STRUKTURY DANYCH
I ZŁOŻONOŚĆ OBLICZENIOWA

Część 6

Drzewa czerwono-czarne, samoorganizujące się, listy z przeskokami

DRZEWA „CZERWONO-CZARNE” (*ang.: red-black trees*)

(R.Bayer -1972)

W drzewie „czerwono-czarnym” każdy węzeł (oprócz klucza, wskaźników do potomków i rodzica, oraz opcjonalnych danych nie wpływających na uporządkowanie drzewa) ma składową niosącą informację o jego kolorze, który może być czerwony albo czarny.

Dla drzew „czerwono-czarnych” każda ścieżka wiodąca od korzenia do liścia jest co najwyżej dwa razy dłuższa od ścieżki łączącej dowolny inny liść z korzeniem.

Dzięki temu

$$h_{RB} \leq 2 \lg_2 (N + 1),$$

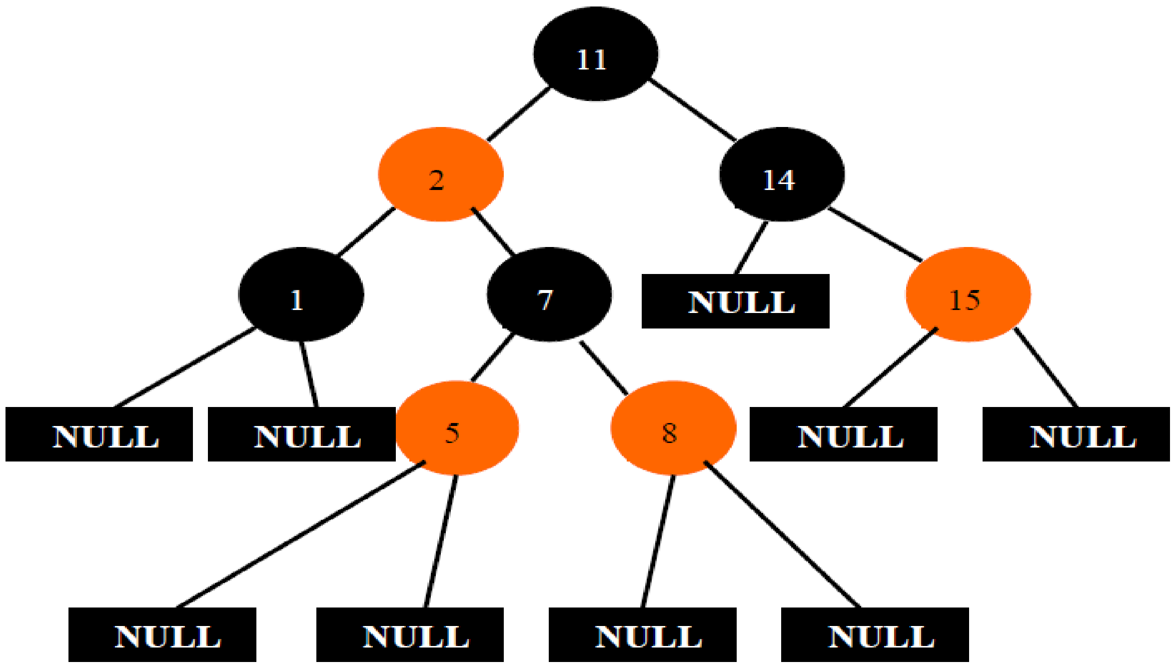
i w konsekwencji operacje wyszukiwania, wstawiania i usuwania węzłów mają złożoność O(lg N) .

```
struct node_rec
    {
    eltype key;
    struct node_rec *left, *right , *parent;
    boolean color;//czerwony albo czarny
    datatype Ti;
    };
```

Specyficzną cechą drzew „czerwono-czarnych" jest przyjęta konwencja, iż liśćmi są jedynie „czarne" węzły NULL (a więc każdemu rzeczywistemu liściowi z kluczem i „brelokiem" przyporządkowuje się dwóch potomków NULL, zaś jeżeli którykolwiek z węzłów wewnętrznych lub korzeń nie posiada któregoś z potomków, to w miejsce „braku" jest „przyszywany" nowy liść NULL).

Właściwości drzewa „czerwono-czarnego"

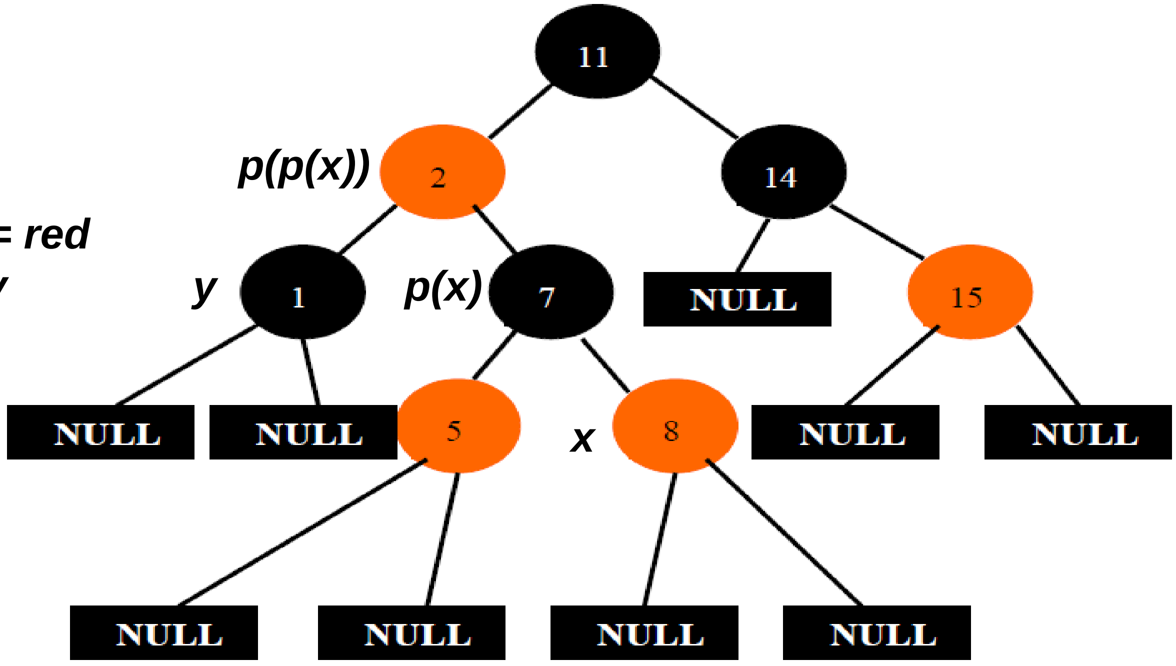
- **każdy węzeł jest „czerwony" albo „czarny";**
- **każdy liść (NULL) jest „czarny";**
- **obaj potomkowie węzła „czerwonego" są „czarni";**
- **każda ścieżka od korzenia do liścia ma tyle samo węzłów „czarnych";**
- **korzeń drzewa jest „czarny".**



Dla potrzeb prezentacji algorytmu wstawiania nowego węzła do drzewa „czerwono-czarnego”przyjęto następującą uproszczoną konwencję oznaczeń:

p(x) – „ojciec” węzła x
p(p(x)) – „dziadek” węzła x
root – korzeń drzewa
z(x) – składowa z (z ∈ {color, left, right, p}) węzła x
left_rotate (x) – lewa rotacja węzła x względem „ojca”
right_rotate (x) – prawa rotacja węzła x względem „ojca”

Np.:
color(x) = red
left(x) = NULL
right(p(x)) = x
color(p(p(x))) = red
left(p(p(x))) = y



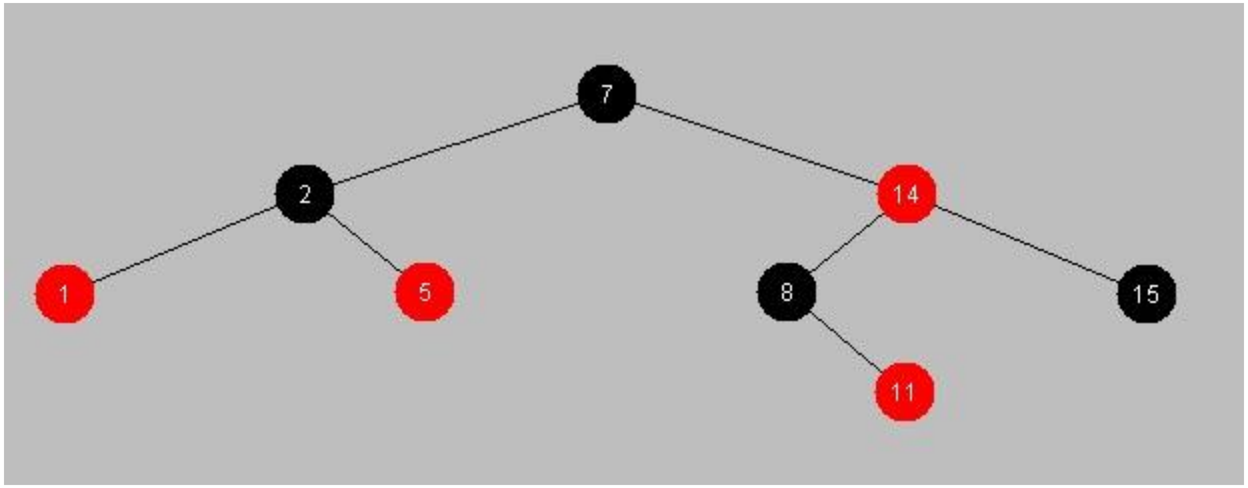
```
Red_Black_Insert (x)
{
Insert(x); // "klasyczne" wyznaczenie miejsca dla nowego węzla BST
color(x) = RED;
while((x != root) && (color(p(x)) == RED))
        {
        if(p(x) == left(p(p(x))) // "ojciec"  x jest lewym potomkiem „dziadka"
                {
                y = right(p(p(x))); // y jest „stryjem"
                if(color(y) == RED)
                        {
                        color(p(x)) = BLACK;
                        color(y) = BLACK;                    1
                        color(p(p(x))) = RED;
                        x = p(p(x)); // „wędrówka" o 2 poziomy do góry
                        }
                else // „stryj' jest BLACK
                        {
                        if(x == right(p(x)) // x jest prawym potomkiem
                                {
                                left_rotate(x);              2
                                x = p(x);
                                }
                        color(p(x)) = BLACK;
                        color(p(p(x))) = RED;                 3
                        right_rotate(p(x));
                        }
                }
```

```
else // "ojciec" x jest prawym potomkiem „dziadka"
    {
    y = left(p(p(x))); // y jest „stryjem"
    if(color(y) == RED)
        {
        color(p(x)) = BLACK;
        color(y) = BLACK;                                  4
        color(p(p(x))) = RED;
        x = p(p(x)); // „wędrówka" o 2 poziomy do góry
        }
    else // „stryj' jest BLACK
        {
        if(x == left(p(x)) // x jest lewym potomkiem
            {
            right  rotate(x);                              5
            x = p(x);
            }
        color(p(x)) = BLACK;
        color(p(p(x))) = RED;                              6
        left_rotate(p(x));
        }
    }
  }
color(root) = BLACK; // korzeń zawsze pozostaje „czarny"
}
```

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (na diagramach pomija się „czarne” liście NULL)

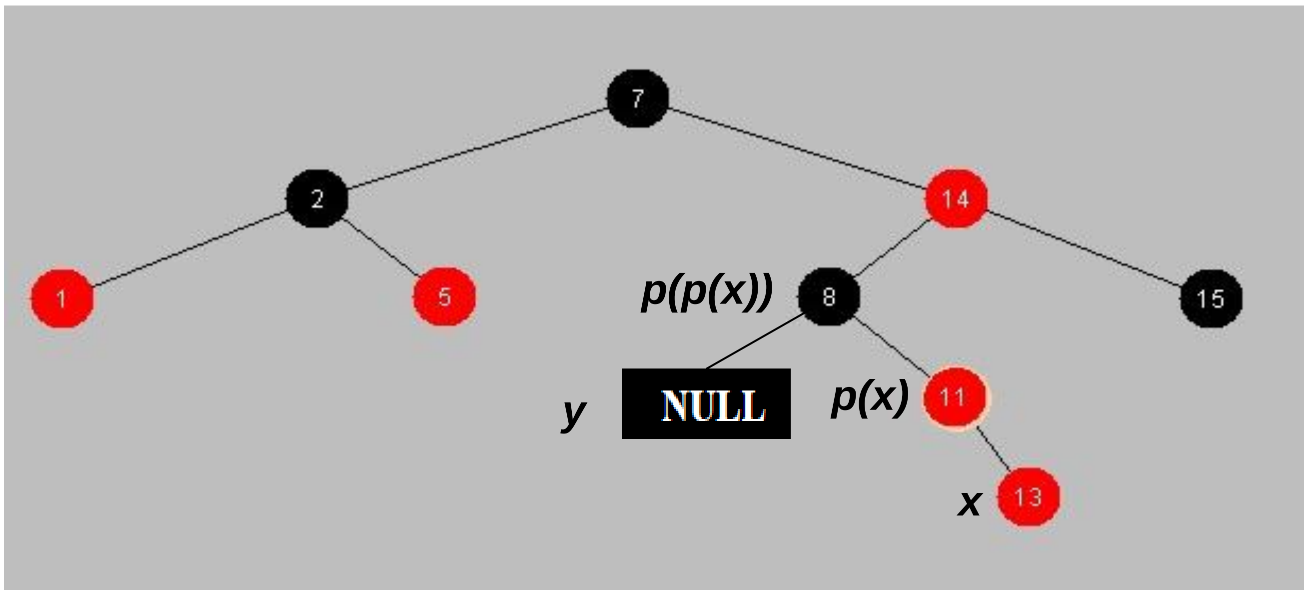
Wstawienie węzła o kluczu „13”



Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (na diagramach pomija się „czarne” liście NULL)

Wstawienie węzła o kluczu „13”



„klasyczne” wstawienie liścia *Insert(x)*

color(p(x)) = RED

p(x) = right(p(p(x))

color(y) = BLACK

x = right(p(x))

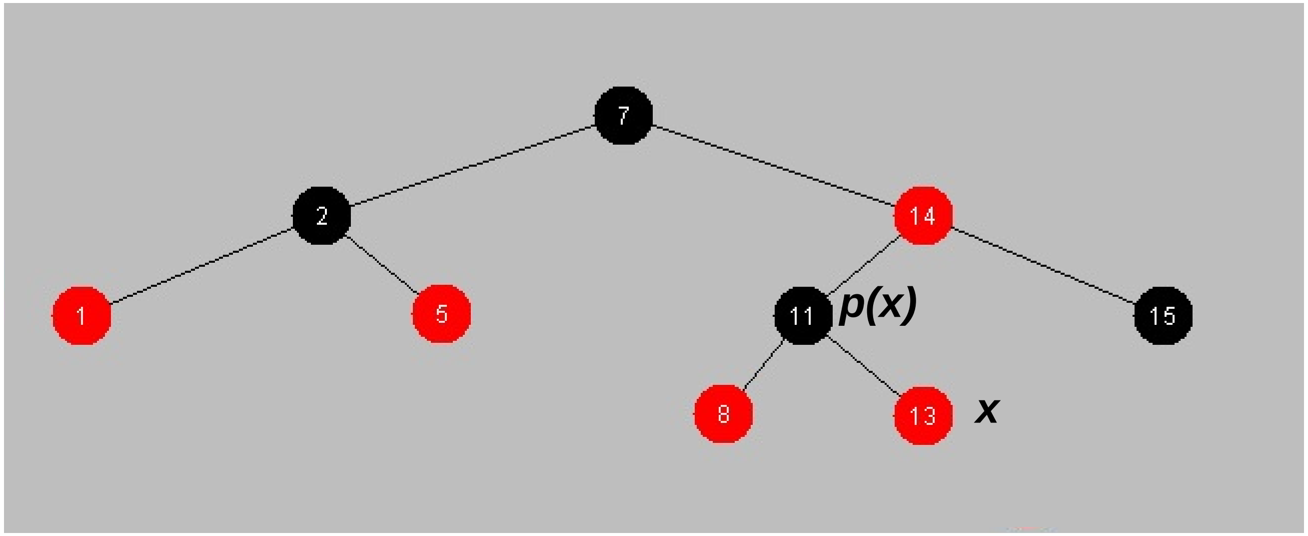
color(p(x)) ← BLACK

color(p(p(x)) ← RED

left_rotate(p(x))

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (na diagramach pomija się „czarne” liście NULL)
Wstawienie węzła o kluczu „13”

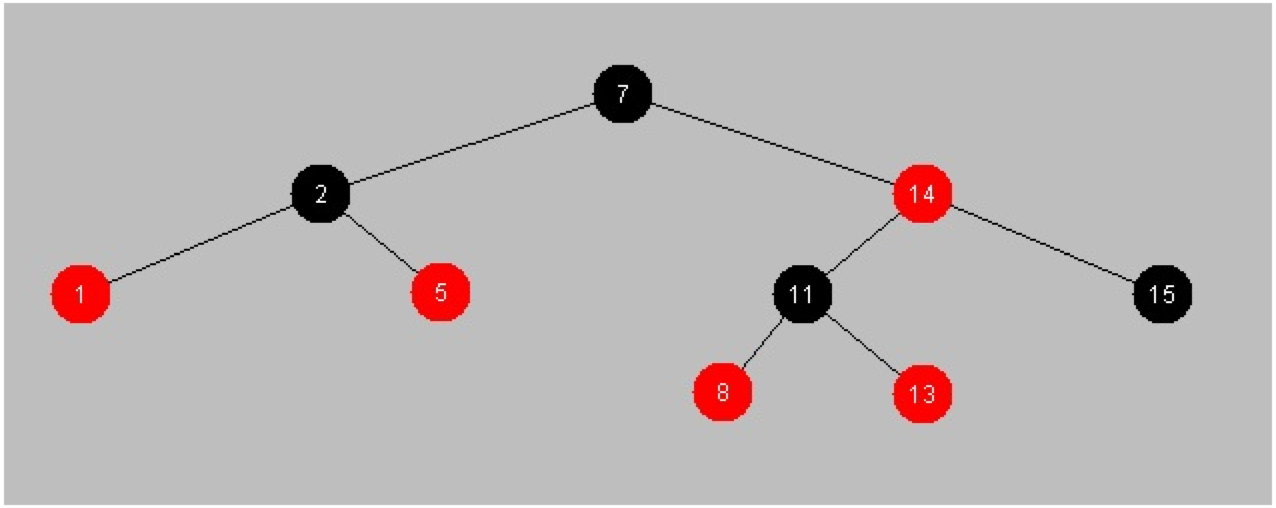


color(p(x)) = BLACK
Koniec rekonstrukcji

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

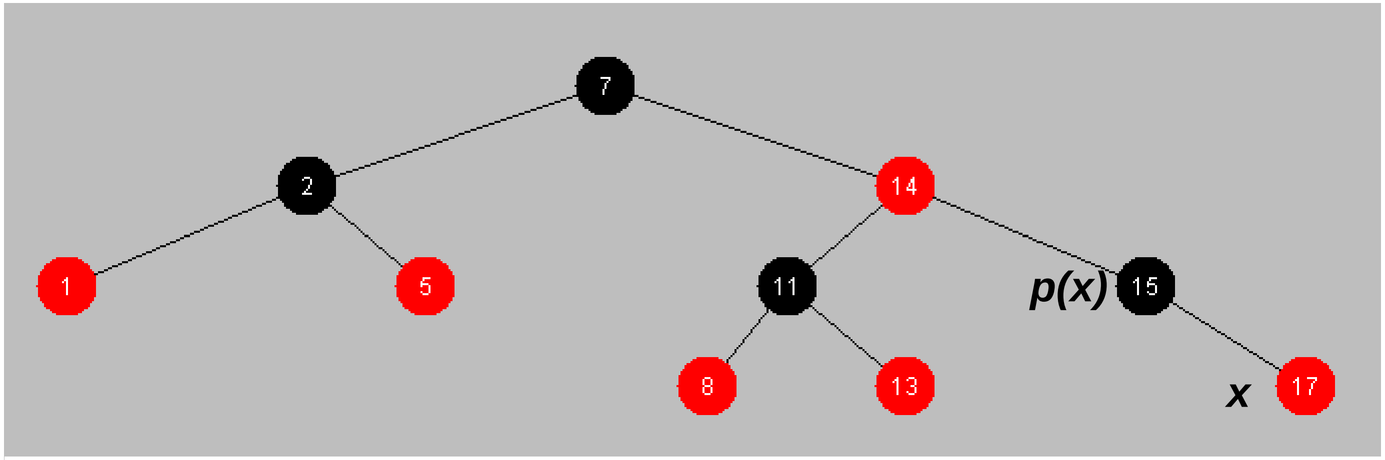
Wstawienie węzła o kluczu „17”



Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „17”



„klasyczne” wstawienie liścia *Insert(x)*

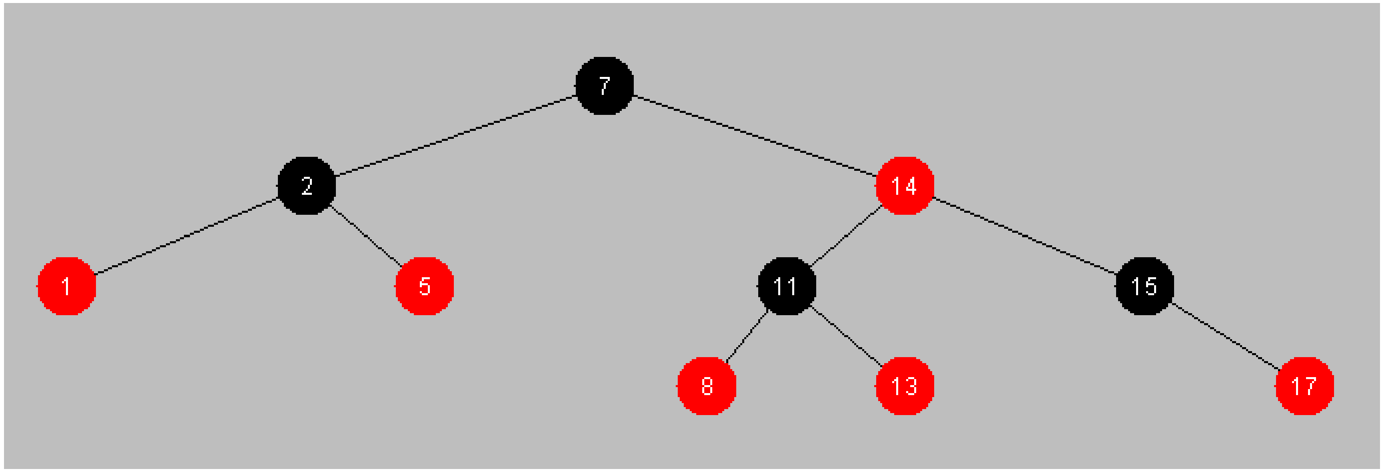
color(p(x)) = BLACK

Koniec rekonstrukcji

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

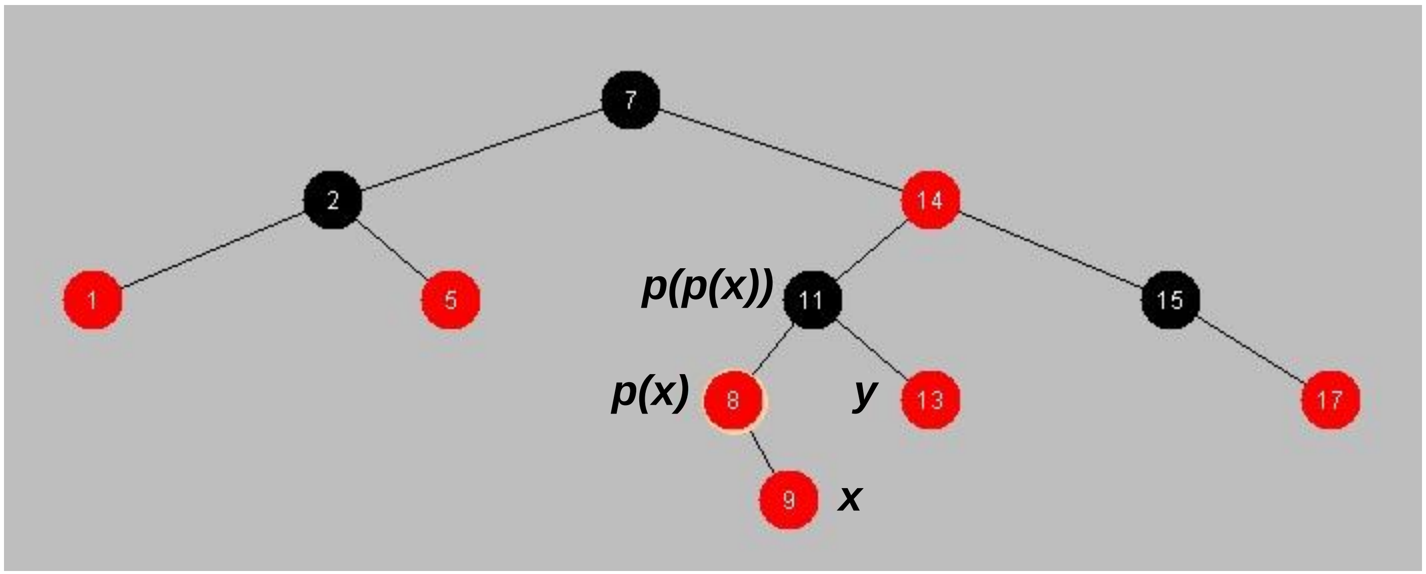
Wstawienie węzła o kluczu „9”



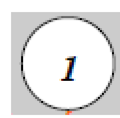
Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „9”



„klasyczne” wstawienie liścia *Insert(x)*
color(p(x)) = RED
p(x) = left(p(p(x))
color(y) = RED

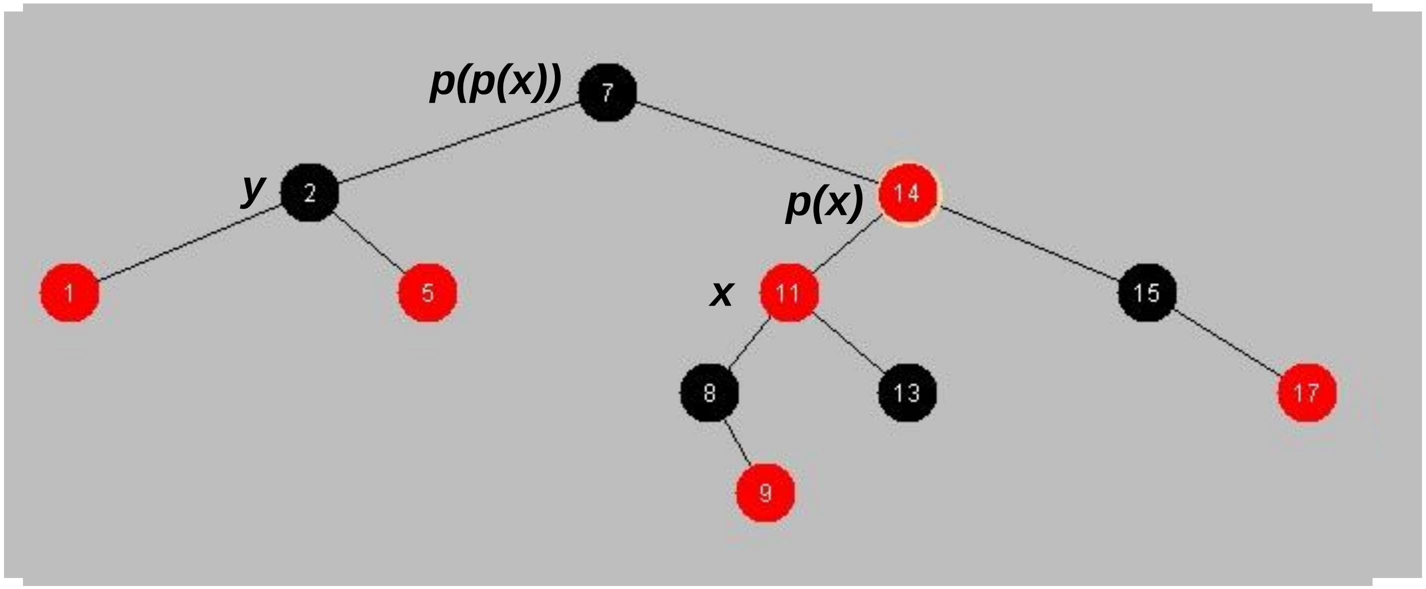


color(p(x)) ← BLACK
color(y) ← BLACK
color(p(p(x))) ← RED
x ← p(p(x))

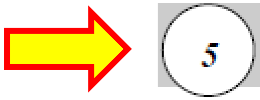
Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „9”



color(p(x)) = RED
p(x) = right(p(p(x))
color(y) = BLACK
x = left(p(x))

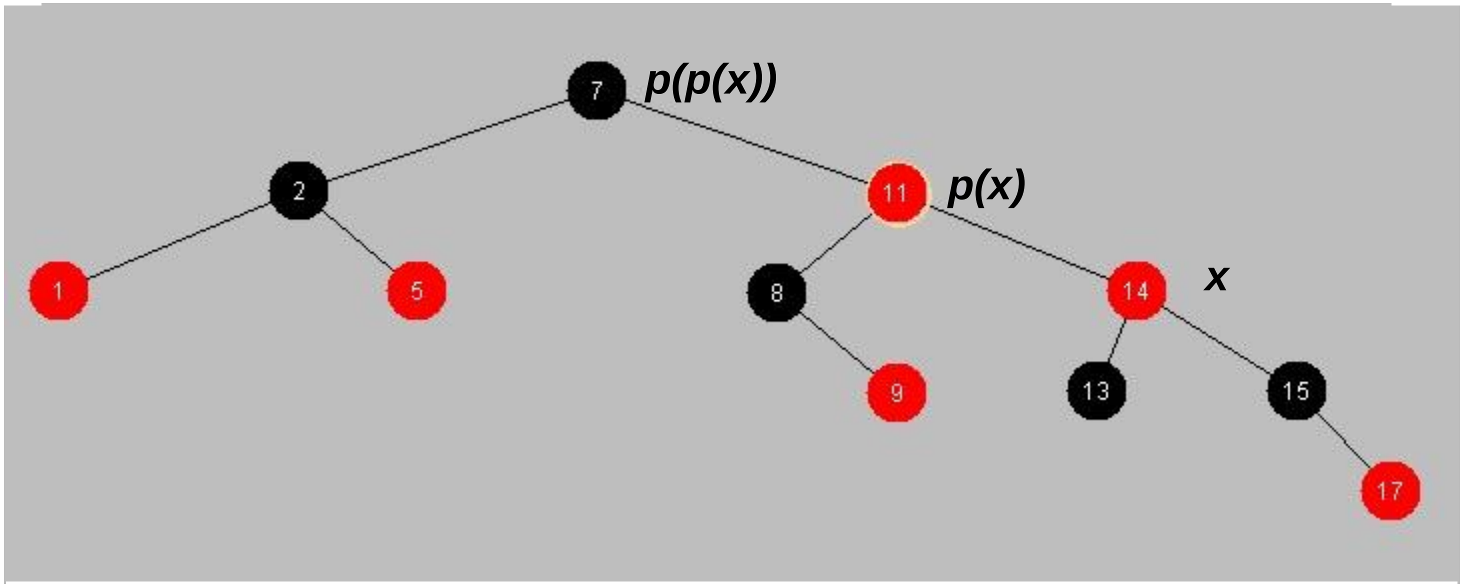


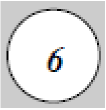
right_rotate(x)
x ← p(x)

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „9”



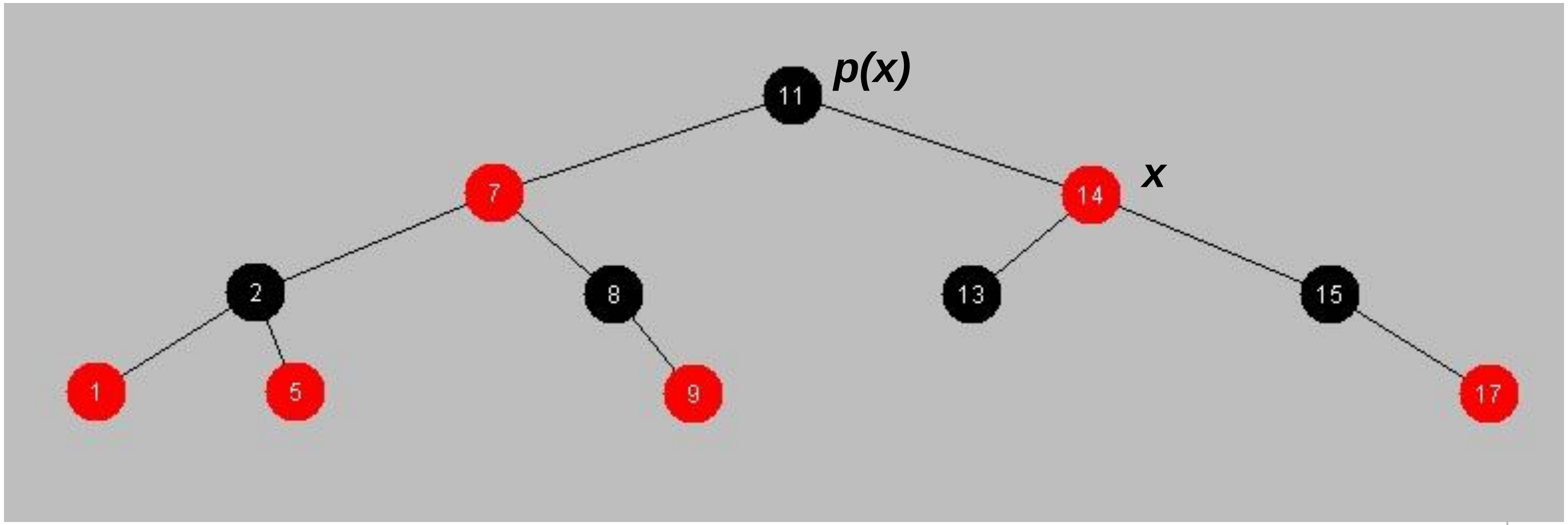


color(p(x)) ← BLACK
color(p(p(x))) ← RED
left _rotate(p(x))

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „9”

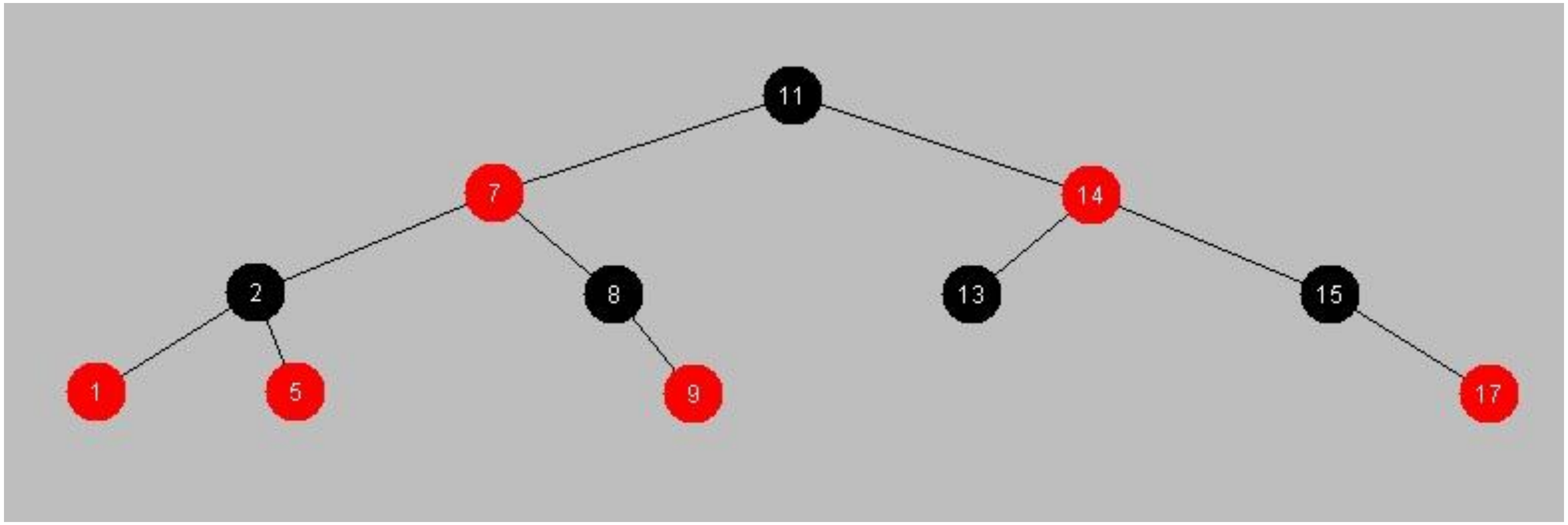


color(p(x)) = BLACK
Koniec rekonstrukcji

Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

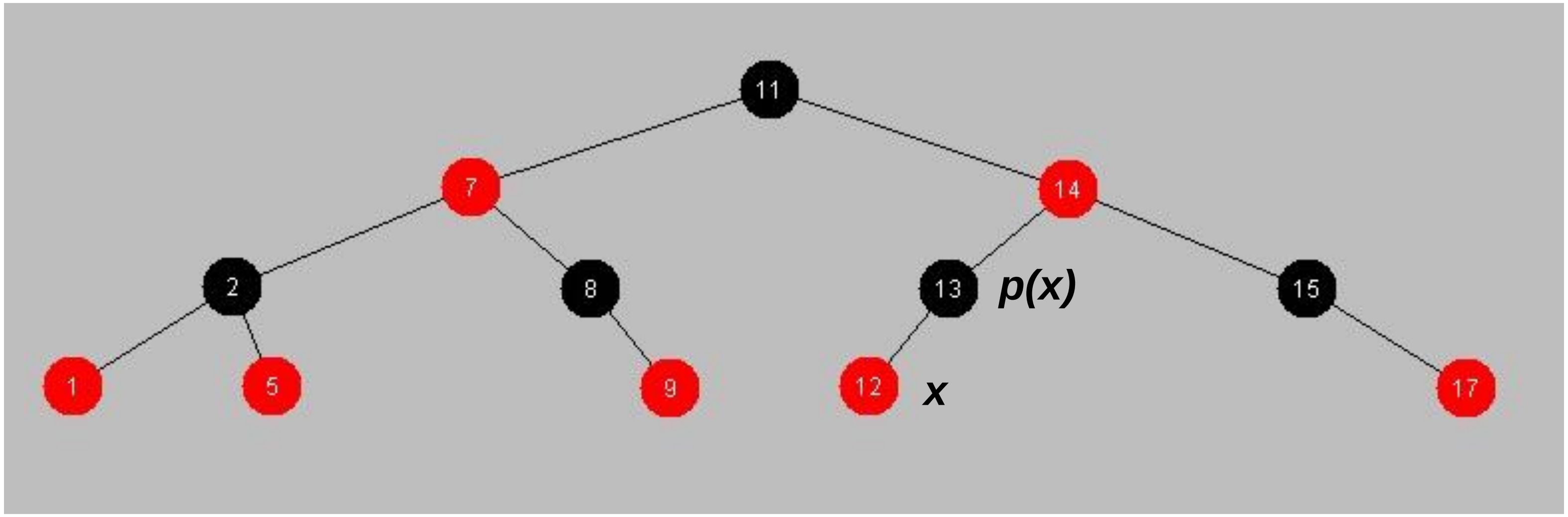
Wstawienie węzła o kluczu „12”



Wstawianie nowego węzła drzewa „czerwono-czarnego”

Przykład (cd.)

Wstawienie węzła o kluczu „12”



„klasyczne” wstawienie liścia *Insert(x)*

color(p(x)) = BLACK

Koniec rekonstrukcji

DRZEWA SAMOKORYGUJĄCE SIĘ, SAMOORGANIZUJĄCE SIĘ (ang.: self-adjusting, self-organizing trees)

Wyważanie lub dokładne wyważanie drzew BST ma sens wtedy, gdy częstotliwość odwołań do wszystkich składowych jest identyczna.

W praktycznych zastosowaniach (np.: skorowidze, słowniki) niektóre z węzłów drzewa są „odwiedzane” częściej niż pozostałe.

Wówczas rozsądne jest takie kształtowanie struktury drzewa, by składowe, do których odwołuje się częściej, znajdowały się bliżej korzenia drzewa.

W zasadzie powinno się rozszerzyć wówczas postać danej typu podstawowego o licznik „odwiedzin” w węźle o określonej wartości klucza, który jest inicjowany podczas wstawiania nowego węzła do drzewa i zwiększany o 1 podczas każdych odwiedzin.

```
struct node_rec
    {
    eltype key;
    struct node_rec *left, *right;
    int visit_counter;
    datatype Ti;
    };
```

Jednak samo zwiększanie stanu licznika *visit_counter* jest niewystarczające.

Należy również zaproponować taką metodę rekonstrukcji drzewa, by węzły „odwiedzane” częściej były przesuwane na poziomy bliższe korzeniowi.

Metody te powinny być mniej czasochłonne i mniej pamięciochłonne, niż w przypadku wyważania drzew.

W powszechnie spotykanych rozwiązaniach tego problemu dominują dwa „klasyczne” podejścia:

- **odwiedzony węzeł jest przesuwany o jeden lub więcej poziomów „wyżej”** (przy wielokrotnych odwołaniach i powtarzaniu tej operacji węzeł będzie się systematycznie „piął” ku poziomowi korzenia);

- **odwiedzony węzeł jest przesuwany od razu na poziom korzenia** (przy zachowaniu, dzięki odpowiedniej procedurze, relacji porządkującej drzewo; przy takim podejściu wychodzi się z założenia, iż prawdopodobieństwo ponownego odwiedzenia tego węzła jest względnie duże).

DRZEWA ROZCHYLANE *(ang.: splay trees)*

(D.D.Sleator, R.E.Tarjan – 1985)

Odwiedzony węzeł jest „promowany” do poziomu korzenia, przy czym formalnie rotacje są zagregowane w operacje podwójne, zależne od wzajemnych relacji między węzłem „promowanym” i jego bezpośrednim „rodzicem” oraz dziadkiem (jeśli węzeł w wyniku tych rotacji „dotrze” do poziomu 2-go, to pozostaje ostatnia pojedyncza rotacja).

Podwójne rotacje

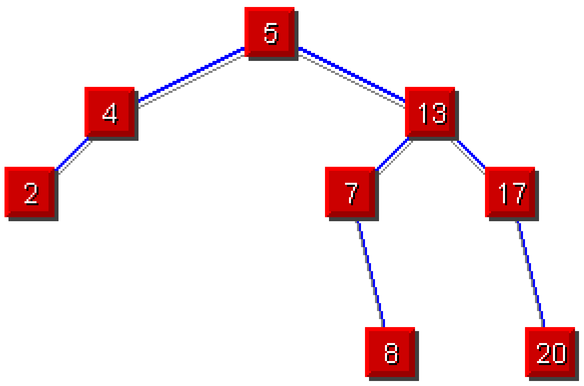
Konfiguracja jednorodna („zig-zig”) – węzeł jest prawym potomkiem „rodzica”, zaś „rodzic” prawym potomkiem „dziadka” „promowanego” węzła (lub odpowiednio: węzeł jest lewym potomkiem „rodzica”, zaś „rodzic” lewym potomkiem „dziadka” „promowanego” węzła) .

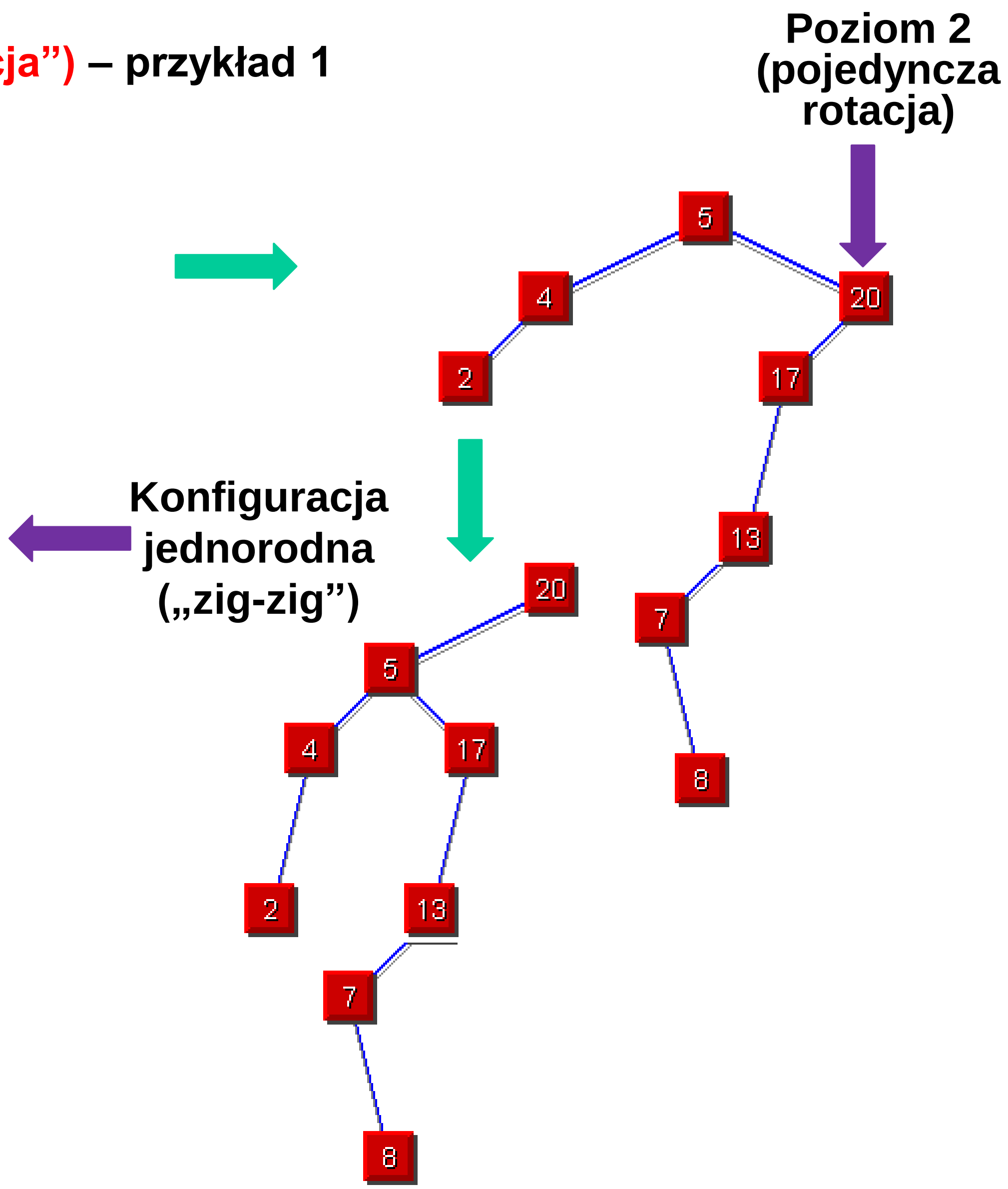
Najpierw jest rotowany „rodzic”, zaś następnie jego „promowany” potomek.

Konfiguracja niejednorodna („zig-zag”) – węzeł jest prawym potomkiem „rodzica”, zaś „rodzic” lewym potomkiem „dziadka” „promowanego” węzła (lub odpowiednio: węzeł jest lewym potomkiem „rodzica”, zaś „rodzic” prawym potomkiem „dziadka” „promowanego” węzła) .

„Bohaterem” obu rotacji jest „promowany” potomek.

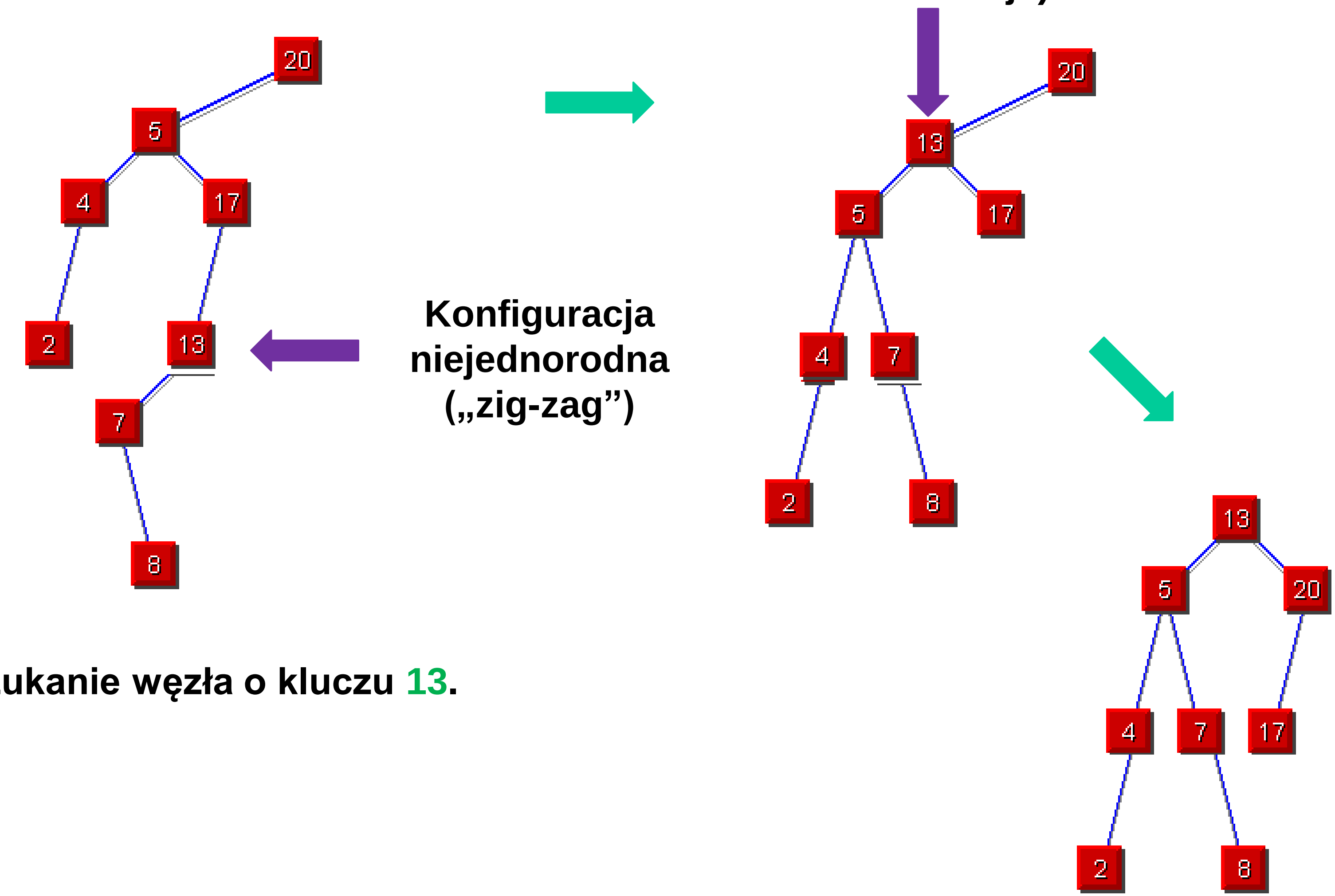
Wyszukiwanie węzła (i „promocja”) – przykład 1





Wyszukanie węzła o kluczu 20.

Wyszukiwanie węzła (i „promocja”) – przykład 2



Wyszukanie węzła o kluczu 13.

Wstawianie nowego węzła do drzewa rozchylanego

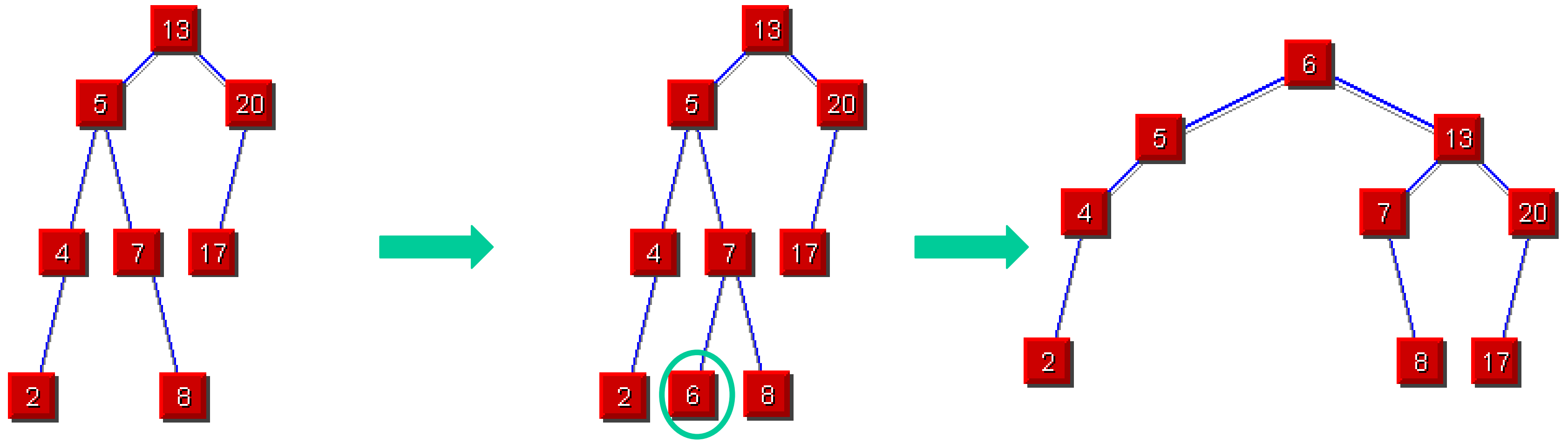
Faza I

**„Klasyczne" poszukiwanie wolnego miejsca dla nowego węzła.
Nowy węzeł staje się liściem.**

Faza II

„Promowanie" nowego węzła za pomocą rotacji (z uwzględnieniem chwilowych konfiguracji – jednorodnej lub niejednorodnej) do poziomu korzenia.

Wstawianie nowego węzła – przykład



Wstawienie węzła o kluczu 6.

Usuwanie węzła z drzewa rozchylanego

Wariant A

Faza I

„Klasyczne" usunięcie węzła z drzewa BST (wraz z alternatywną promocją następnika albo poprzednika w przypadku usunięcia węzła stopnia 2).

Faza II

„Promowanie" „rodzica" usuniętego węzła za pomocą rotacji (z uwzględnieniem chwilowych konfiguracji – jednorodnej lub niejednorodnej) do poziomu korzenia.

Wariant B

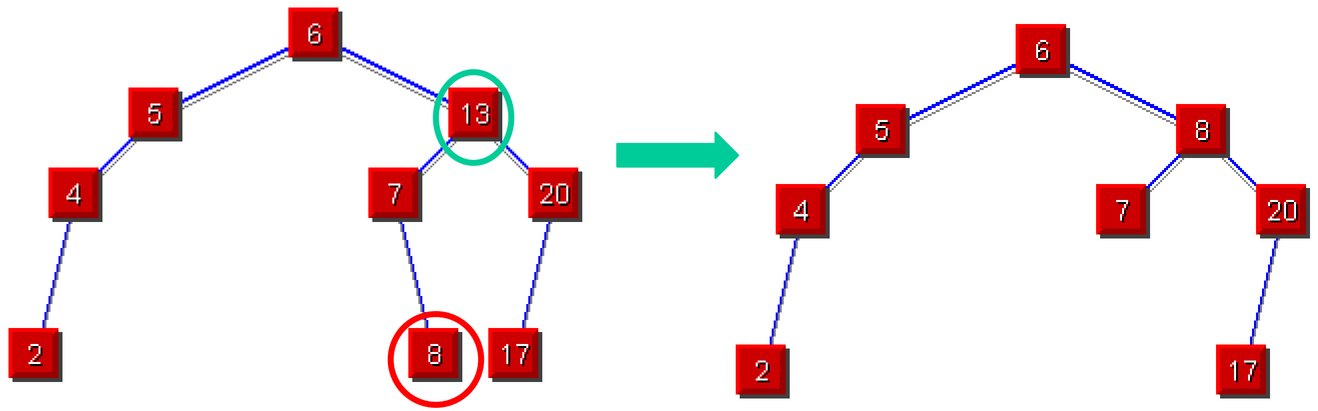
Faza I

„Promowanie" węzła przeznaczonego do usunięcia za pomocą rotacji (z uwzględnieniem chwilowych konfiguracji – jednorodnej lub niejednorodnej) do poziomu korzenia.

Faza II

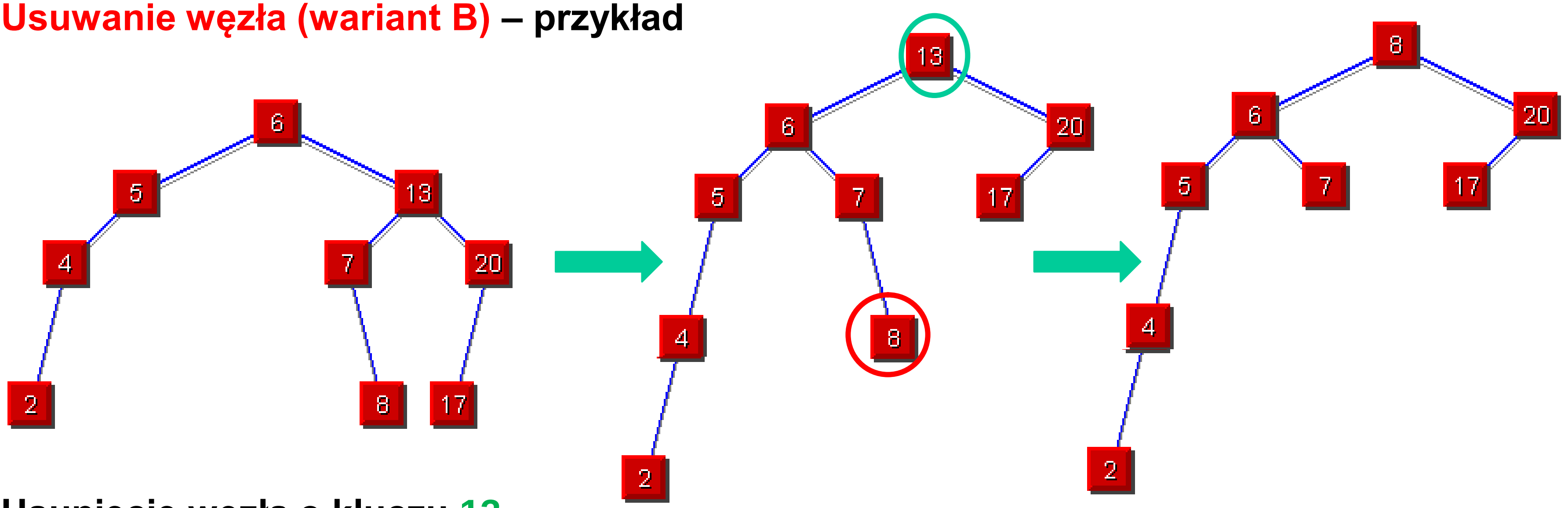
„Klasyczne" usunięcie węzła/korzenia z drzewa BST (wraz z alternatywną promocją następnika albo poprzednika w przypadku usunięcia węzła stopnia 2).

Usuwanie węzła (wariant A) – przykład



Usunięcie węzła o kluczu 13.

Usuwanie węzła (wariant B) – przykład



Usunięcie węzła o kluczu 13.

OPTYMALNE DRZEWA BST (ang.: optimal BST, OBST)

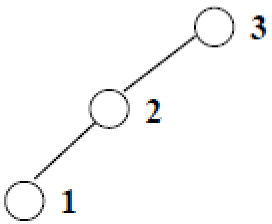
Znajomość prawdopodobieństwa *a priori* „odwiedzin" w każdym z węzłów drzewa BST może skłaniać do takiej konstrukcji drzewa poszukiwań binarnych, by węzły o większym prawdopodobieństwie „odwiedzin" π_i znajdowały się na poziomach najwyższych.

Optymalnym drzewem poszukiwań binarnych będzie takie, dla którego ważona długość drogi poszukiwań:

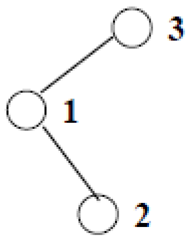
$$w = \sum_{i=1}^{N} \pi_i * p_i$$

gdzie p_i jest numerem poziomu, na którym znajduje się węzeł i-ty , będzie minimalna.

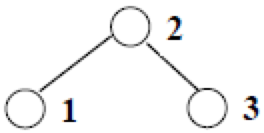
Przykład:



$w_1 = 11/7$

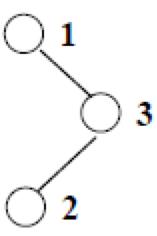


$w_2 = 12/7$

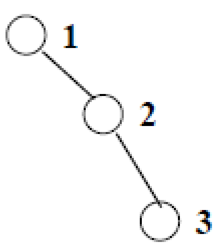


$w_3 = 12/7$

$\pi_1 = 1/7$ $\pi_2 = 2/7$ $\pi_3 = 4/7$



$w_4 = 15/7$



$w_5 = 17/7$

LISTY Z PRZESKOKAMI (*ang.: skip lists*)

(W.Pugh -1989)

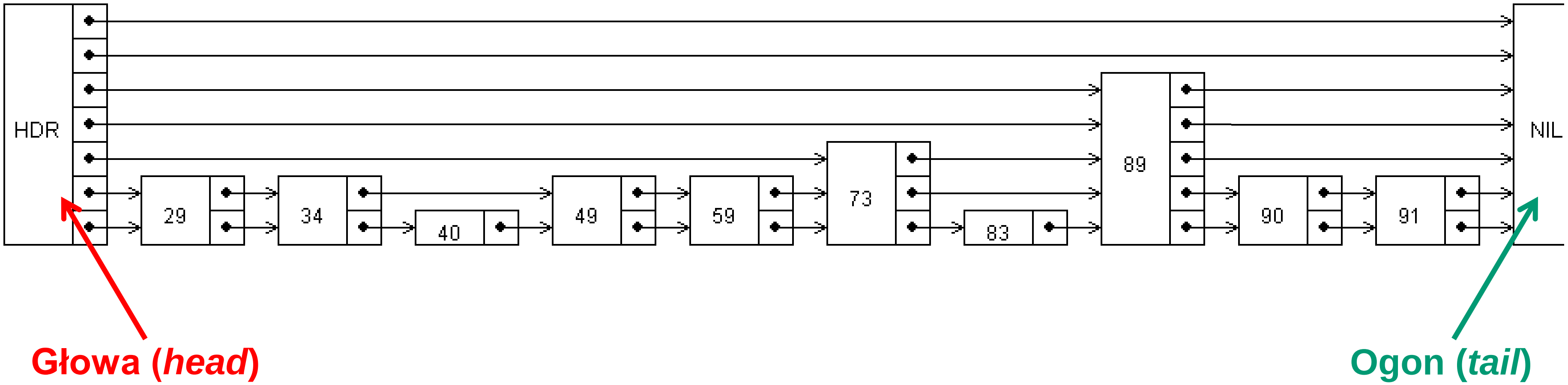
Każdy z węzłów listy (liniowej) z przeskokami oprócz składowej kluczowej oraz „breloka" ma określoną wysokość, ograniczoną od góry maksymalną wartością, będącą parametrem charakteryzującym całą listę.

Złożoność operacji wyszukiwania, wstawiania i usuwania węzłów wynosi O(N), tak jak w „klasycznej" liście liniowej, lecz oczekiwana złożoność czasowa wykonywania tych operacji słownikowych jest klasy O(lg N).

Pierwszy węzeł, zwany głową (*head*), oraz ostatni, zwany ogonem (*tail*), mają wysokość maksymalną i są tworzone w procesie inicjalizacji pustej listy. Wszystkie „właściwe" węzły będą lokowane między głową i ogonem (można z głową i ogonem powiązać umowne wartości składowych kluczowych, odpowiednio: MINUS_MIN /-∞ i PLUS_MAX/+∞).

Wysokość węzła jest losowo definiowana podczas wstawiania nowego węzła do listy. Dodatkowym parametrem charakteryzującym listę jest prawdopodobieństwo p „pojawienia się" węzła o określonej wartości klucza na wysokości (poziomie) $i+1$, pod warunkiem, że „pojawił się" on już na wysokości (poziomie) i (wg. pierwotnej propozycji W.Pugh - $p = 0.5$).

Przykład listy z przeskokami o wysokości LMAX=7 i prawdopodobieństwie PROB=0.5.



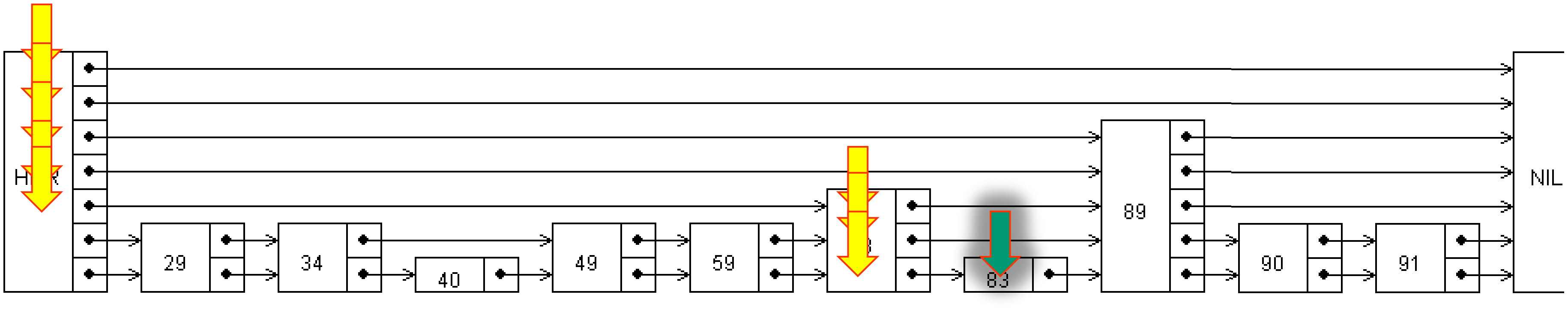

```
struct node_rec
    {
    eltype key;
    int height;
    struct node_rec *next[LMAX];
    datatype Ti;
    };
```

Wyszukiwanie węzła o wskazanej wartości klucza

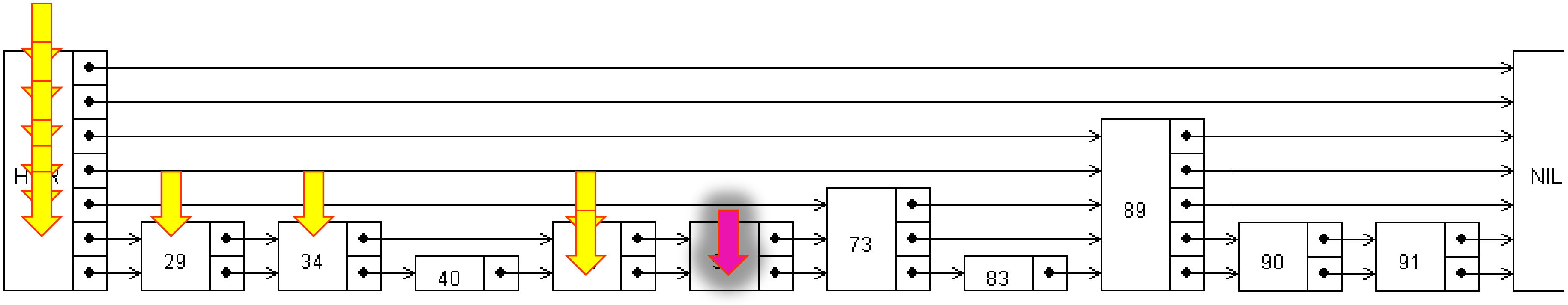
```
search(head, key)
{
  x ← head;
  for (i ← LMAX-1;  i ≥ 0; i--)//początek wyszukiwania na najwyższym poziomie
            while(x→next[i]→key < key)
                        x ←  x→next[i];//"wędrówka"  wzdłuż listy na poziomie i-tym

  x ← x→next[0];
  if(x→key  = key)
            return (x);
  return(NULL);
}
```

Wyszukiwanie węzła o wskazanej wartości klucza – przykłady

Poszukiwanie węzła o kluczu 83 (SUKCES).



Poszukiwanie węzła o kluczu 50 (NIEPOWODZENIE).



Wstawianie nowego węzła

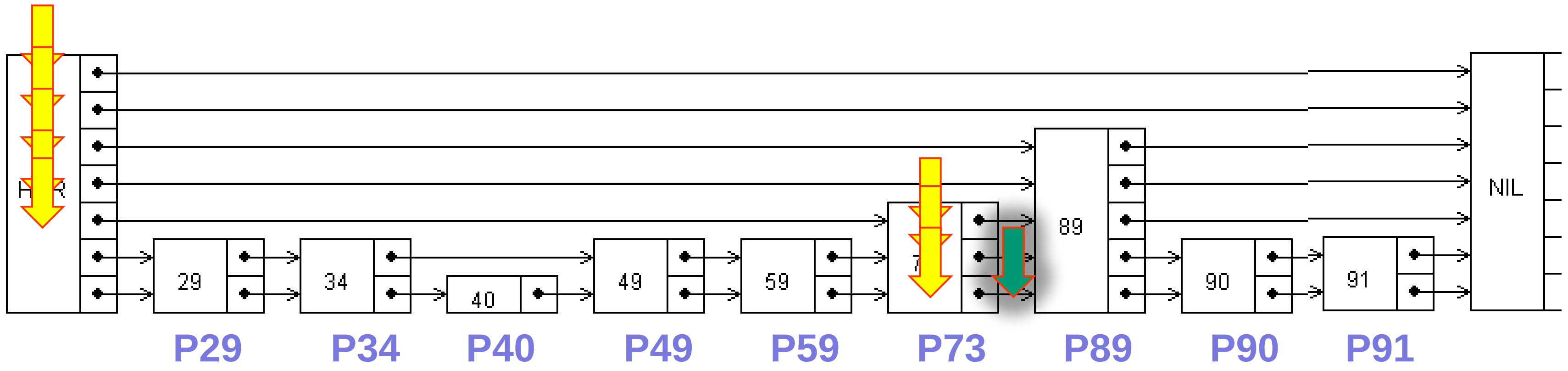
```
insert(head, new_key)
{    x ← head;
     for (i ← LMAX-1;  i ≥ 0; i--)//początek wyszukiwania na najwyższym poziomie
     {
      while(x→next[i]→key < new_key)
            x ←  x→next[i];//”wędrówka”  wzdłuż listy na poziomie i-tym
      update[i] ← x;//ref. do ostatniego węzła na poziomie i-tym o kluczu < new_key
     }
     x ← x→next[0];
     if(x→key = new_key) return(-2);//jest już na liście węzeł o kluczu new_key
     new_node→key ←  new_key;
     new_node→height ←  random_level();//generowanie losowej „wysokości"
     for(i ← 0; i < new_node→height; i++)
            {
            new_node→next[i] ← update[i]→next[i];
            update[i]→next[i] ← new_node;
            }//”wklejenie” nowego węzła do listy
     return(0);
}//UWAGA: Generując wcześniej height dla nowego węzła można pierwszą pętlę for zacząć od i = height
```

Wstawianie nowego węzła – procedura generowania wysokości nowego węzła

```
random_level()
{
int level ←1;
while((rand()%100 < PROB*100) &(level < LMAX))
        level++;
return(level);
}
```

update[6] = head
update[5] = head
update[4] = head
update[3] = head
update[2] = P73
update[1] = P73
update[0] = P73

Wstawianie nowego węzła – przykład



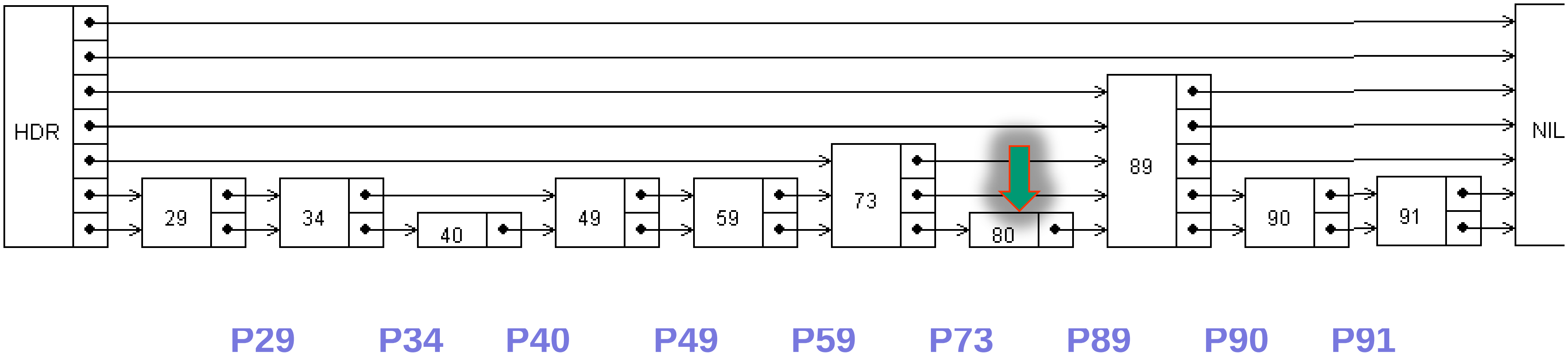
Wstawienie węzła o kluczu 80.

Wstawianie nowego węzła – procedura generowania wysokości nowego węzła

```
random_level()
{
int level ←1;
while((rand()%100 < PROB*100) &(level < LMAX))
        level++;
return(level);
}
```

update[6] = head
update[5] = head
update[4] = head
update[3] = head
update[2] = P73
update[1] = P73
update[0] = P73

Wstawianie nowego węzła – przykład



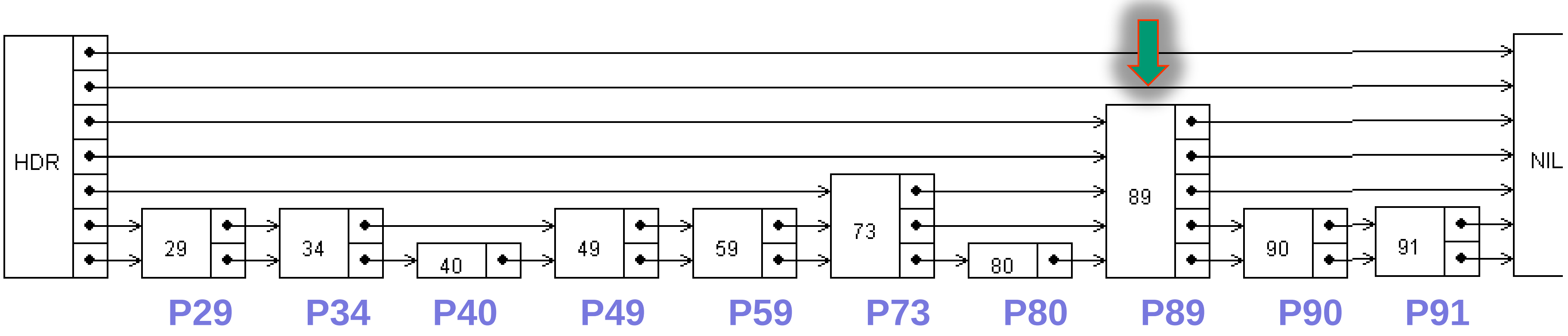
Wstawienie węzła o kluczu 80.

Usuwanie węzła o wskazanej wartości klucza

```
delete (head, key)
{
    x ← head;
   for (i ← LMAX-1;  i ≥ 0; i--)//początek wyszukiwania na najwyższym poziomie
   { while(x→next[i]→key < key)
          x ← x→next[i];//"wędrówka"  wzdłuż listy na poziomie i-tym
    update[i] ← x;//referencja do ostatniego węzła na poziomie i-tym o kluczu < key
    }
   x ← x→next[0];
   if(x→key > key) return(-1);//brak węzła o wskazanej wartości klucza
   for(i ← 0; i < x→height; i++)
          update[i]→next[i] ← x→next[i];//"sklejenie" listy w miejscu usuwanego węzła
   return(0);
}
```

Usuwanie węzła – przykład

update[6] = head
update[5] = head
update[4] = head
update[3] = head
update[2] = P73
update[1] = P73
update[0] = P80



Usuwanie węzła o kluczu 89.

Usuwanie węzła – przykład

update[6] = head

update[5] = head

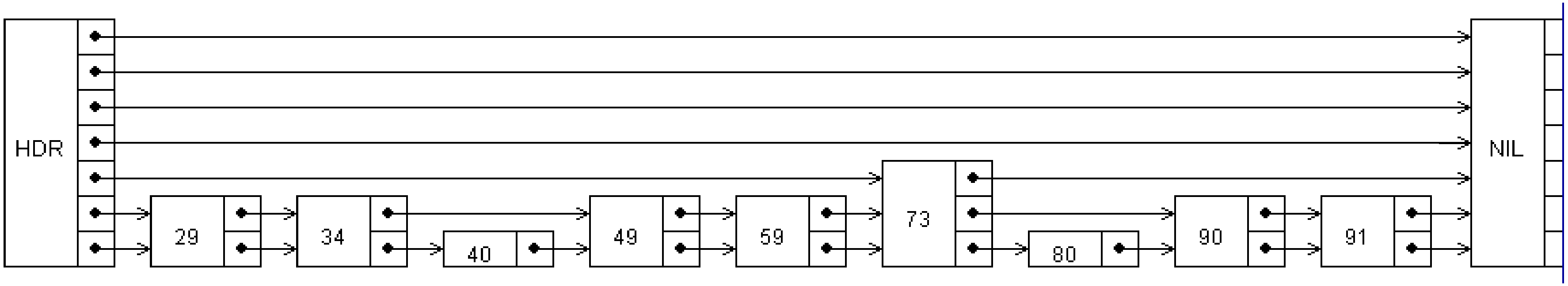
update[4] = head

update[3] = head

update[2] = P73

update[1] = P73

update[0] = P80



Usuwanie węzła o kluczu 89.

Koniec części 6