Drzewa czerwono-czarne

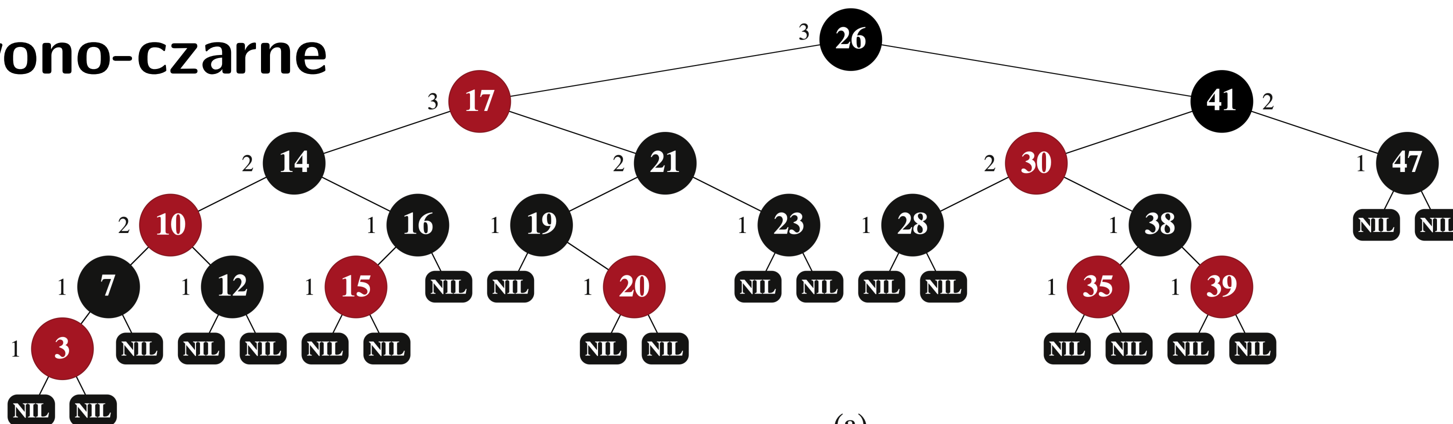
Definicja: **drzewem czerwono-czarnym** nazywamy drzewo binarne drzewo uporządkowane (BST), które spełnia dodatkowe **własności czerwono-czarne**:

1. Każdy węzeł jest albo czerwony albo czarny.
2. Korzeń jest czarny. (ten warunek można pominąć i wszystko działa tak samo)
3. Wszystkie liście mają wartość NIL i są czarne.
4. Jeśli węzeł drzewa jest czerwony, to jego potomki są czarne.
5. Dla każdego węzła drzewa, wszystkie proste ścieżki od tego węzła do jego liści zawierają tę samą liczbę czarnych węzłów.

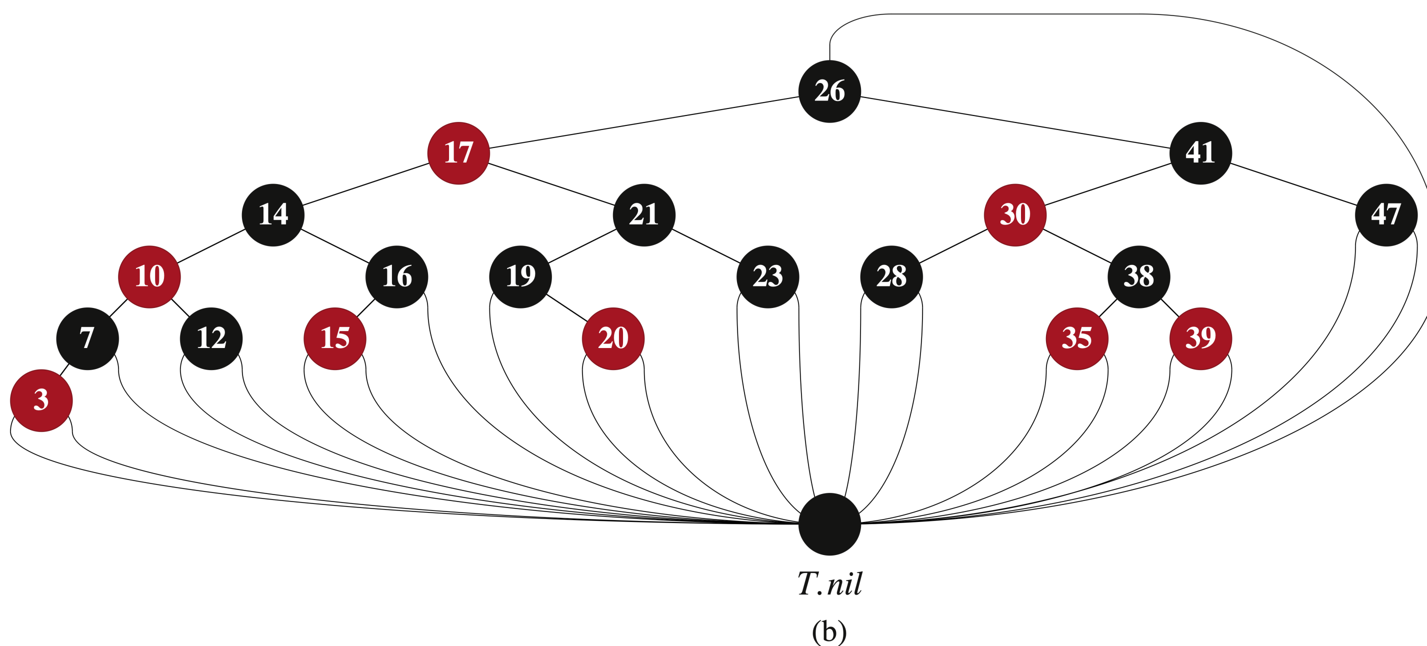
Jak wkrótce zobaczymy, ta definicja, narzucając stałą liczbę węzłów czarnych we wszystkich „sąsiednich" ścieżkach drzewa (własność 5), i dopuszczając tylko pewną limitowaną liczbę węzłów czerwonych (własność 4), gwarantuje, że takie drzewa będą w przybliżeniu zrównoważone. Bardziej precyzyjnie, można udowodnić, że wysokość drzewa czerwono-czarnego z n kluczami może wynosić maksymalnie $2\log_2(n+1)$, a zatem jest $O(\log_2 n)$.

To oznacza, że nawet w najgorszym przypadku drzewa czerwono-czarne są asymptotycznie równoważne drzewom w pełni zrównoważonym.

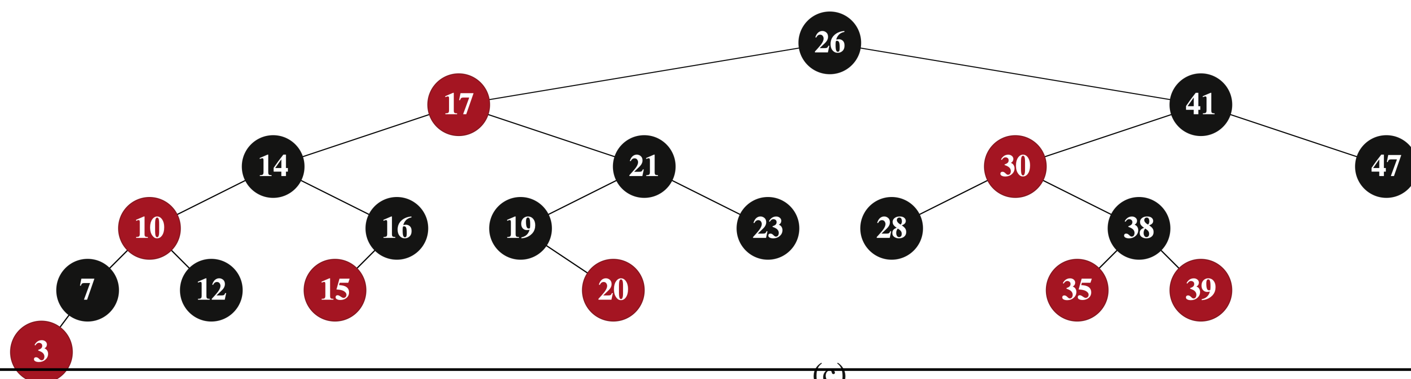
Drzewa czerwono-czarne — przykład



(a)



(b)



(c)

Drzewa czerwono-czarne — reprezentacja

W reprezentacji drzew czerwono-czarnych przydatne okazuje się wykorzystanie wartownika, który jest obiektem takiego samego typu jak wszystkie węzły wewnętrzne drzewa, i który będzie wykorzystany do reprezentacji wszystkich pustych liści drzewa. To znaczy, wszystkie puste wskaźniki NIL w węzłach drzewa zastępowane są wskaźnikiem do tego wartownika.

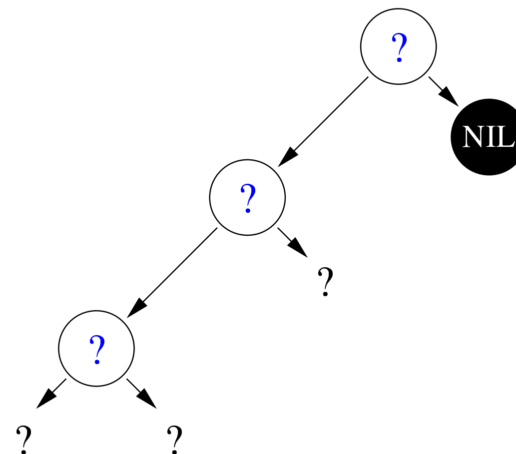
Ta konwencja pozwala traktować wszystkie węzły drzewa tak jakby miały niepustych potomków, co upraszcza kod niektórych procedur. Wartownik pełni również rolę rodzica korzenia drzewa.

Technicznie, wskaźnik do wartownika jest pamiętany jako atrybut nil obiektu T ($T.nil$) reprezentującego całe drzewo. Jednocześnie atrybut $root$ tego obiektu zawiera wskaźnik do korzenia drzewa. Wartości atrybutów p, $left$, $right$, i key wartownika nie mają znaczenia, jednak wartownik musi być węzłem czarnym, zgodnie z własnością 3 drzewa czerwono-czarnego.

Drzewa czerwono-czarne — wypełnienie

Intuicyjnie, sens drzew czerwono-czarnych można wyjaśnić w oparciu o własność piątą, wymagającą identycznej długości „czarnej" wszystkich ścieżek drzewa od korzenia do liści. Ponieważ liczba „dodatkowych" węzłów czerwonych jest ograniczona na mocy własności czwartej — tylko co drugi poziom drzewa mogą zajmować węzły czerwone — zatem całkowite długości tych ścieżek mogą różnić się najwyżej dwukrotnie. Jednocześnie zarówno korzeń, jak i liście drzewa, zawierające klucze o wartości NIL, są czarne, zatem wszystkie możliwe ścieżki w drzewie podlegają temu wymaganiu o równej wysokości (czarnej).

To oznacza, że niedopuszczalne jest, aby węzeł drzewa czerwono-czarnego posiadał niepustego (niebędącego liściem) wnuka, obojętnie jakiego koloru, jeśli nie posiada dwóch niepustych potomków.



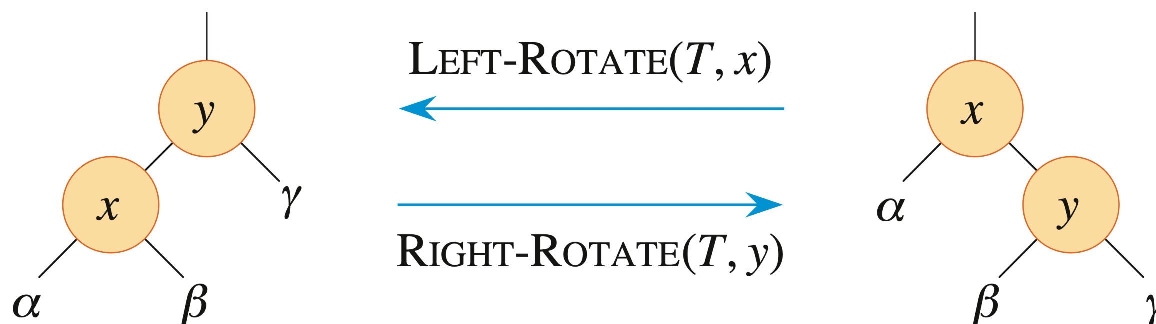
Dzięki temu mamy pewność, że drzewa czerwono-czarne są względnie zapełnione w swojej strukturze, z dokładnością do maksymalnie dwukrotnej różnicy wysokości wewnątrz drzewa.
Pytanie jednak, czy istnieją efektywne procedury budowania takich drzew, to znaczy dodawania i usuwania węzłów?

Rotacje na binarnych drzewach przeszukiwań

Drzewa czerwono-czarne mają pożądaną własność częściowego zrównoważenia, asymptotycznie równoważną drzewom w pełni zrównoważonym. Powstaje jednak pytanie czy istnieją efektywne procedury budowy i obsługi takich drzew, to znaczy, dodawania i usuwania z nich kluczy.

Jak widzieliśmy wcześniej, zwykłe procedury TREE-INSERT i TREE-DELETE działają na drzewach BST, a więc i na drzewach czerwono-czarnych. Jednak modyfikują one strukturę drzewa, więc wynikiem ich działania na drzewie spełniającym wszystkie własności czerwono-czarne może być drzewo, które ich już nie spełnia.

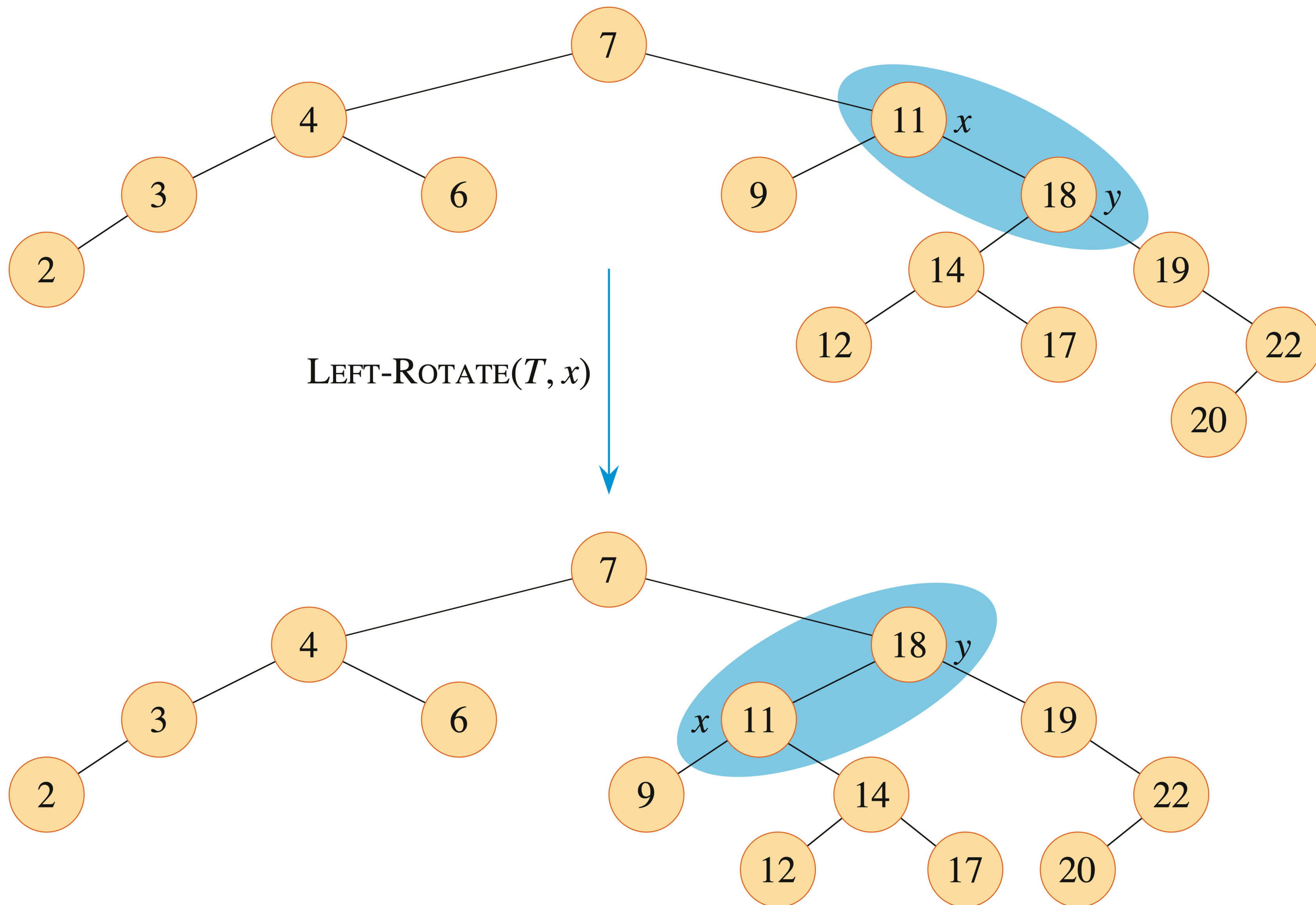
Kluczem do wielu operacji przekształcania drzew BST będą następujące operacje **rotacji** zmieniające drzewo BST w inne drzewo, zawierające te same klucze, i nadal tak samo uporządkowane, ale o innej strukturze:



```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left        // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil       // if y's left subtree is not empty ...
 4       y.left.p = x       // ... then x becomes the parent of the subtree's root
 5  y.p = x.p               // x's parent becomes y's parent
 6  if x.p == T.nil         // if x was the root ...
 7       T.root = y         // ... then y becomes the root
 8  elseif x == x.p.left    // otherwise, if x was a left child ...
 9       x.p.left = y       // ... then y becomes a left child
10  else x.p.right = y      // otherwise, x was a right child, and now y is
11  y.left = x              // make x become y's left child
12  x.p = y
```

Zauważmy, że w trakcie rotacji zmianie ulega jedynie struktura drzewa BST, z zachowaniem uporządkowania węzłów. Zmieniają się jedynie wskaźniki $left, right, p$, natomiast procedura nie ingeruje w kolor węzłów. W związku z tym może ona naruszyć własności czerwono-czarne.

Rotacje na BST — przykłady



Dodawanie węzłów do drzew czerwono-czarnych

RB-INSERT(T, z)

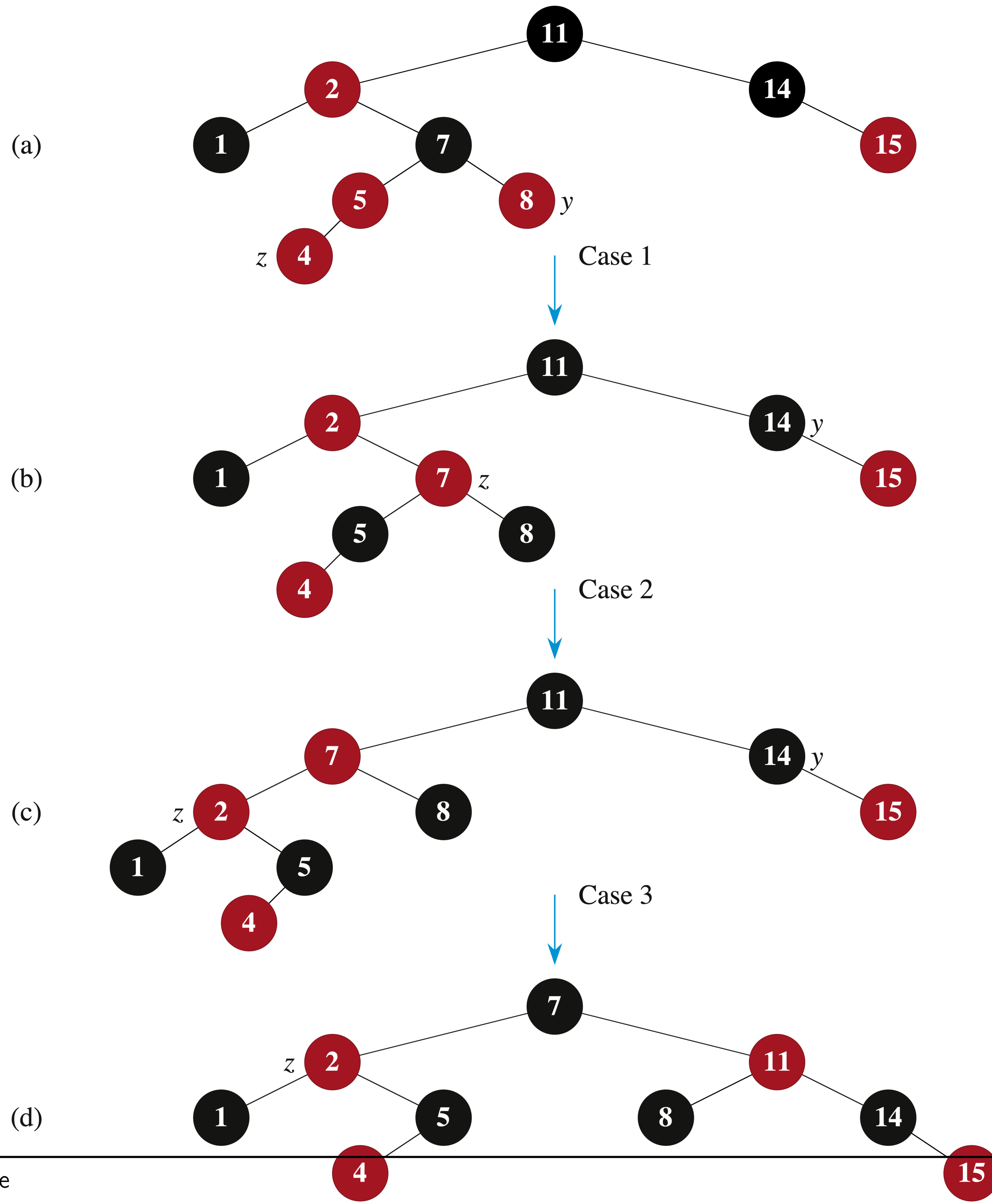
```
 1  x = T.root                     // node being compared with z
 2  y = T.nil                      // y will be parent of z
 3  while x ≠ T.nil                // descend until reaching the sentinel
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y                        // found the location—insert z with parent y
 9  if y == T.nil
10      T.root = z                 // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil                 // both of z's children are the sentinel
15  z.right = T.nil
16  z.color = RED                  // the new node starts out red
17  RB-INSERT-FIXUP(T, z)          // correct any violations of red-black properties
```

Ta procedura różni się od podstawowej TREE-INSERT tylko dwoma ostatnimi wierszami: nowy węzeł jest czerwony i należy przywrócić własności czerwono-czarne.

RB-INSERT-FIXUP(T, z)

```
1  while z.p.color == RED
2      if z.p == z.p.p.left            // is z's parent a left child?
3          y = z.p.p.right             // y is z's uncle
4          if y.color == RED           // are z's parent and uncle both red?
5              z.p.color = BLACK       ⎫
6              y.color = BLACK         ⎬ case 1
7              z.p.p.color = RED       ⎪
8              z = z.p.p               ⎭
9          else
10             if z == z.p.right
11                 z = z.p             ⎫ case 2
12                 LEFT-ROTATE(T, z)   ⎭
13             z.p.color = BLACK       ⎫
14             z.p.p.color = RED       ⎬ case 3
15             RIGHT-ROTATE(T, z.p.p)  ⎭
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

Dodawanie węzłów do drzew czerwono-czarnych — przykład



Usuwanie węzłów z drzew czerwono-czarnych

Operacja usuwania węzła z drzewa czerwono-czarnego, pomimo iż zajmuje $O(\log n)$ operacji, jest bardziej skomplikowana niż dodawanie węzła, analogicznie, jak w przypadku ogólnych drzew BST.

Operacja usuwania węzła RB-DELETE jest wzorowana na zwykłej procedurze TREE-DELETE i wykorzystuje poniższy, zaadaptowany dla drzew czerwono-czarnych, wariant procedury TRANSPLANT:

RB-TRANSPLANT(T, u, v)

1 **if** $u.p == T.nil$
2 $\quad T.root = v$
3 **elseif** $u == u.p.left$
4 $\quad u.p.left = v$
5 **else** $u.p.right = v$
6 $v.p = u.p$

RB-TRANSPLANT w zasadzie nie różni się od podstawowej TRANSPLANT. Operacja w wierszu 6 może być wykonana bezwarunkowo, ponieważ ze względu na wykorzystanie strażnika $T.nil$ przypisanie wartości w pustym liściu nie generuje wyjątku.

RB-DELETE(T, z)

```
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)         // replace z by its right child
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)          // replace z by its left child
 9  else y = TREE-MINIMUM(z.right)           // y is z's successor
10      y-original-color = y.color
11      x = y.right
12      if y ≠ z.right                       // is y farther down the tree?
13          RB-TRANSPLANT(T, y, y.right)     // replace y by its right child
14          y.right = z.right                // z's right child becomes
15          y.right.p = y                    // y's right child
16      else x.p = y                         // in case x is T.nil
17      RB-TRANSPLANT(T, z, y)               // replace z by its successor y
18      y.left = z.left                      // and give z's left child to y,
19      y.left.p = y                         //     which had no left child
20      y.color = z.color
21  if y-original-color == BLACK    // if any red-black violations occurred,
22      RB-DELETE-FIXUP(T, x)       //     correct them
```

RB-DELETE jest wzorowana na TREE-DELETE i różni się od tamtej kilkoma szczegółami, mającymi na celu zadbanie o zachowanie własności czerwono-czarnych. Konkretnie, mamy w niej dodatkowe wiersze: 1, 2, 4, 7, 10, 11, 16, 20, 21, i 22.

Zmienna y ma tu wartość albo usuwanego węzła z gdy ma on tylko jednego potomka (wiersz 1), lub następnika z (który również ma tylko jednego potomka), gdy sam z ma dwóch niepustych potomków (wiersz 9). W każdym przypadku zapamiętywany jest oryginalny kolor węzła y (wiersze 2 i 10).

Procedura zapamiętuje również jako x węzeł, który przechodzi na dotychczasowe miejsce y'a. To może być jedyny potomek y'a (wiersze 4 i 7), albo prawy potomek y'a gdy jest on następnikiem z'a i wiadomo, że nie ma lewego potomka (wiersz 11; w tym przypadku prawy potomek może być pusty, czyli $T.nil$).

Nieco szczególną rolę odgrywa wiersz 16. Ustawia on y jako rodzica x'a, ale x jest normalnie potomkiem y'a, więc ma ten atrybut ustawiony prawidłowo. Jednak wyjątkiem jest sytuacja gdy x jest pusty (równy $T.nil$), i oczywiście pusty węzeł ma w stosowanej tu reprezentacji wielu rodziców, ale nie są oni jawnie tam wpisani. Ale tu sytuacja jest szczególna, bo niżej wywołana procedura TREE-DELETE-FIXUP zakłada, że rodzicem x'a w każdym przypadku będzie y, więc ta wartość jest tu wpisywana do wartownika $T.nil$ korzystając z faktu, że nigdzie indziej wartość tego atrybutu w wartowniku nie jest wykorzystywana.

Jako uzupełnienie transplantacji węzła y w miejsce z (wiersz 17), y otrzymuje kolor z'a (wiersz 20).

Na koniec, można zauważyć (patrz CLRS strona 329), że w przypadku gdy y był oryginalnie czerwony, to powyższe operacje nie naruszyły własności czerwono-czarnych. W przeciwnym wypadku, wywoływana jest procedura Tree-Delete-Fixup, która te własności przywraca.

RB-DELETE-FIXUP(T, x)

```
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left                          // is x a left child?
 3          w = x.p.right                         // w is x's sibling
 4          if w.color == RED
 5              w.color = BLACK                   ⎫
 6              x.p.color = RED                   ⎬ case 1
 7              LEFT-ROTATE(T, x.p)               ⎪
 8              w = x.p.right                     ⎭
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                     ⎫ case 2
11              x = x.p                           ⎭
12          else
13              if w.right.color == BLACK
14                  w.left.color = BLACK          ⎫
15                  w.color = RED                 ⎬ case 3
16                  RIGHT-ROTATE(T, w)            ⎪
17                  w = x.p.right                 ⎭
18              w.color = x.p.color               ⎫
19              x.p.color = BLACK                 ⎪
20              w.right.color = BLACK             ⎬ case 4
21              LEFT-ROTATE(T, x.p)               ⎪
22              x = T.root                        ⎭
```

```
23          else // same as lines 3–22, but with "right" and "left" exchanged
24              w = x.p.left
25              if w.color == RED
26                  w.color = BLACK
27                  x.p.color = RED
28                  RIGHT-ROTATE(T, x.p)
29                  w = x.p.left
30              if w.right.color == BLACK and w.left.color == BLACK
31                  w.color = RED
32                  x = x.p
33              else
34                  if w.left.color == BLACK
35                      w.right.color = BLACK
36                      w.color = RED
37                      LEFT-ROTATE(T, w)
38                      w = x.p.left
39                  w.color = x.p.color
40                  x.p.color = BLACK
41                  w.left.color = BLACK
42                  RIGHT-ROTATE(T, x.p)
43                  x = T.root
44  x.color = BLACK
```

Drzewa czerwono-czarne — podsumowanie

Istnieje wiele rodzajów drzew częściowo zrównoważonych, przybliżających ideę pełnego zrównoważenia drzewa przez zagwarantowanie jego asymptotycznie logarytmicznej wysokości w przypadku najgorszym. Drzewa czerwono-czarne są jednym z nich.

Definicja drzew czerwono-czarnych jest skomplikowana, i takie są również procedury ich budowy (dodawania i usuwania węzła), wymagające korekt rozróżniających 6 przypadków przy dodawaniu i 8 przypadków przy usuwaniu węzła.

Na przykład, można porównać drzewa czerwono-czarne z innym rodzajem drzew częściowo zrównoważonych — drzewami AVL. Ich definicja jest dużo prostsza — **w drzewie AVL** wysokość dwóch poddrzew dowolnego węzła drzewa nie może różnić się więcej niż o 1. Drzewa AVL są prostsze intuicyjnie, oraz prostsze są operacje ich budowy, również wykorzystujące rotacje. Jednak ta prosta definicja drzew AVL jest bardziej restrykcyjna, i wysokość drzewa AVL w najgorszym przypadku nie przekracza wartości $1.44\log_2(n)$ w porównaniu z $2\log_2(n)$ dla drzew czerwono-czarnych.

Oznacza to nieco lepsze czasy wyszukiwania elementu, ale jednocześnie konieczność częstszych korekt drzew AVL przy dodawaniu i usuwaniu węzła. Drzewa czerwono-czarne pozwalają na bardziej liberalne odstępstwa od pełnego zrównoważenia, ale ich koncepcja i procedury budowy są bardziej skomplikowane.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Ile wynosi największa możliwa liczba węzłów wewnętrznych w drzewie czerwono-czarnym o czarnej wysokości k? Jaka jest najmniejsza taka liczba?

2. Niech a, b, i c będą dowolnymi węzłami w poddrzewach odpowiednio: α, β, i γ prawego drzewa na rysunku na stronie 5. Jak zmienią się głębokości węzłów a, b, i c gdy na tym drzewie zostanie wykonana lewa rotacja węzła x?

3. Zbuduj drzewo czerwono-czarne przez dodawanie kolejno kluczy 41, 38, 31, 12, 19, 8 procedurą RB-INSERT do początkowo pustego drzewa.

4. Dla drzewa czerwono-czarnego zbudowanego w poprzednim ćwiczeniu, wykonaj kolejno operacje usuwania węzłów: 8, 12, 19, 31, 38, 41 procedurą RB-DELETE pokazując stan drzewa po każdej pojedynczej operacji usuwania węzła.

5. Załóżmy, że węzeł x został dodany do drzewa czerwono-czarnego procedurą RB-INSERT, oraz następnie natychmiast usunięty procedurą RB-DELETE. Czy otrzymane drzewo czerwono-czarne zawsze będzie identyczne do początkowego? Uzasadnij odpowiedź.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 13.

2. Applety ilustrujące działanie drzew czerwono-czarnych i drzew AVL z Uniwersytetu San Francisco (prof. David Galles):
`https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`
`https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`