poddrzewa) x (*right*$[x] = NIL$ – brak prawego syna)

Porządek *inorder*: Dla każdego węzła x wszystkie klucze w lewym poddrzewie x są $\leq$*key*$[x]$, a wszystkie klucze w prawym poddrzewie x są $\geq$*key*$[x]$.

BST






```
Inorder-Tree-Walk(x)
if x ≠ NIL then
    Inorder-Tree-Walk(left[x])
    wypisz key[x]
    Inorder-Tree-Walk(right[x])
```

Czas: $O(n)$ (każdy węzeł – 2 rekurencyjne wywołania)

BST – Search

```
Tree-Search(x, k)
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)
```

Czas: $O(h)$ (h – wysokość poddrzewa o korzeniu x)

BST – Search

```
Tree-Search(x, k)
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)
```

Czas: $O(h)$ (h – wysokość poddrzewa o korzeniu x)

```
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]
return x
```

Minimum Maximum

Tree-Minimum(x)

while *left*$[x] \neq NIL$ do

   $x \leftarrow$ *left*$[x]$

return x

Czas: $O(h)$

Tree-Maximum(x)

while *right*$[x] \neq NIL$ do

   $x \leftarrow$ *right*$[x]$

return x

Czas: $O(h)$

Successor

```
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
```

Czas: $O(h)$ (h wysokość całego drzewa zawierającego x)

Uwaga: zwraca NIL gdy x nie ma następnika.

Successor

```
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
```

Czas: $O(h)$ (h wysokość całego drzewa zawierającego x)
Uwaga: zwraca NIL gdy x nie ma następnika.

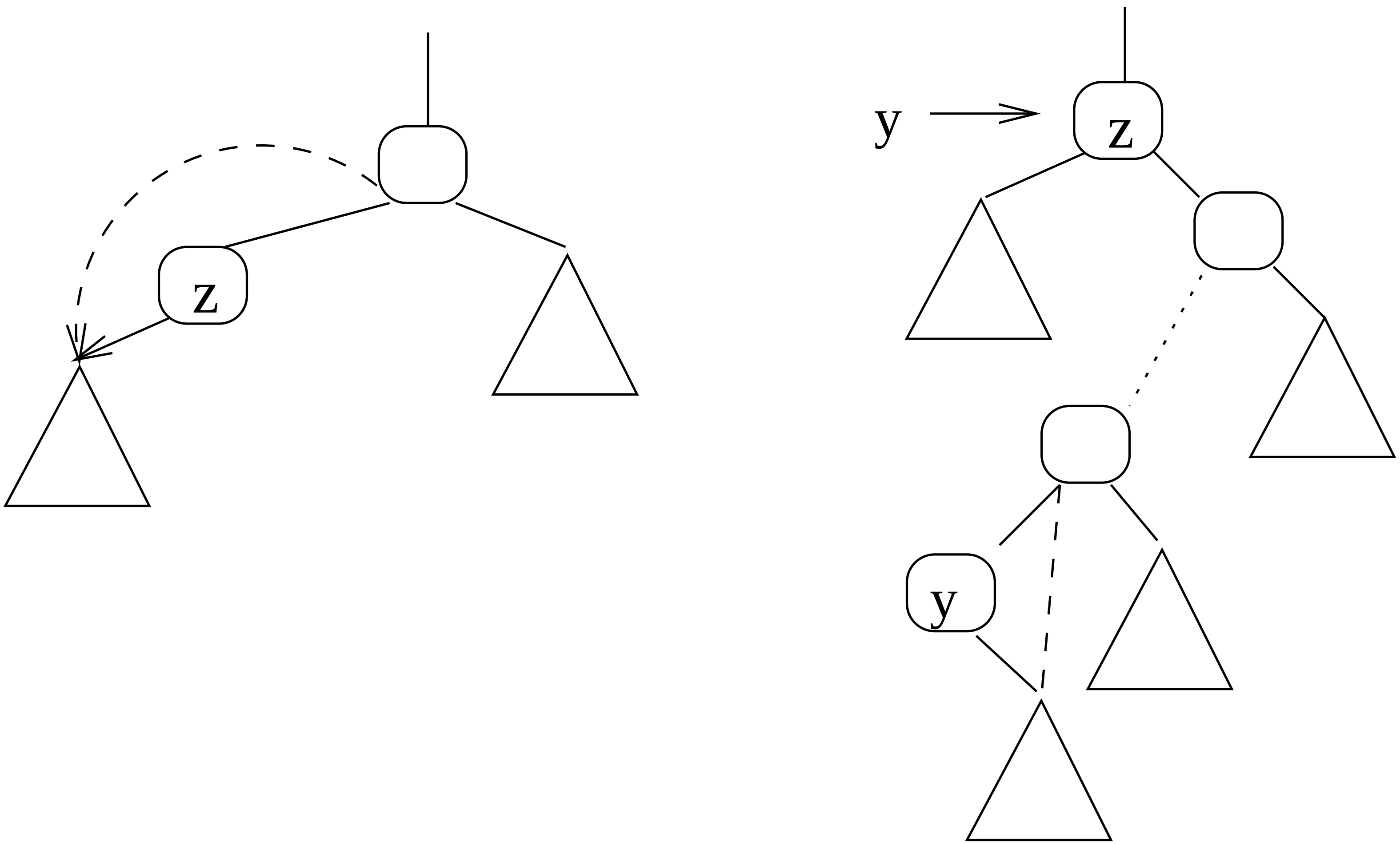
Tw. Operacje na zb. dynamicznych: `Search`, `Minimum`, `Maximum`, `Succesor`, `Predecessor` są wykonywane na BST wysokości h w czasie $O(h)$.

Insert

Wstawiamy z (*key*$[z]$ – klucz, *left*$[z]$ =*right*$[z]$ $= NIL$)

```
Tree-Isert(T, z)                    ▷ Czas:  O(h)
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
```

Delete



Delete

```
Tree-Delete(T, z)
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
...
```

Delete

...
if $y \neq z$ then
    $\mathit{key}[z] \leftarrow \mathit{key}[y]$
    $\triangleright$ skopiuj pozostałe pola danych y do z
return y

Czas: $O(h)$

Delete

...

if $y \neq z$ **then**

    key$[z] \leftarrow$*key*$[y]$

    $\triangleright$ skopiuj pozostałe pola danych y do z

return y

Czas: $O(h)$

Tw. Operacje `Insert`, `Delete` działają na drzewie BST w czasie $O(h)$, gdzie h – wysokość drzewa.

Drzewa czerwono-czarne

- T – czerwono-czarne drzewo BST (porządek inorder)
- *root*$[T]$ – wskaźnik na korzeń (*root*$[T] = NIL$ – drzewo puste)
- *key*$[x]$ – klucz w węźle x
- *p*$[x]$ – wskaźnik na rodzica x (dla korzenia: NIL)
- *left*$[x]$ – wskaźnik na lewego syna (korzeń lewego poddrzewa) x (*left*$[x] = NIL$ – brak lewego syna)
- *right*$[x]$ – wskaźnik na prawego syna (korzeń prawego poddrzewa) x (*right*$[x] = NIL$ – brak prawego syna)
- *color*$[x]$ – RED albo $BLACK$
- traktujemy stałe NIL wskazywane przez pola *left* i *right* jako liście drzewa (*color*$[NIL] = BLACK$)

Własności czerwono-czarne

1. każdy węzeł jest czerwony albo czarny
2. każdy liść (NIL) jest czarny
3. jeśli węzeł jest czerwony to obaj jego synowie są czarni
4. każda (prosta) ścieżka z ustalonego węzła do liścia ma tyle samo czarnych węzłów.

Czarna wysokość węzła x ($bh(x)$) – liczba czarnych węzłów na ścieżce z x do liścia (wykluczając x).

Lemat. Drzewo czerwono-czarne o n węzłach *wewnętrznych* ma wysokość $\leq 2\lg(n+1)$.

Dowód Lematu o wysokości

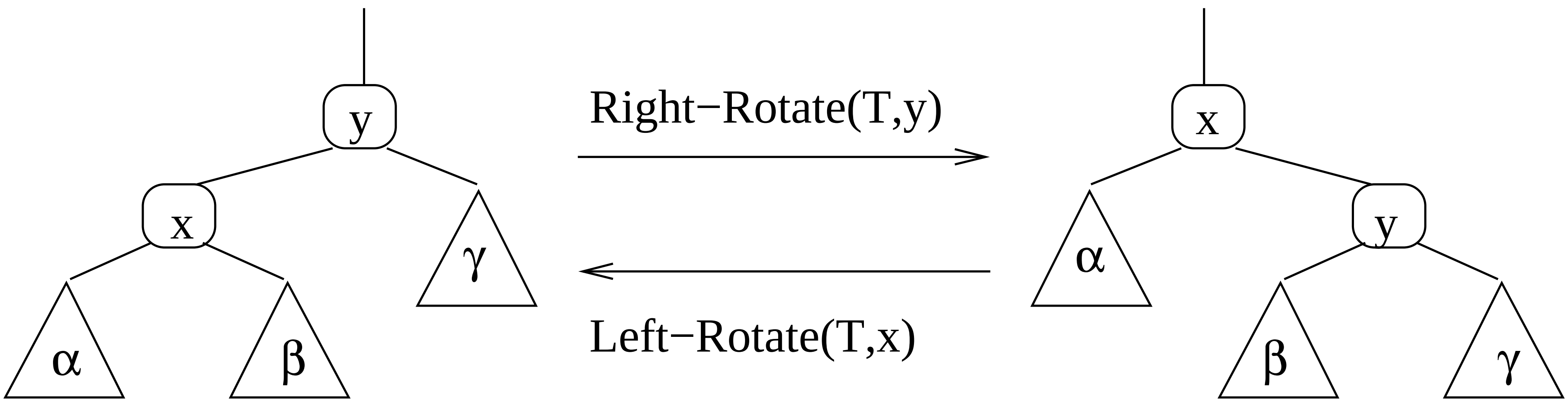
D-d. *Fakt*: każde poddrzewo o korzeniu x ma $\geq 2^{bh(x)} - 1$ węzłów wewnętrznych.
Podstawa indukcji: $bh(x) = 0$. Wtedy x – liść (NIL). Jest $2^0 - 1 = 0$ węzłów wewnętrznych w poddrzewie x.
Krok inukcyjny: Niech x – węzeł wewnętrzny o dodatniej wysokości. Każdy z synów x ma wysokość albo $bh(x)$(jeśli jest czerwony) albo $bh(x) - 1$ (jeśli jest czarny). Z zał. ind.: x ma $\geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ węzłów wewnętrznych. *Koniec dowodu Faktu*.
Z własności 3. co najmniej połowa węzłow na ścieżce od korzenia do liścia (nie licząc korzenia) jest czarna. Stąd czarna wysokość korzenia jest $\geq h/2$, czyli $n \geq 2^{h/2} - 1$.
Równoważnie: $\lg(n+1) \geq h/2$ czyli $h \leq 2\lg(n+1)$. $\Box$

drzewa czerwono-czarne

Wn. `Search`, `Minimum`, `Maximum`, `Successor`, `Predecessor` w wersji dla BST działają w czasie $O(\lg n)$. `Tree-Insert`, `Tree-Delete` też działają w czasie $O(\lg n)$, ale psują własności czerwono-czarne.

drzewa czerwono-czarne

Wn. `Search`, `Minimum`, `Maximum`, `Successor`, `Predecessor` w wersji dla BST działają w czasie $O(\lg n)$. `Tree-Insert`, `Tree-Delete` też działają w czasie $O(\lg n)$, ale psują własności czerwono-czarne.



Left-Rotate

Left-Rotate(T,x)
$y \leftarrow \mathit{right}[x]$ ▷ y będzie podnoszony
$\mathit{right}[x] \leftarrow \mathit{left}[y]$ ▷ poddrzewo β - pod x
if $\mathit{left}[y] \neq NIL$
    then $p[\mathit{left}[y]] \leftarrow x$ ▷ popraw p[korzeń β]
$p[y] \leftarrow p[x]$ ▷ były ojciec x będzie ojcem y
if $p[x] = NIL$
    then $\mathit{root}[T] \leftarrow y$ ▷ y - nowy korzeń T
    else if $x = \mathit{left}[p[x]]$ ▷ y - pod $p[x]$
            then $\mathit{left}[p[x]] \leftarrow y$
            else $\mathit{right}[p[x]] \leftarrow y$
$\mathit{left}[y] \leftarrow x$ ▷ x - pod y
$p[x] \leftarrow y$
Czas: $O(1)$

RB-Insert

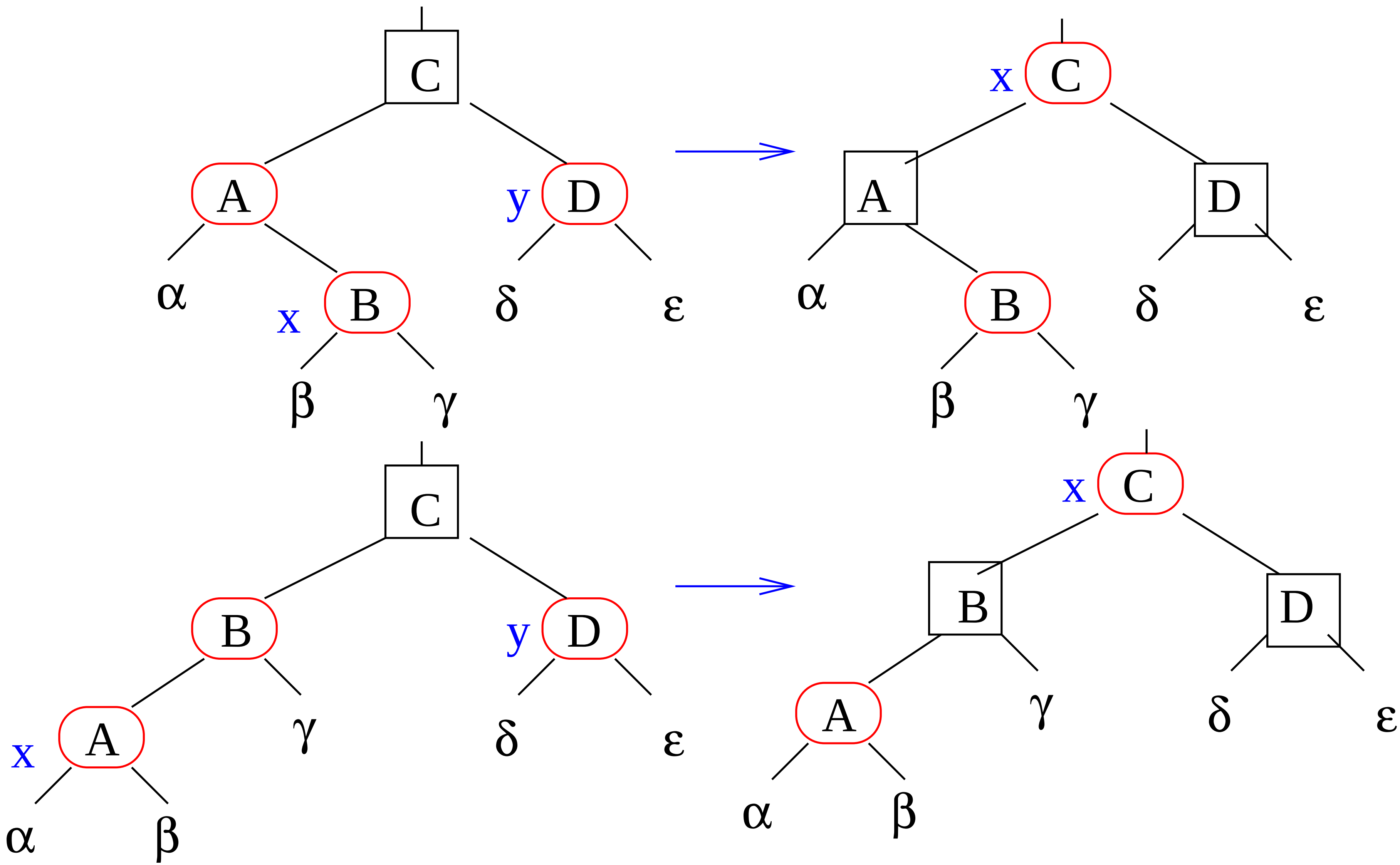
Zakładamy, że korzeń T – zawsze czarny (**)

RB-Insert(T, x)

Tree-Insert(T, x)
color$[x] \leftarrow RED$ ▷ może naruszyć własność 3.
while $x \neq$ *root*$[T]$ and *color*[*p*$[x]$]$= RED$ do
    if *p*$[x] =$ *left*[*p*[*p*$[x]$]] then ▷ (*)
        $y \leftarrow$ *right*[*p*[*p*$[x]$]] ▷ *p*[*p*$[x]$] istnieje (**)
        if *color*$[y] = RED$ then ▷ przypadek 1
            color[*p*$[x]$]$\leftarrow BLACK$
            color$[y] \leftarrow BLACK$
            color[*p*[*p*$[x]$]]$\leftarrow RED$
            $x \leftarrow$ *p*[*p*$[x]$]
        else

...

RB-Insert - przypadek 1



RB-Insert

...

else ▷ może być: $y = NIL$

if $x = right[p[x]]$ then ▷ przypadek 2

$x \leftarrow p[x]$

Left-Rotate(T,x)

$color[p[x]] \leftarrow BLACK$ ▷ przypadek 3

$color[p[p[x]]] \leftarrow RED$

Right-Rotate(T,$p[p[x]]$)

else tak jak (*) z zamienionymi *left* i *right*

$color[root[T]] \leftarrow BLACK$ ▷ wymuszamy (**)

Czas: $O(h) = O(\lg n)$

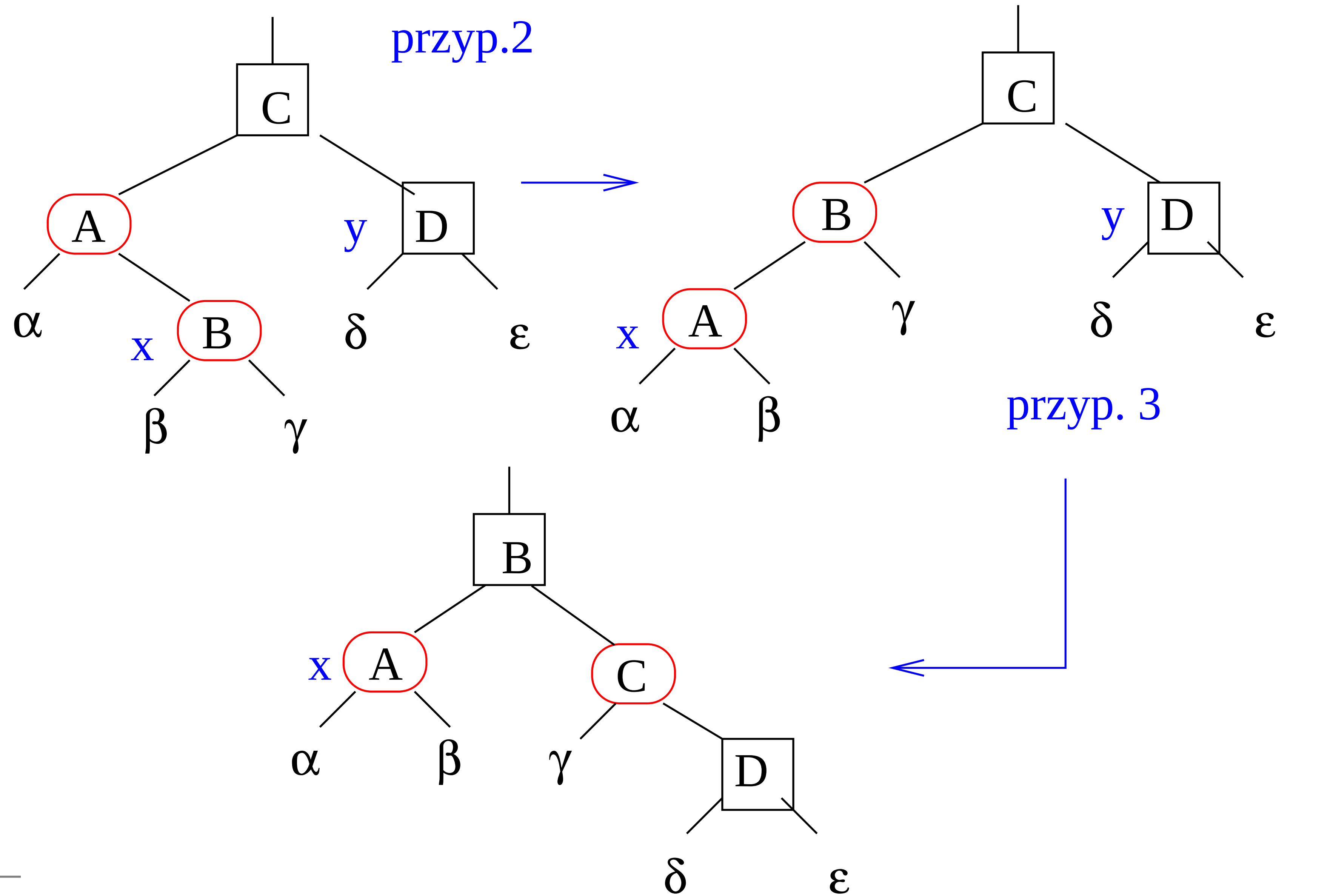
Przypadek 1 – w $O(1)$ przesuwa x dwa piętra w górę.

Przypadek 2 – w $O(1)$ sprowadzamy do przypadku 3.

Przypadek 3 – w $O(1)$ sprowadzamy do przerwania pętli.

Uwaga: zawsze ≤ 2 rotacje

RB-Insert - przypadki 2 i 3



RB-Delete

Zamiast NIL – wartownik *nil*$[T]$.

```
RB-Delete(T, z)
if left[z] = nil[T] or right[z] = nil[T]
   then y ← z
   else y ← Tree-Successor(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
...
```

RB-Delete

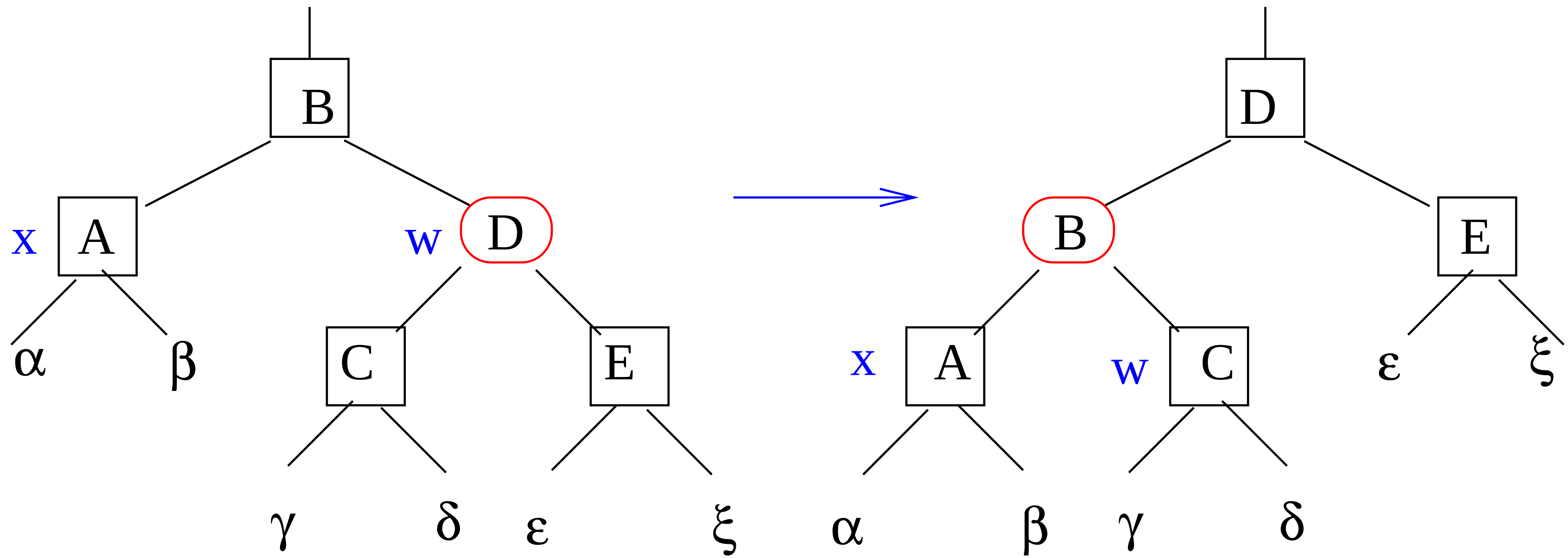
...
if $y \neq z$ then
    key$[z] \leftarrow$*key*$[y]$
    $\triangleright$ skopiuj pozostałe pola danych y do z
if *color*$[y] = BLACK$
    then RB-Delete-Fixup(T,x)
return y

Jeśli usunięty y był czarny to może zostać naruszona własność 4. (każda ścieżka zawierająca poprzednio y ma o 1 czarny węzeł mniej). Zakładamy, że x dziedziczy po y czarną jednostkę. Jeśli x był czarny, to staje się “podwójnie czarny”.

RB-Delete-Fixup

RB-Delete-Fixup(T, x)
while $x \neq$ *root*$[T]$ and *color*$[x] = BLACK$ do
    if $x =$ *left*[*p*$[x]$] then ▷ (*)
        $w \leftarrow$ *right*[*p*$[x]$]
        ▷ w nie-liść bo x dwu-czarny
        if *color*$[w] = RED$ then ▷ przyp. 1
            color$[w] \leftarrow BLACK$
            color[*p*$[x]$] $\leftarrow RED$
            Left-Rotate(T, *p*$[x]$)
            $w \leftarrow$ *right*[*p*$[x]$]
...

RB-Delete-Fixup: przypadek 1



RB-Delete-Fixup

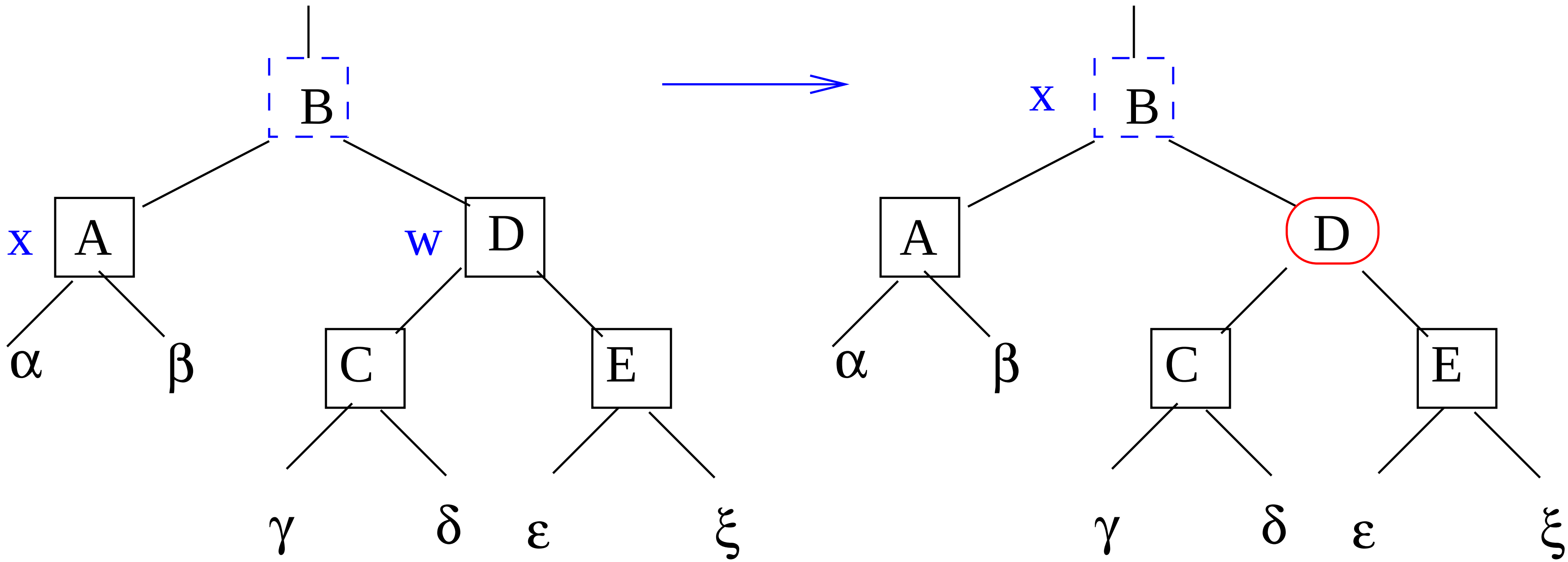
...

if *color*[*left*[w]] $= BLACK$ **and**
    color[*right*[w]] $= BLACK$ **then** $\triangleright$ prz.2
  color[w] $\leftarrow RED$
  $x \leftarrow$ *p*[x]
else

...

RB-Delete-Fixup: przypadek 2

Obaj synowie w są czarni.

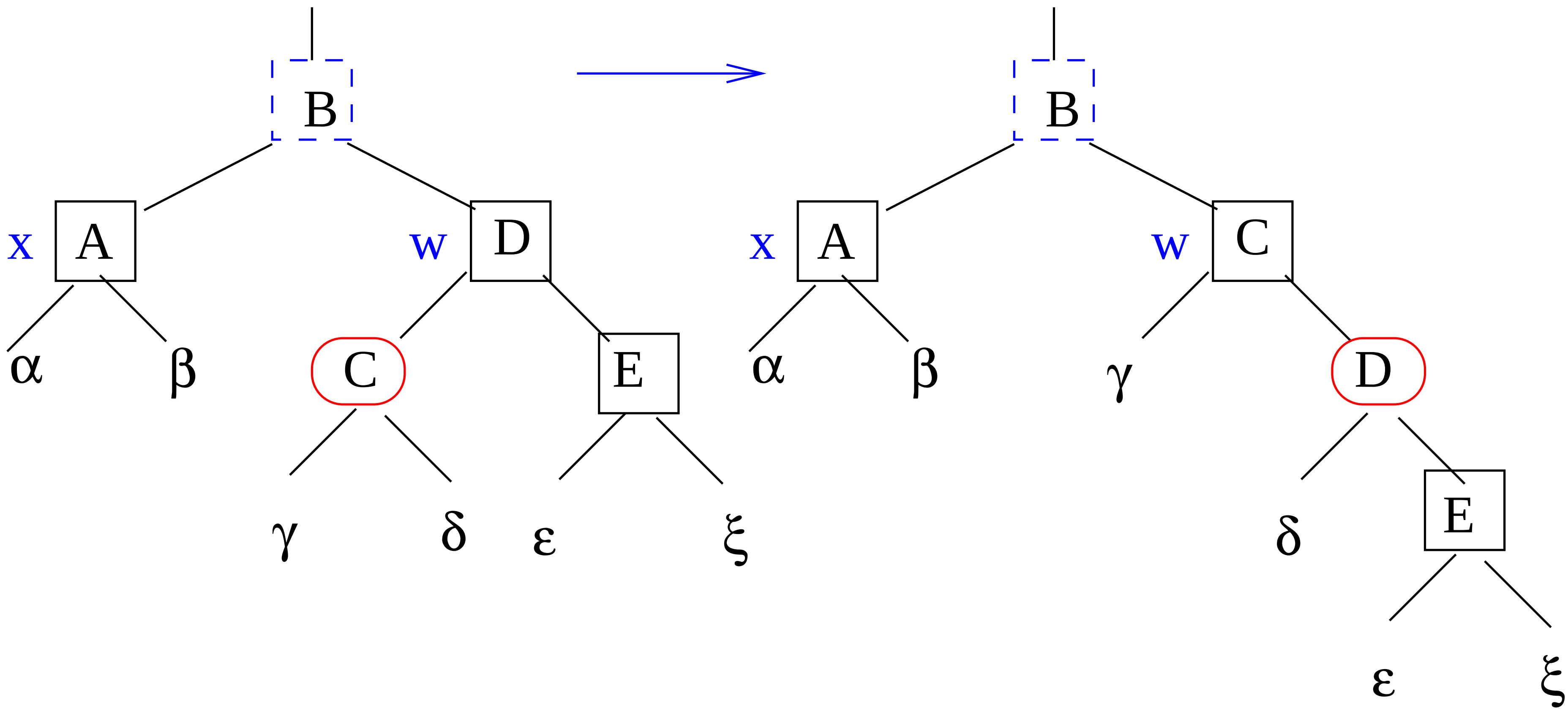


RB-Delete-Fixup

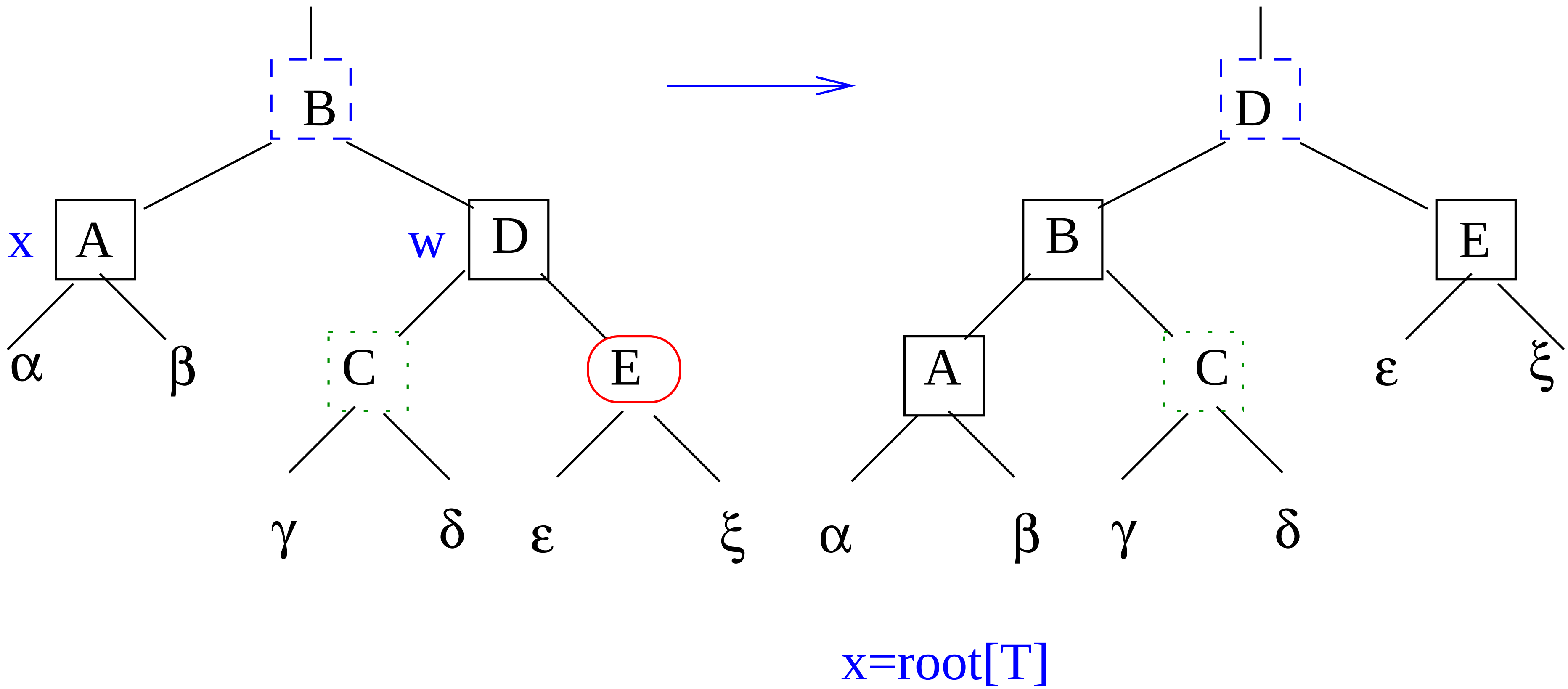
```
...
            if color[right[w]] = BLACK then ▷ prz.3
                color[left[w]] ← BLACK
                color[w] ← RED
                Right-Rotate(T, w)
                w ← right[p[x]]
            color[w] ← color[p[x]] ▷ prz.4
            color[p[x]] ← BLACK
            color[right[w]] ← BLACK
            Left-Rotate(T, p[x])
            x ← root[T] ▷ przerwanie while
    else jak (*) z zamienionymi left i right
color[x] ← BLACK
```

RB-Delete-Fixup: przypadek 3

Prawy syn w jest czarny. C jest czerwony bo tutaj: “nie prawda, że obaj synowie w są czarni”



RB-Delete-Fixup: przypadek 4



RB-Delete-Fixup: analiza czasu

Przypadek 1 – w $O(1)$ sprowadzamy do przypadku 2, po którym x będzie czerwony (przerwanie pętli), lub do przyp. 3 lub 4.
Przypadek 2 – w $O(1)$ przesuwamy x w górę (może zajść $O(h)$ razy).
Przypadek 3 – w $O(1)$ sprowadzamy do przypadku 4.
Przypadek 4 – w $O(1)$ do przewania pętli ($x =$*root*$[T]$)

Wniosek: Czas `RB-Delete-Fixup`: $O(h) = O(\lg n)$

Uwaga: Zawsze ≤ 3 rotacje (w przypadku 2 nie ma rotacji).
Uwaga: po zakończeniu korzeń T jest czarny.

Dynamiczne statystyki pozycyjne

i-ta statystyka pozycyjna – i-ty co do wielkości element
Implementacja: drzewo czerwono-czarne plus dodatkowy atrybut *size*$[x]$ w każdym węźle x ("wzbogacone" drzewo czerwono-czarne).

size$[x]$ – rozmiar poddrzewa o korzeniu x

size$[NIL] = 0$

(*size*[*nil*$[T]$]$= 0$ – dla drzew z wartownikiem)

size$[x]$ $=$*size*[*left*$[x]$] $+$*size*[*right*$[x]$]$+1$

OS-Select

```
OS-Select(x, i)
  r ← size[left[x]] + 1
  if i = r then
    return x
  else
    if i < r then
      return OS-Select(left[x], i)
    else
      return OS-Select(right[x], i − r)
```

Czas: $O(h) = O(\lg n)$

OS-Rank

`OS-Rank(`T`,`x`)`

$r \leftarrow$ *size*`[`*left*$[x]$`]`$+1$

$y \leftarrow x$

`while` $y \neq$ *root*$[T]$ `do`

  `if` $y =$ *right*`[`*p*$[y]$`]` `then`

    $r \leftarrow r +$ *size*`[`*left*`[`*p*$[y]$`]]`$+1$

  $y \leftarrow$ *p*$[y]$

`reutrn` r

Czas: $O(\lg n)$
(każda iteracja `while` – $O(1)$ i y przesunięty bliżej korzenia.)

Wstawianie

Modyfikacje `RB-Insert`(T, x):

- pole *size*$[x]$ zainicjować na 1.
- dodać 1 do pól *size* wierzchołków odwiedzanych w *Tree-Insert*
- zastosować rotacje zmodyfikowane jak poniżej

`Left-Rotate`(T, x)

1 $y \leftarrow$ *right*$[x]$ $\triangleright$ y będzie podnoszony

...

$\triangleright$ dodajemy nowe linie: 13 i 14

13 *size*$[y] \leftarrow$ *size*$[x]$

14 *size*$[x] \leftarrow$ *size*[*left*$[x]$]$+$ *size*[*right*$[x]$]$+1$

Usuwanie

Modyfikacje `RB-Delete`(T, z):

- po odłączeniu y (po linii `12`) przechodzimy po wskaźnikach *p* od y do korzenia odejmując 1 od pól *size* odwiedzonych węzłów.
- w *RB-Delete-Fixup* – zmodyfikowane rotacje (jak przy wstawianiu)

Czasy wstawiania i usuwania: $O(h) = O(\lg n)$

Wzbogacanie drzewa czerwono-czarnego

Tw. Niech *f* – dodatkowe pole, takie że w każdym węźle x wartość *f*$[x]$ można obliczyć na podstawie informacji z węzłów x, *left*$[x]$, *right*$[x]$ (włączając *f*[*left*$[x]$] oraz *f*[*right*$[x]$]). Wtedy można w czasie wstawiania i usuwania aktualizować pola *f* nie przekraczając złożoności czasowej $O(\lg n)$.

D-d. Wstawianie: Modyfikacja `RB-Insert(`T`,`x`)`:

- *f*$[x]$ - można obliczyć w $O(1)$
- po `Tree-Insert(`T`,`x`)` – przejście od x do korzenia, poprawiając pola *f* (czas: $O(\lg n)$)
- w drugiej fazie (linie 2 do 18): Po każdej rotacji pary węzłów x i y, poprawić pola *f*$[x]$, *f*$[y]$ oraz pola *f* na ścieżce od węzłów x, y do korzenia (czas: $O(\lg n)$). W `RB-Insert(`T`,`x`)` są ≤ 2 rotacje.

Wzbogacanie drzewa czerwono-czarnego

Usuwanie: Modyfikacja `RB-Delete(`T`,`z`)`:

- po usunięciu y – poprawki na ścieżce z y do korzenia (czas: $O(\lg n)$)
- jeśli usuwany węzeł z – zastąpiony przez swój następnik y, to poprawki pól *f* na ścieżce z z do korzenia (czas: $O(\lg n)$)
- W `RB-Delete-Fixup` ≤ 3 rotacje – po każdej poprawki w czasie $O(\lg n)$

□

Drzewa przedziałowe

Przedział domknięty: $[t_1, t_2] = \{x \in R : t_1 \leq x \leq t_2\}$ reprezentowany jako obiekt i o polach *low*$[i] = t_1$ i *high*$[i] = t_2$. Przedziały i, i' *zachodzą* na siebie jeśli $i \cap i' \neq \emptyset$ (t.j. *low*$[i]$ $\leq$*high*$[i']$ oraz *low*$[i']$ $\leq$*high*$[i]$.)

Drzewo przedziałowe – drzewo czerwono-czarne, w każdym węźle x – przedział *int*$[x]$ (kluczem x jest *low*[*int*$[x]$]), oraz *max*$[x]$ – maksymalny prawy koniec w poddrzewie x.

max$[x] = \max\{$*high*[*int*$[x]$], *max*[*left*$[x]$], *max*[*right*$[x]$]$\}$

Operacje:

- `Interval-Insert(`T`,`x`)` $O(\lg n)$
- `Interval-Delete(`T`,`x`)` $O(\lg n)$
- `Interval-Search(`T`,`i`)` – zwraca wskaźnik do takiego węzła x, że i oraz *int*$[x]$ zachodzą na siebie lub NIL, jeśli w T nie ma takiego x.

Interval-Search

`Interval-Search`(T,i)
1 $x \leftarrow$ *root*$[T]$
2 **while** $x \neq NIL$ **and** $i \cap$ *int*$[x] = \emptyset$ **do**
3   **if** *left*$[x] \neq NIL$ **and** *max*[*left*$[x]$]$\geq$*low*$[i]$
4   **then** $x \leftarrow$*left*$[x]$
5   **else** $x \leftarrow$*right*$[x]$
6 **return** x

Czas: $O(\lg n)$

Poprawność: Jeśli warunek w wierszu 3 – fałszywy, to nie ma po co iść do lewego poddrzewa x. W p.p. lewe poddrzewo x zawiera przedział zachodzący na i (i tam go znajdziemy) lub zawiera przedział i', taki że *high*$[i]$ $<$*low*$[i']$. W drugim przypadku nie ma po co iść do prawego podrzewa: każdy przedział i'' w prawym poddrzewie x ma (klucz) *low*$[i'']$ $\geq$*low*$[i']$ $>$*high*$[i]$.