# Analiza algorytmów

prowadzący: Michał Hańćkowiak

# Plan wykładu.

1. Algorytmy na 1- procesorze (sekwencyjne):
   - słowniki; przy pomocy drzew BST, RB, ???
   - *algorytmy grafowe*; problem MST (=Minimum Spanning Tree), algorytm Prima dla MST używający kolejki priorytetowej (z kopca)
   - kopce; binarne, dwumianowe, Fibonacciego
   - analiza kosztu zamortyzowanego (na użytek kopca Fib.)
   - ???
2. Algorytmy na $n$- procesorach (rozproszone, z przesyłaniem komunikatów):
   - problemy grafowe w sieci komunikacyjnej...
   - MIS w drzewie ukorzenionym i w grafie stałego stopnia
   - MWIS, MM, MDS, MCDS w grafach planarnych

# Literatura.

- Cormen, Leiserson, Rivest "Wprowadzenie do algorytmów"
- materiały do wykładu AAL260 na stronie http://main2.amu.edu.pl/~mhanckow

# Modele obliczeń.

- 1 procesor, model **sekwencyjny**, maszyna RAM
  - RAM (=Random Access Machine), maszyna ze swobodnym dostępem do komórek pamięci
  - jest to matematyczny/uproszczony model maszyny z 1 procesorem
  - program to ciąg rozkazów (typu odczytanie komórki pamięci) przypominających rozkazy prawdziwego procesora...
- $n$- procesorów, model **równoległy**, maszyna PRAM
  - Parallel RAM; model maszyny składającej się z procesorów mających dostęp do wspólnej pamięci
  - procesory działają "synchronicznie" tj wykonują równocześnie/ równolegle po jednym rozkazie
  - wyróżnia się rodzaje PRAM: CRCW, CREW (C-Concurrent, E-Exclusive, R-Read, W-Write)
- $n$- procesorów; model **rozproszony** z przesyłaniem komunikatów
  - matematyczny model sieci komputerowej, która jest modelowana jako graf
  - dwa podstawowe rodzaje: synchroniczny i asynchroniczny; w synchronicznym wierzchołki sieci działaja w tzw *rundach*: każdy wierz wysyła komunikaty do sąsiadów, odbiera komunikaty od sąsiadów, i wykonuje obliczenia lokalne

# Złożonośc algorytmów sekwencyjnych.

$D_n$ - zbiór danych wejścowych rozmiaru $n$
$t(d)$ - liczba **operacji dominujących** wykonywanych przez algorytm po uruchomieniu dla danych wejściowych $d \in D_n$
$X_n$ - zmienna losowa, której wartością jest $t(d)$ dla $d \in D_n$

**def:** pesymistyczna złożoność czasowa algorytmu (ang. worstcase complexity) to $W(n) = sup\{t(d) : d \in D_n\}$

**def:** pesymistyczna wrażliwość algorytmu to $\Delta(n) = sup\{t(d_1) - t(d_2) : d_1, d_2 \in D_n\}$

**def:** oczekiwana złożoność czasowa algorytmu to $A(n) = E(X_n)$

**def:** oczekiwana wrażliwość algorytmu to $\delta(n) = \sqrt{Var(X_n)}$ czyli odchylenie standardowe $X_n$

**def:** złożoność pamięciowa - ile pamięci wymaga algorytm

Algorytmy sekwencyjne.

# Słownik.

Słownik S to "cos" na czym można wykonywać następujące operacje:

- insert(S, element) - wstawianie elementu od słownika S; element= (klucz,wartość)
- delete(S, element) - usunięcie elementu
- element= search(S, klucz) - znalezienie elementu o zadanym kluczu

Słownik jest rzeczą bardzo przydatną w praktyce programistycznej ...

Słowniki można implementować przy pomocy:
drzew BST, RB, przez haszowanie, w posortowanej tablicy

Interesuje nas złożoność czasowa operacji insert/delete/search...

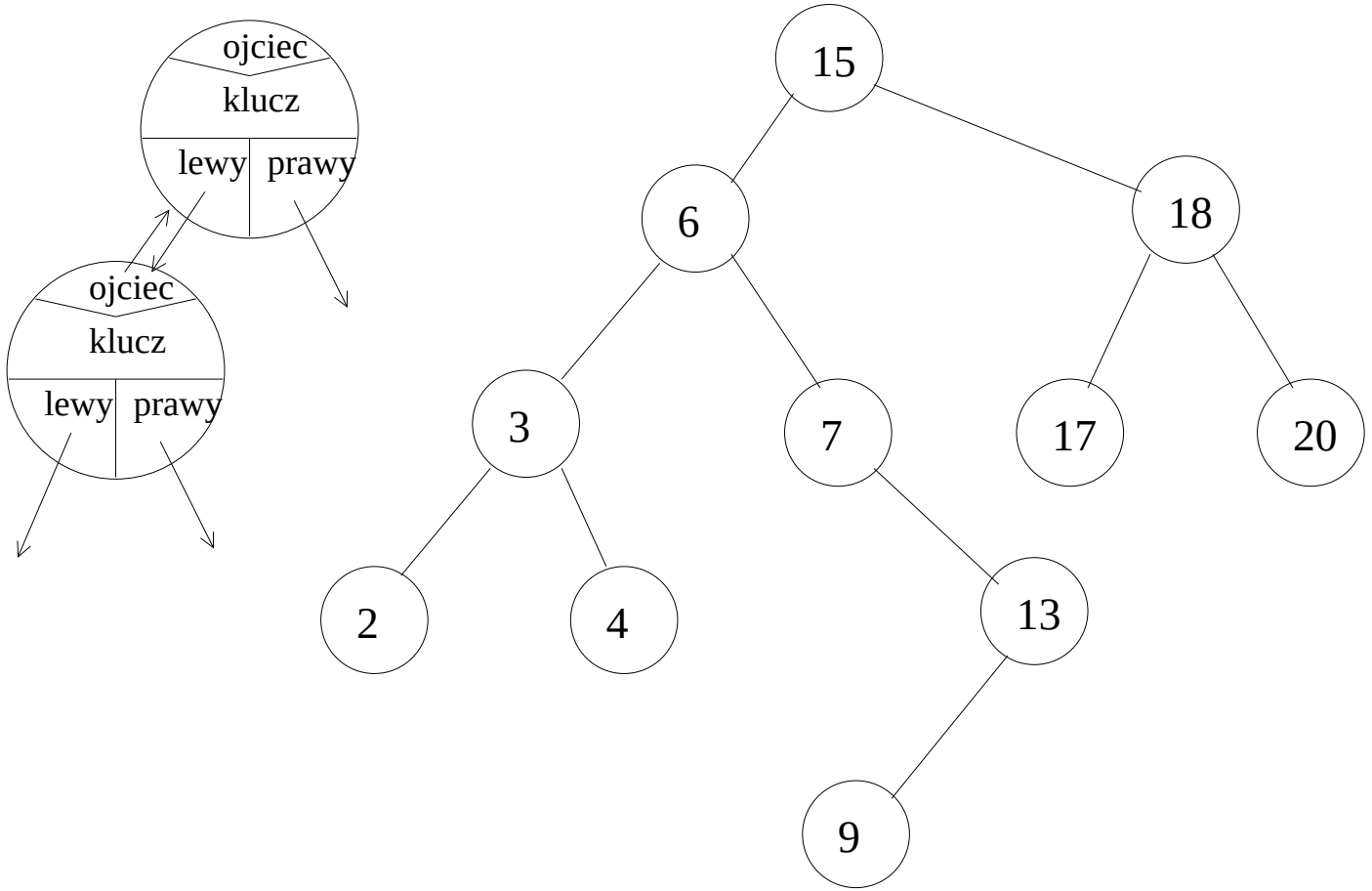
# Drzewa BST (ang. Binary Search Tree).

1. drzewo BST jest drzewem binarnym (niekoniecznie pełnym);
   każdy wierzchołek $v$ drzewa jest strukturą z polami: $v.lewy$, $v.prawy$, $v.ojciec$, $v.klucz$ (dla uproszczenia nie umieszczamy pola wartość)

2. **własność drzewa BST**:
   dla każdego wierzchołka $v$ drzewa:
   wierzchołki lewego poddrzewa maja klucze $\leq v.klucz$
   wierzchołki prawego poddrzewa maja klucze $\geq v.klucz$

Własność drzewa BST: przechodząc je metodą *InOrder* (i wyswietlając klucze wierzchołków) otrzymamy ciąg posortowany rosnąco!

Przykład drzewa BST:



# Operacje na drzewie BST.

**BST_Insert(T,z)**: Wstawianie elementu $z$ do drzewa BST $T$;
$T$ to struktura posiadająca pole $T.root$ wskazujące na korzeń drzewa;
$z$ ma wyzerowane pola za wyjątkiem $z.klucz$;
wstawiamy tak aby nie zepsuć własności BST!!!

```
proc BST_Insert(T,z)
  y:= nil
  x:= T.root
  while x != nil do
    y:= x
    if z.klucz < x.klucz then x:= x.lewy else x:= x.prawy
  z.ojciec:= y
  if y == nil then
    T.root:= z
  else
    if z.klucz < y.klucz then y.left:= z else y.right:= z
```

**BST_Search(x,k)**: Szukanie elementu z kluczem $k$ w drzewie o korzeniu $x$; jeśli klucz nie występuje to proc zwraca nil

```
proc BST_Search(x,k)
  if x == nil or k== x.klucz then return x
  if k < x.klucz then
    return BST_Search(x.lewy, k)
  else
    return BST_Search(x.prawy, k)

proc BST_Search(x,k)
  while x != nil and k != x.klucz do
    if k < x.klucz then x:= x.lewy else x:= x.prawy
  return x
```

**BST_Maximum(x)**: Wystarczy iść po drzewie od korzenia $x$ do liścia wybierając zawsze prawe poddrzewo (iść tak długo jak się da).

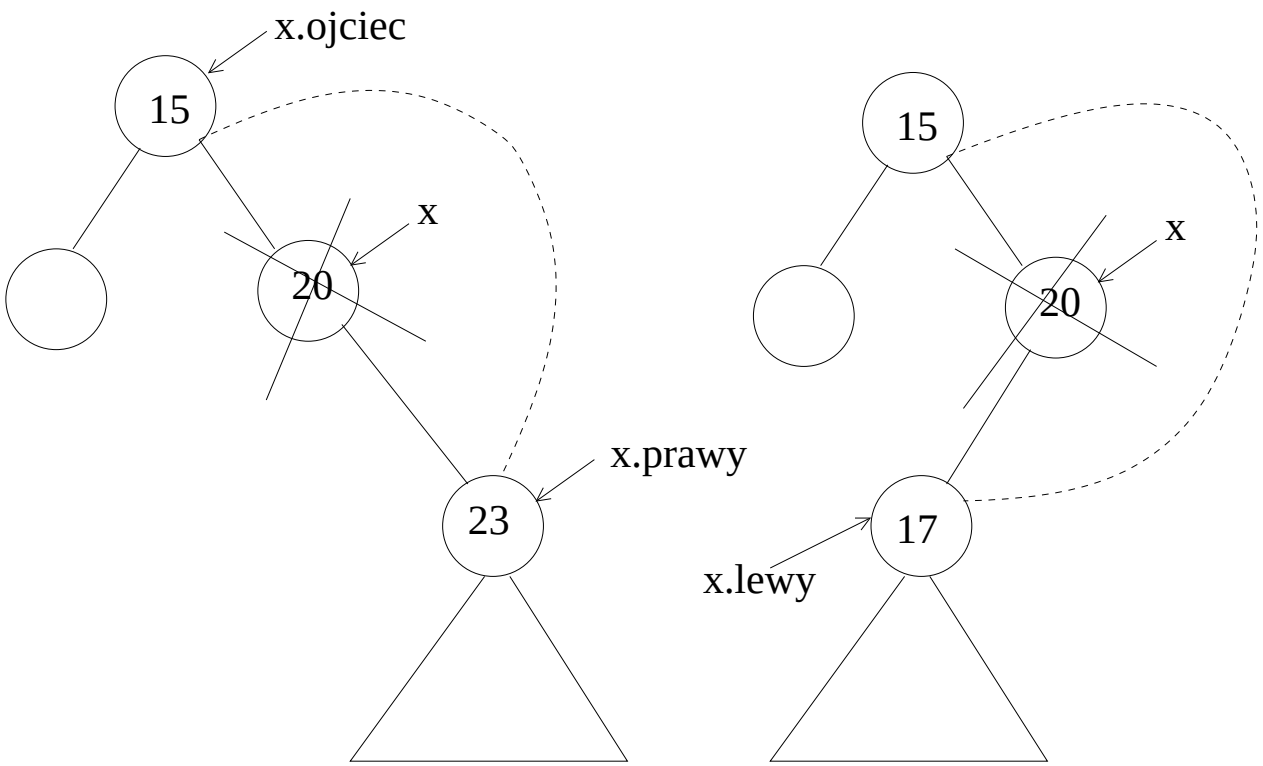
**BST_Minimum(x)**: Wystarczy iść po drzewie od korzenia $x$ do liścia wybierając zawsze lewe poddrzewo.

**BST_Nastepnik(x)**: Szukamy następnika wierzchołka $x$. Dwa przypadki:
1) jeśli wierzchołek $x$ posiada prawe poddrzewo, to wtedy zwracamy minimalny element w prawym poddrzewie
2) w przeciwnym wypadku szukamy wierzchołka takiego, że największym kluczem jego lewego poddrzewa jest $x.klucz$; innymi słowy idziemy od wierzchołka $x$ w kierunku korzenia tak długo aż napotkamy wierzchołek, który jest lewym synem swojego rodzica; wartość rodzica jest właśnie poszukiwanym następnikiem (na rysunku następnikiem 13 jest 15, następnikiem 6 jest 7)

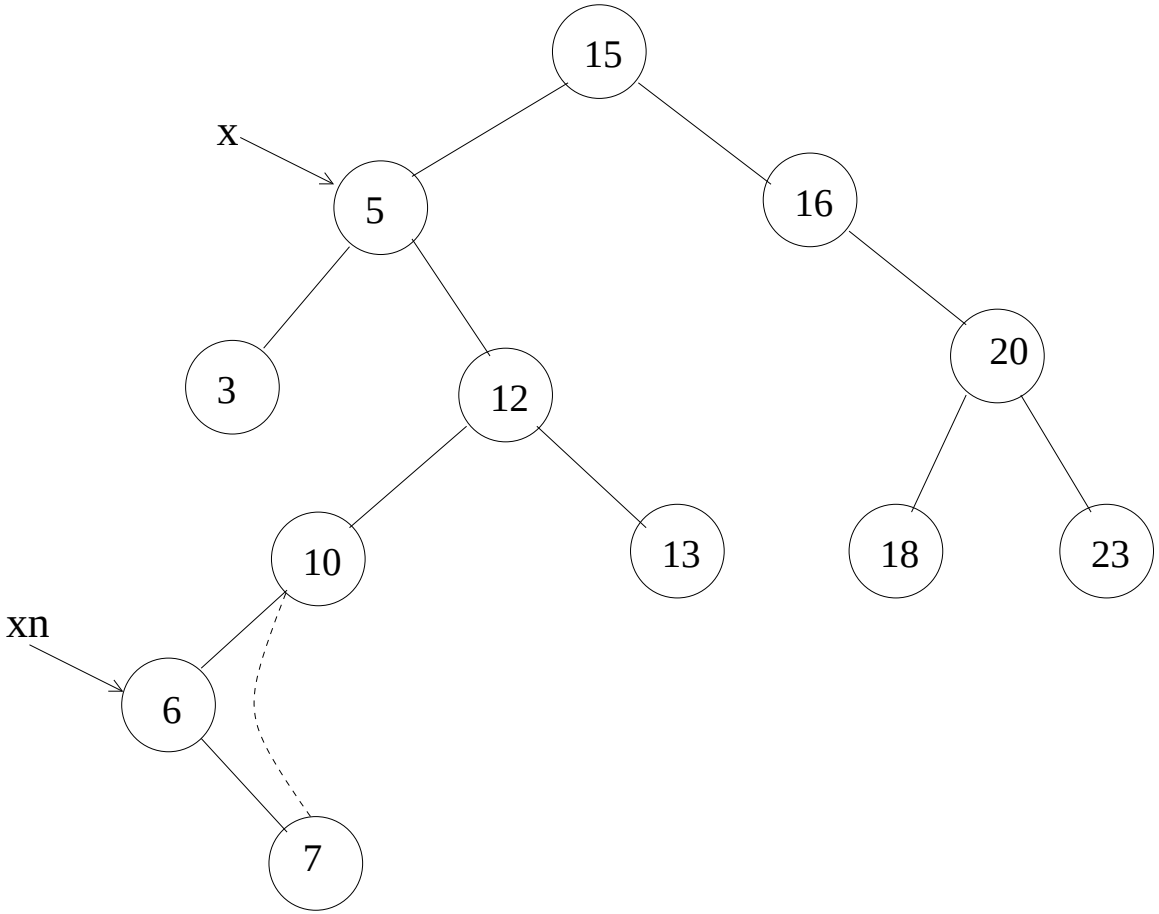
**BST_Usun(T,x)**: Usuwamy wierzchołek $x$. Trzy przypadki:
1) $x$ jest liściem drzewa - wtedy zwyczajnie go usuwamy
2) $x$ ma tylko jednego potomka - wtedy dodajemy odpowiednie połączenie między wierzchołkiem $x.ojciec$ i $x.lewy$ (lub $x.prawy$), a $x$ usuwamy (patrz rysunek)
3) $x$ ma dwóch potomków - wtedy znajdujemy następnik $x$ i oznaczmy go przez $xn$; o $xn$ wiemy, że nie ma lewego potomka; dodajemy odpowiednie połączenie między wierzchołkami $xn.ojciec$ i $xn.prawy$, następnie przypisujemy $x.klucz := xn.klucz$ i usuwamy $xn$
(patrz rysunek)

Dlaczego 3 przypadek nie psuje własności BST w wierz $x$ ???

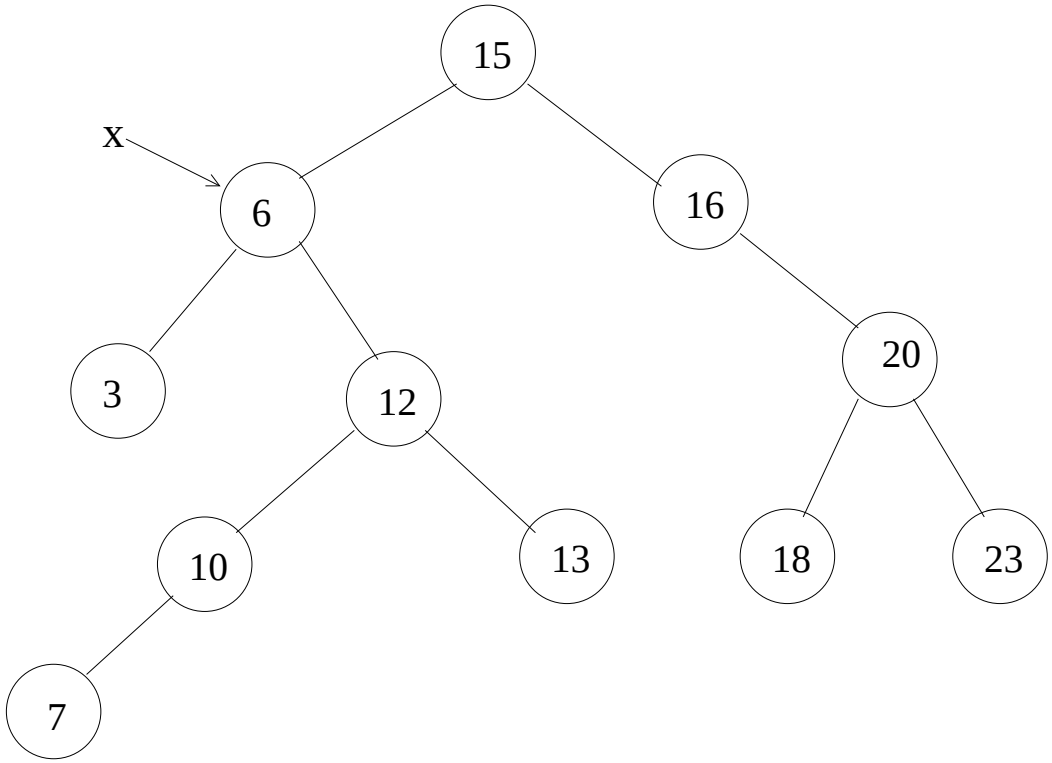
# BST_Usun(T,x)/ przypadek (2)



# BST_Usun(T,x)/ przypadek (3)



BST_Usun(T,x)/ przypadek (3) c.d.



# Złożoność operacji na drzewie BST.

Pesymistyczną złożoność czasową wszystkich operacji na drzewie BST można oszacować przez $O(h)$, gdzie $h$ jest wysokością drzewa BST.

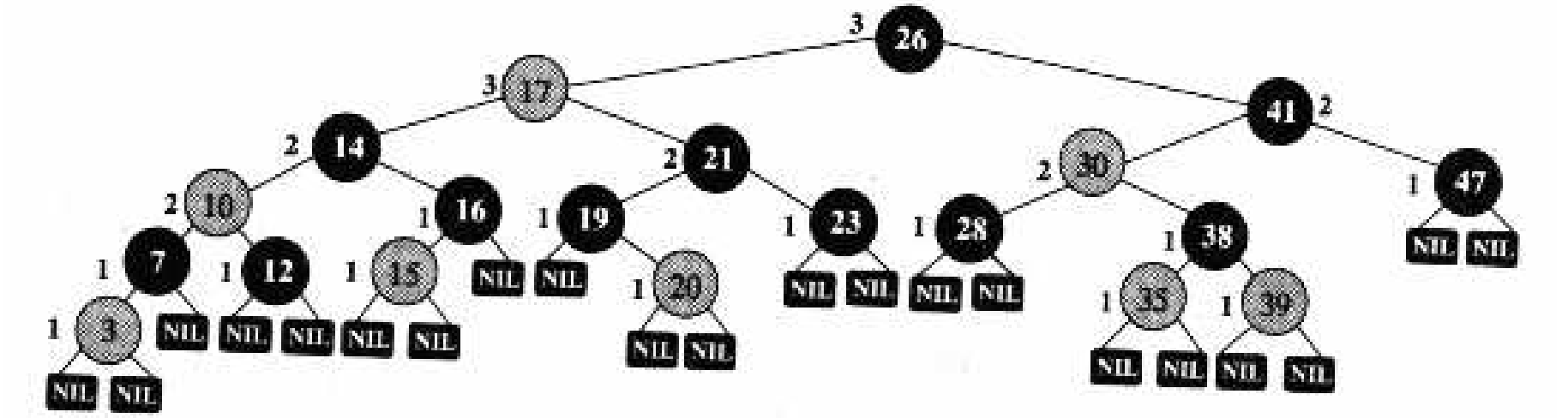
Niestety drzewo BST może mieć wysokość $O(n)$, gdzie $n$ to liczba wierzchołków drzewa.

Można jednak udowodnić następujące Twierdzenie (Cormen str 299):

**Twierdzenie 13.6** Oczekiwana wysokość drzewa BST zbudowanego przy pomocy samych operacji BST_Insert na losowej permutacji liczb ze zbioru $\{1, \ldots, n\}$ wynosi $O(\log n)$. (Wszystkie permutacje równo prawdopodobne!)

# Drzewa czerwono-czarne (RB, ang. Red Black).

Jest to specjalny rodzaj drzew BST, w którym mamy gwarancję, że drzewo jest w przybliżeniu "zrównoważone" mimo wykonywania operacji modyfikujących Insert/Delete ...



# Drzewa czerwono-czarne (RB).

**Właściwości RB**:

1. każdy wierz. jest czerwony(R) lub czarny(B)
2. każdy liść (wirtualny liść "nil") jest B
3. jeśli wierz. jest R to ma synów B
4. dla każdego wierz. $v$ w drzewie: każda ścieżka $v$-liść (skierowana) ma tyle samo wierz B

Drzewo RB musi zachowywać te właściwości pomimo wykonywania na nim operacji Insert/Delete...

Z właściwości drzewa RB wynika:

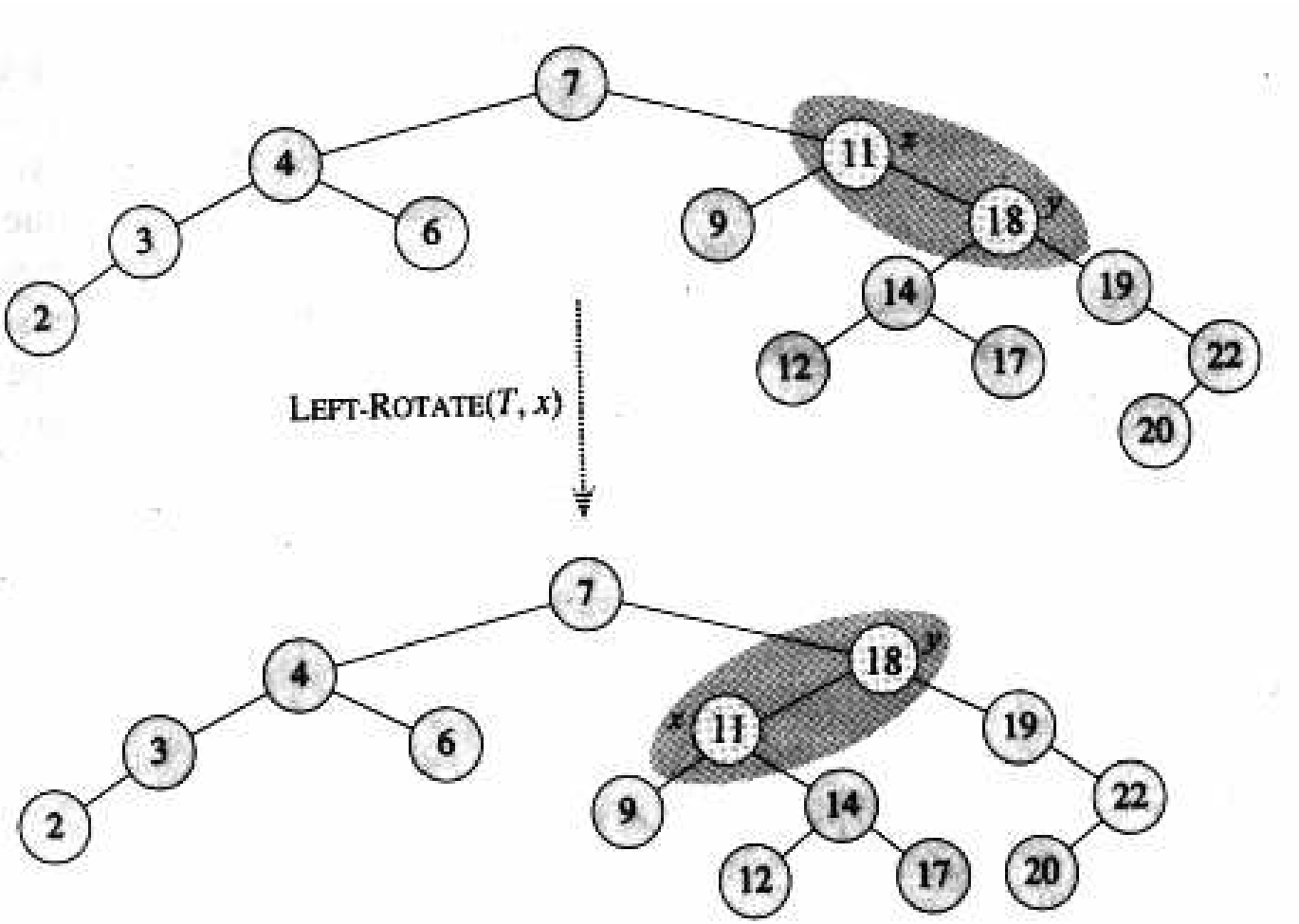
**Lemat 14.1** Drzewo RB o $n$-wierz. wewnętrznych ma wysokość $\leq 2\log_2(n+1)$

**Dowód** lematu 14.1 ...

# Operacje rotacji na drzewie RB.

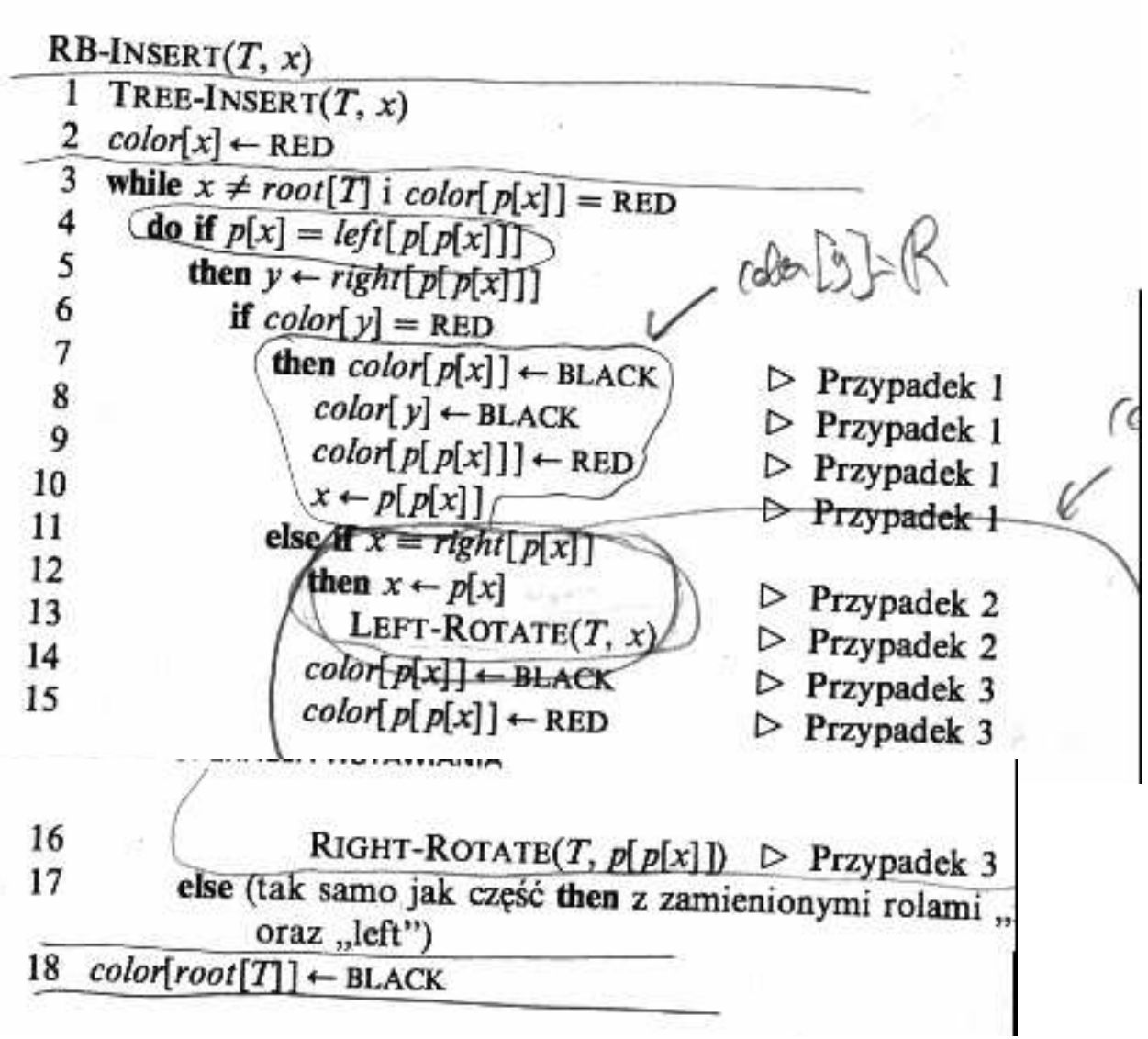
Operacje te zachowują własność BST w drzewie RB...

RB_RotateLeft(T, x)



# Operacje Insert/Delete na drzewie RB.

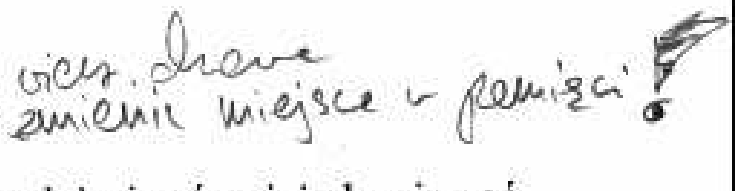
# RB_Insert



```
RB-INSERT(T, x)
 1  TREE-INSERT(T, x)
 2  color[x] ← RED
 3  while x ≠ root[T] i color[p[x]] = RED
 4    do if p[x] = left[p[p[x]]]
 5       then y ← right[p[p[x]]]
 6            if color[y] = RED
 7               then color[p[x]] ← BLACK                ▷ Przypadek 1
 8                    color[y] ← BLACK                   ▷ Przypadek 1
 9                    color[p[p[x]]] ← RED               ▷ Przypadek 1
10                    x ← p[p[x]]                        ▷ Przypadek 1
11               else if x = right[p[x]]
12                       then x ← p[x]                   ▷ Przypadek 2
13                            LEFT-ROTATE(T, x)          ▷ Przypadek 2
14                    color[p[x]] ← BLACK                ▷ Przypadek 3
15                    color[p[p[x]]] ← RED               ▷ Przypadek 3
16               RIGHT-ROTATE(T, p[p[x]])                ▷ Przypadek 3
17       else (tak samo jak część then z zamienionymi rolami „
               oraz „left")
18  color[root[T]] ← BLACK
```

# RB_Delete

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] lub right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11        then left[p[y]] ← x
12        else right[p[y]] ← x
13  if y ≠ z
14  then key[z] ← key[y]
15        ▷ Jeśli y ma inne pola, to należy je również skopiować.
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```



```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] i color[x] = BLACK
 2     do if x = left[p[x]]
 3        then w ← right[p[x]]
 4           if color[w] = RED
 5              then color[w] ← BLACK                          ▷ Przypadek 1
 6                   color[p[x]] ← RED                         ▷ Przypadek 1
 7                   LEFT-ROTATE(T, p[x])                      ▷ Przypadek 1
 8                   w ← right[p[x]]                           ▷ Przypadek 1
 9           if color[left[w]] = BLACK i color[right[w]] = BLACK
10              then color[w] ← RED                            ▷ Przypadek 2
11                   x ← p[x]                                  ▷ Przypadek 2
12              else if color[right[w]] = BLACK
13                   then color[left[w]] ← BLACK               ▷ Przypadek 3
14                        color[w] ← RED                       ▷ Przypadek 3
15                        RIGHT-ROTATE(T, w)                   ▷ Przypadek 3
16                        w ← right[p[x]]                      ▷ Przypadek 3
17                   color[w] ← color[p[x]]                    ▷ Przypadek 4
18                   color[p[x]] ← BLACK                       ▷ Przypadek 4
19                   color[right[w]] ← BLACK                   ▷ Przypadek 4
20                   LEFT-ROTATE(T, p[x])                      ▷ Przypadek 4
21                   x ← root[T]                               ▷ Przypadek 4
22        else (tak samo jak część then
          z zamienionymi rolami „right" oraz „left")
23  color[x] ← BLACK
```