Алгоритми та складність

II семестр

Лекція 2

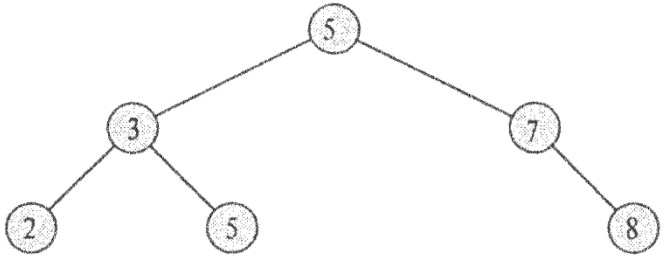
Дерево пошуку

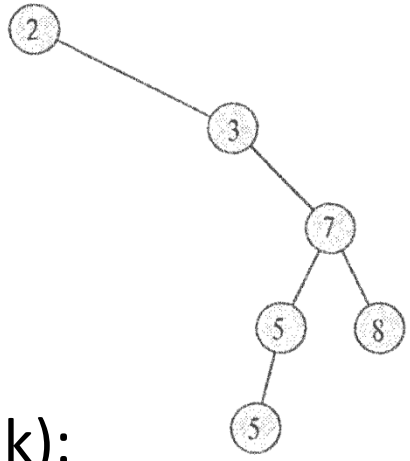
- Структура даних, що підтримує ряд операцій з динамічними множинами:
  - пошук елемента;
  - пошук мінімального та максимального значення;
  - пошук попереднього і наступного елемента;
  - вставка та видалення елемента.
- Набір операцій дозволяє використовувати дерево пошуку для реалізації як словника, так і черги з пріоритетами.
- Основні операції виконуються за час, пропорційний висоті дерева.

Бінарні дерева пошуку

- Поле-ключ *key*.
- Вказівники на батька, лівого та правого синів *p*, *left*, *right*.
- x – корінь: *p*[x]=NIL.
- x – лист: *left*[x]=*right*[x]=NIL.
- Для кожного вузла *x* виконується *властивість бінарного дерева пошуку*:
  - якщо вузол *y* знаходиться в лівому піддереві вузла *x*, то *key*[y] ≤ *key*[x];
  - якщо вузол *y* знаходиться в правому піддереві вузла *x*, то *key*[x] ≤ *key*[y].

Бінарні дерева пошуку

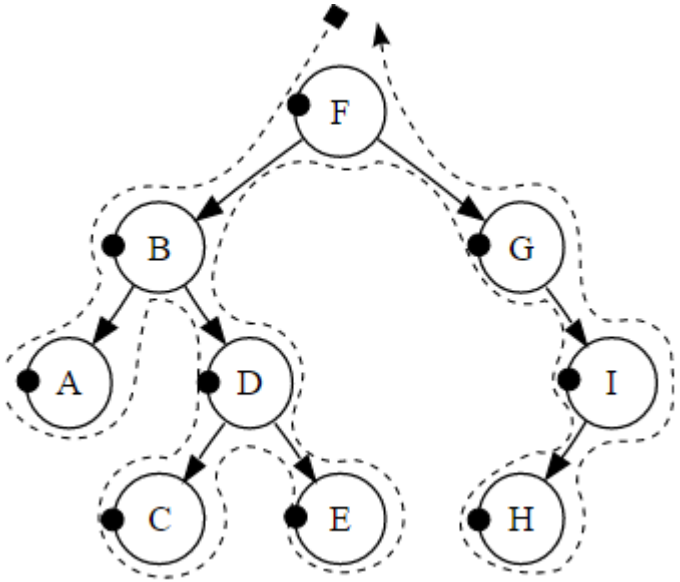




Обходи дерев

- Прямий порядок (preorder tree walk):

  корінь, ліве піддерево, праве піддерево.

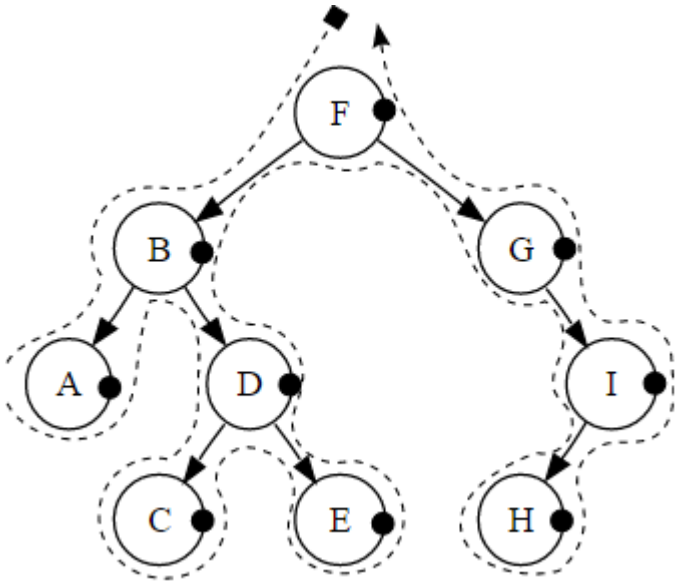


F, B, A, D, C, E, G, I, H

Бінарні дерева пошуку

- Обернений порядок (postorder tree walk):

  ліве піддерево, праве піддерево, корінь.

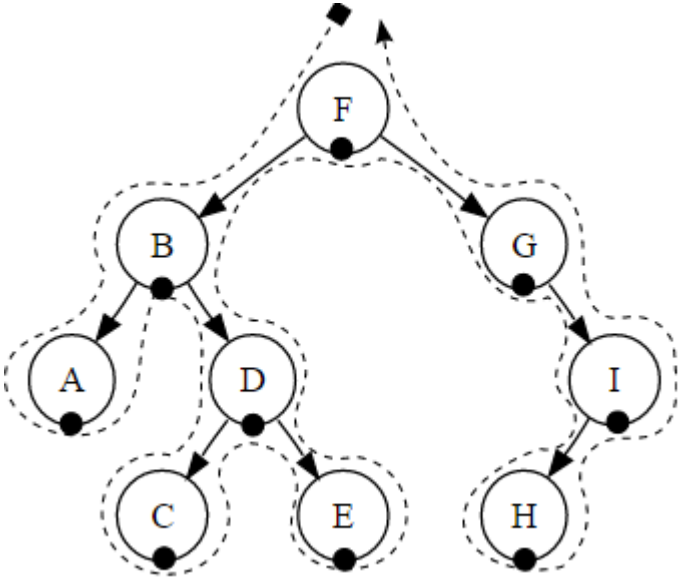


A, C, E, D, B, H, I, G, F

Бінарні дерева пошуку

- Симетричний порядок (inorder tree walk):

  ліве піддерево, корінь, праве піддерево.



A, B, C, D, E, F, G, H, I

```
INORDER_TREE_WALK(x)
1  if x ≠ NIL
2      then INORDER_TREE_WALK(left[x])
3           print key[x]
4           INORDER_TREE_WALK(right[x])
```

Бінарні дерева пошуку

- Коректність алгоритму INORDER_TREE_WALK(*root*[T]) безпосередньо випливає з властивості бінарного дерева пошуку.
- Після початкового виклику процедура викликається для кожного вузла дерева рівно двічі: для лівого та правого сина, тому дерево обходиться за лінійний час:

Теорема. Якщо x – корінь піддерева з n вузлами, то INORDER_TREE_WALK(x) виконується за час $\Theta(n)$.

Доведення. Нижня границя $\Omega(n)=n$ – бо відвідуються всі n вершин. Нехай T(n) – час виконання процедури INORDER_TREE_WALK для параметра – кореня дерева з n вузлами. Якщо параметр порожній, покладемо $T(0)=c$, де c – деяка додатна константа (час перевірки $x \neq$ NIL).

Доведення (далі). При n>0 вважатимемо, що процедура викличеться перший раз для піддерева з k вузлами, а другий – для піддерева з (n–k–1) вузлами. Тоді загальний час T(n) = T(k) + T(n–k–1) + d, де d – час роботи за винятком рекурсивних викликів.

Покажемо методом підстановок, що T(n)=O(n). Доведемо T(n)=(c+d)·n+c.

При n=0:

$$T(0) = (c+d)\cdot 0 + c = c.$$

При n>0:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d = \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d = \\ &= (c+d)n + c - (c+d) + c + d = (c+d)n + c, \end{aligned}$$

що і треба довести.

Пошук елемента

```
TREE_SEARCH(x, k)
1  if x = NIL или k = key[x]
2      then return x
3  if k < key[x]
4      then return TREE_SEARCH(left[x], k)
5      else  return TREE_SEARCH(right[x], k)
```

Хвостову рекурсію можна розгорнути в цикл

```
ITERATIVE_TREE_SEARCH(x, k)
1  while x ≠ NIL и k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Відвідані вузли утворюють шлях від кореня донизу, тому час виконання O(h), де h – висота дерева.

Пошук мінімума і максимума

```
TREE_MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

```
TREE_MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

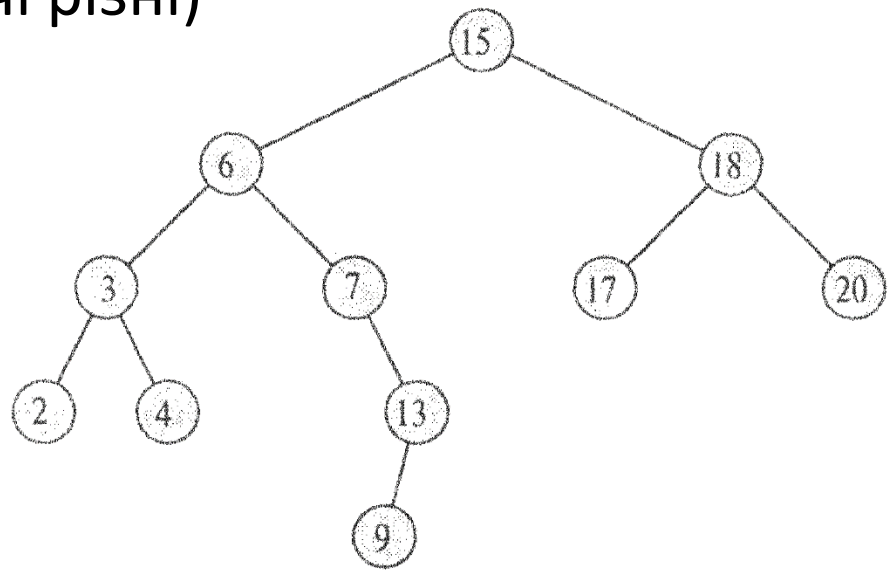
- Властивість бінарного дерева пошуку гарантує коректність обидвох процедур.
- Відвідані вузли утворюють шлях від кореня донизу, тому час виконання O(h), де h – висота дерева.

Попередній і наступний елементи

- Шукається вузол з найменшим ключем, що більший за *key*[x] (якщо всі ключі різні)

Наступник 6 ?

Наступник 13 ?

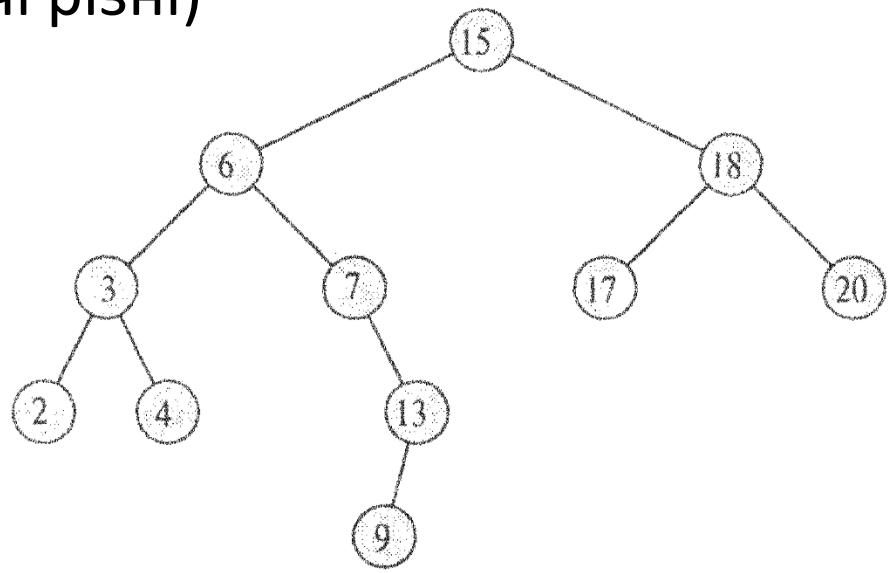


Попередній і наступний елементи

- Шукається вузол з найменшим ключем, що більший за *key*[x] (якщо всі ключі різні)

  Наступник 6 ?

  Наступник 13 ?



- Найменший елемент правого піддерева з коренем x.
- Якщо ж праве піддерево порожнє, найменший предок x, чий лівий син є предком x (рухаємося вгору, поки не знайдемо елемент – лівий син свого предка, його батько – шуканий вузол).

Попередній і наступний елементи

```
TREE_SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE_MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL и x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

- Відвідані вузли утворюють шлях від вузла або донизу, або вгору, тому час виконання O(h), де h – висота дерева.
- Процедура пошуку попередника TREE_PREDECESSOR буде симетричною і також матиме час O(h).

Попередній і наступний елементи

- У випадку вузлів з однаковими ключами попереднім і наступним вузлами можна визначити ті, що повертаються процедурами TREE_PREDECESSOR та TREE_SUCCESSOR відповідно.

Теорема. Операції пошуку, визначення мінімального і максимального елемента, а також елемента-попередника та наступника в бінарному дереві пошуку висоти h можуть бути виконані за час $O(h)$.

Вставка елемента

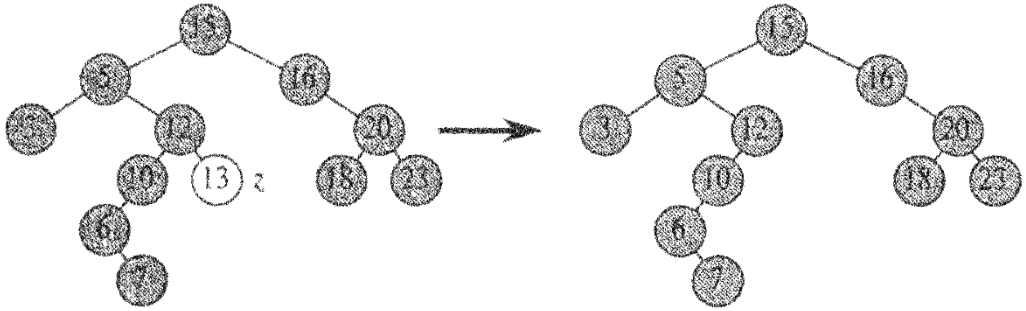
Процедура вставляє в дерево T вузол z, в якого *key*[z]=v, *left*[z]=NIL, *right*[z]=NIL.

```
TREE_INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                  // дерево T – порожнє
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

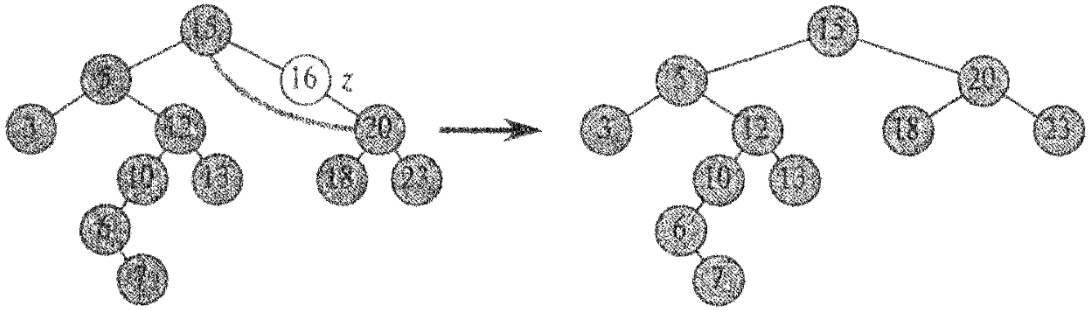
- Час виконання O(h) для дерева висоти h.

Видалення елемента

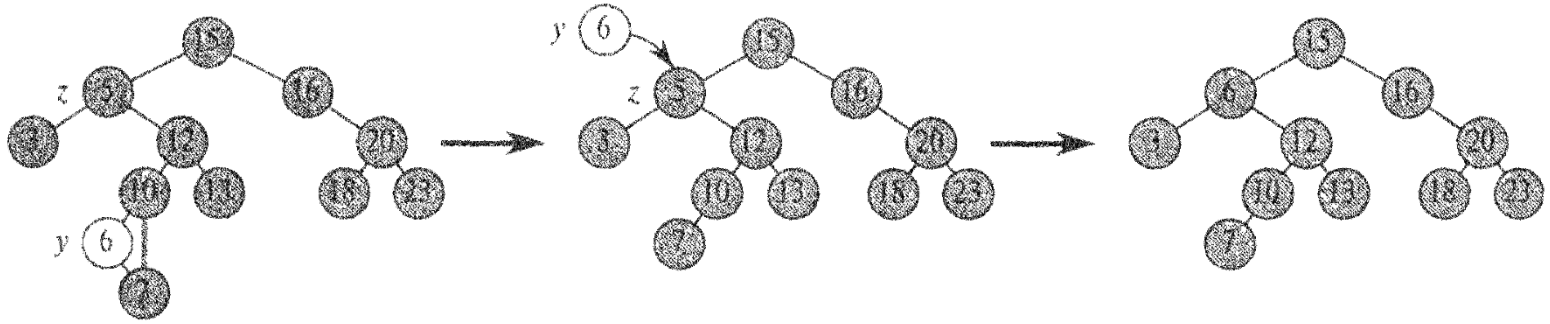
Можливі три ситуації при видаленні вузла z



а)



б)



в)

Видалення елемента

```
TREE_DELETE(T, z)
 1  if left[z] = NIL или right[z] = NIL
 2     then y ← z
 3     else y ← TREE_SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16             Копіювання супутніх даних в z
17  return y
```

- Час виконання O(h) для дерева висоти h.

Випадок однакових ключів

- (Оцініть асимптотичний час роботи процедури TREE_INSERT при вставці n однакових ключів в порожнє дерево.)
- Модифікація алгоритму TREE_INSERT:
  - перед рядком 5 додається перевірка *key*[z] = *key*[x];
  - перед рядком 11 – перевірка *key*[z] = *key*[y];
  - якщо рівності виконуються, реалізується одна з наступних стратегій

  (описи для порівняння ключів z та x)

Випадок однакових ключів

Варіанти стратегій

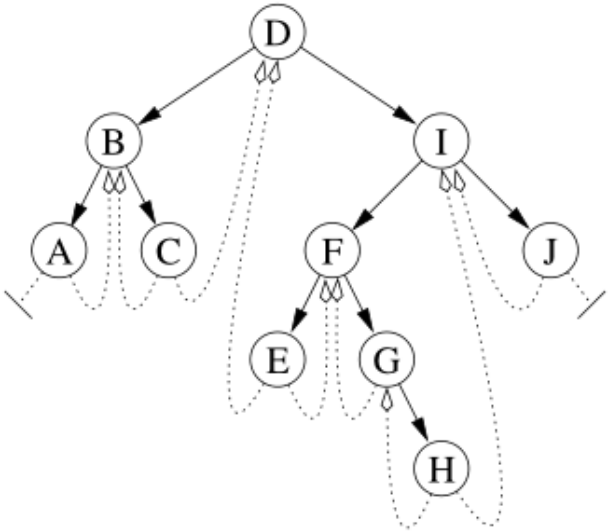
- У вузлі x зберігається логічний прапорець *b*[x], значення якого визначає, вправо чи вліво вставляється ключ зі значенням x. Значення прапорця при кожній такій вставці змінюється на протилежне.
- Всі елементи з однаковими ключами зберігаються в одному вузлі за допомогою списку.
- Дочірній вузол *left*[x] або *right*[x] вибирається випадковим чином.

Прошиті дерева

- При всіх обходах бінарних дерев використовується рекурсія, що призводить до явного чи неявного (залежно від реалізації) використання додаткової пам’яті пропорційно до висоти дерева.
- У випадку незбалансованих дерев це може мати значення.
- Існують способи здійснити обхід без рекурсії і стека.
- Один варіант – "розвертати" вказівники при спуску вниз з відновленням структури дерева на зворотному шляху, використовуючи "розвернуті" вказівники для підйому.
- Інший спосіб – використання прошитих дерев.

Прошиті дерева

- Прошивка дерева прив’язана до конкретного обходу.
- Для спрощення обходу нульові вказівники замінюються вказівниками (нитками) на попередній і/або наступний елементи при обході.
- При цьому у вузлі мають додатково з’явитися біти-маркери характера вказівника (чи це нитка?).
- Так, шукаючи наступника заданого вузла в прошитому дереві, спочатку перевіряємо, чи є його правий вказівник ниткою. Якщо так, переходимо по ній, інакше спускаємось від правого сина по лівим вказівникам до самого низу.



Збалансовані дерева

- Дерево *збалансоване*, якщо його висота гарантовано не перевищує log *n* для *n* елементів.
- Для збалансованих дерев час виконання операцій над динамічними множинами навіть в найгіршому випадку O(log *n*).
  - AVL-дерева (придумані в 1962 році в СРСР);
  - 2-3 дерева;
  - 2-3-4 дерева;
  - AA-дерева;
  - червоно-чорні дерева (RB, Red-Black);
  - … та інші.

  Насправді 2-3 і 2-3-4 дерева є різновидами загальнішої структури – B-дерев.

Червоно-чорні дерева

- Бінарне дерево пошуку з додатковим бітом кольору в кожному вузлі: червоний чи чорний.
- Дерева наближено збалансовані: жоден шлях в червоно-чорному дереві не відрізняється від іншого більше ніж удвічі.
- Кожен вузол містить поля *key, color, p*, *left*, *right*.
- Всі вузли, що містять ключ, – внутрішні.
- Розглядатимемо значення NIL як вказівники на листя (зовнішні вузли) бінарного дерева пошуку.
- Бінарне дерево пошуку буде червоно-чорним, якщо задовольнятиме червоно-чорні властивості.

Червоно-чорні дерева

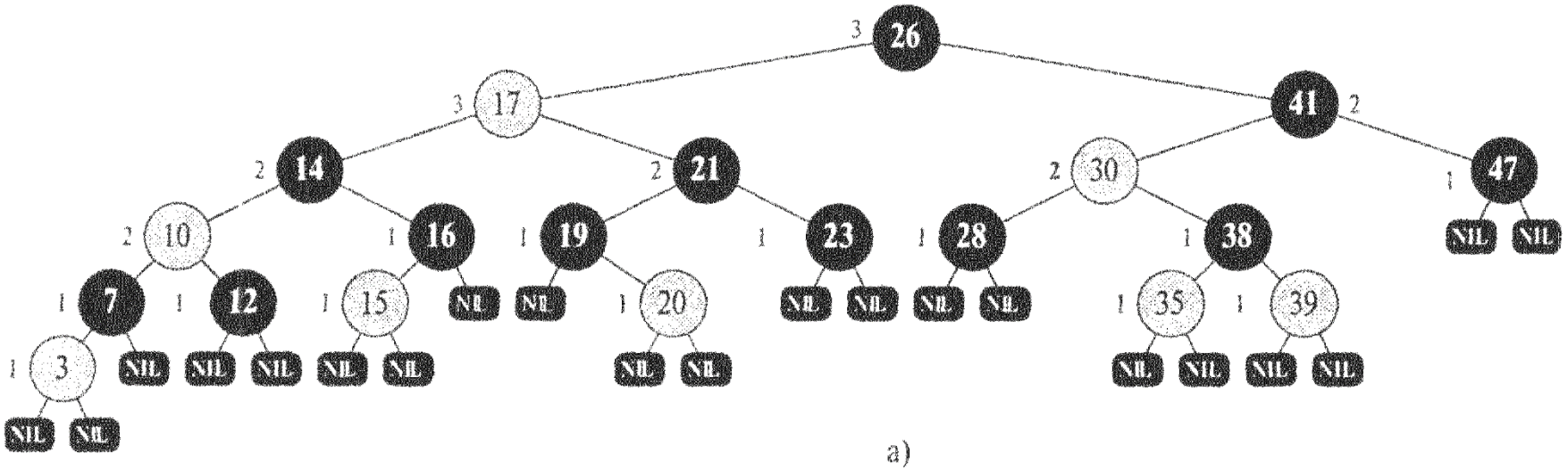
Червоно-чорні властивості

1. Кожен вузол є або червоним, або чорним.
2. Корінь дерева – чорний.
3. Кожен лист дерева (NIL) – чорний.
4. Якщо вузол червоний, то обидва його сини чорні.
5. Для кожного вузла всі шляхи від нього до листів-потомків містять однакову кількість чорних вузлів.

- *Чорна висота* вузла x (*bh*(x), black-height) – кількість чорних вузлів на шляху від вузла x (не рахуючи його самого) до листів; вона визначається однозначно.
- Чорна висота дерева – чорна висота його кореня.

Червоно-чорні дерева

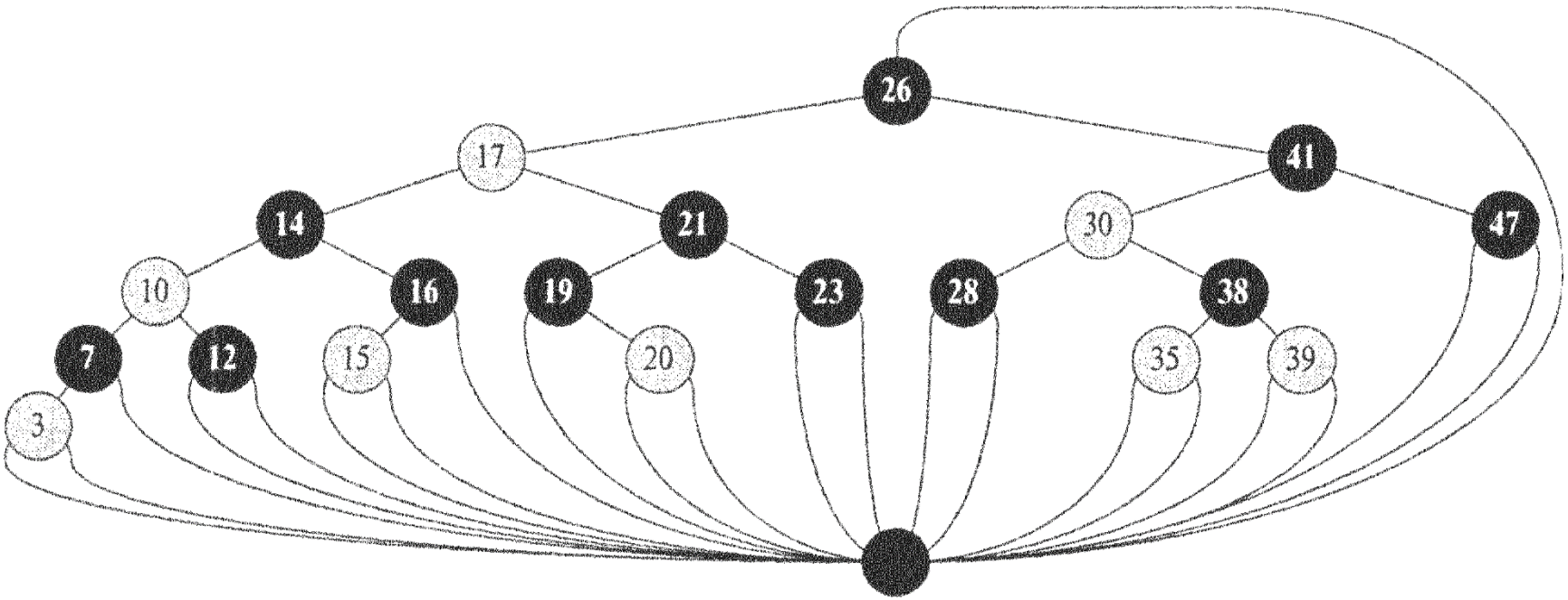
Приклад червоно-чорного дерева. Біля кожного листа вказана його чорна висота.



a)

Червоно-чорні дерева

- Для зручності роботи з червоно-чорним деревом T замінимо всі листи одним вузлом-обмежувачем *nil*[T], що представлятиме значення NIL.
- Значення його поля *color* буде чорне (BLACK).



б)

Червоно-чорні дерева

Твердження. Висота червоно-чорного дерева з n внутрішніми вузлами не перевищує $2\lg(n+1)$.

Доведення. Спочатку покажемо, що піддерево довільного вузла x містить не менше $2^{bh(x)}-1$ внутрішніх вузлів. Доводимо за індукцією.

Нехай висота x дорівнює 0. Тоді вузол є листом (*nil*[T]) і його піддерево містить не менше $2^{bh(x)}-1=2^0-1=0$ внутрішніх вузлів.

Нехай висота x додатна і вузол має два потомки. Чорна висота кожного з потомків *bh*(x) або (*bh*(x)–1), залежно від того, має він червоний чи чорний колір відповідно. Оскільки висота потомків менша за висоту самого вузла x, користуємось припущенням індукції: дерево з коренем у вершині x міститиме принаймні $(2^{bh(x)-1}-1)+$ $+(2^{bh(x)-1}-1)+1=(2^{bh(x)}-1)$ внутрішніх вузлів.

Червоно-чорні дерева

Доведення (далі).

Нехай h – висота дерева. За властивістю 4, як мінімум половина вузлів на шляху від кореня до листа, не враховуючи сам корінь, повинна бути чорними.

Тоді чорна висота кореня має бути не меншою за h/2, звідси

$$n \geq 2^{h/2} - 1.$$

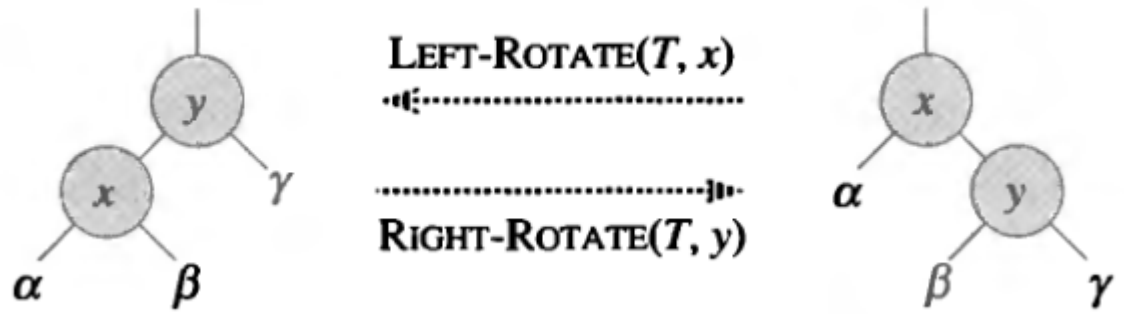
Переносимо одиницю наліво і логарифмуємо:

$$\lg(n+1) \geq h/2,$$

тобто $h \leq 2\lg(n+1)$.

- Отже, операції пошуку, взяття мінімуму, максимуму, попереднього та наступного елемента мають час O(h).

Повороти



Вважаємо, що при лівому повороті вузол x має правого потомка.

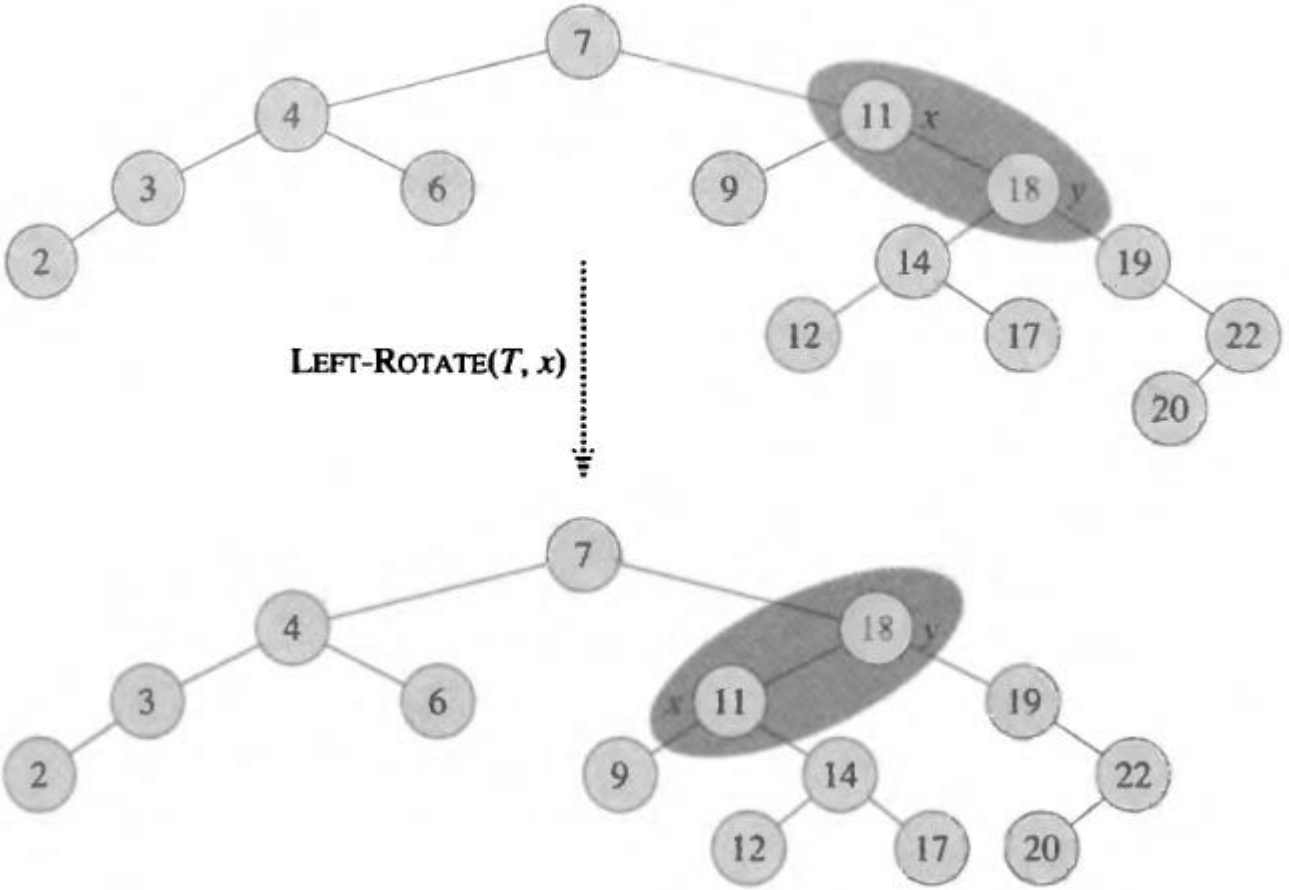
(Правий поворот симетричний.)

Час O(1)

```
LEFT_ROTATE(T, x)
 1  y ← right[x]              ▷ Устанавливаем y.
 2  right[x] ← left[y]        ▷ Левое поддерево y становится
                              ▷ правым поддеревом x
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]               ▷ Перенос родителя x в y
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x               ▷ x — левый дочерний y
12  p[x] ← y
```

Повороти

Приклад виконання лівого повороту:



Властивості дерева пошуку зберігаються.

Вставка вузла

1. Спочатку відбувається звичайна вставка як в бінарне дерево пошуку.
2. Вузол розфарбовується червоним.
3. Викликається допоміжна процедура, що відновлює червоно-чорні властивості (рух знизу вгору).

Прості випадки:

- новий вузол є коренем – перефарбовується в чорний;
- батьківський вузол чорний – властивості не порушені.

Вставка вузла

Модифікована вставка:

```
RB_INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB_INSERT_FIXUP(T, z)
```

Вставка вузла

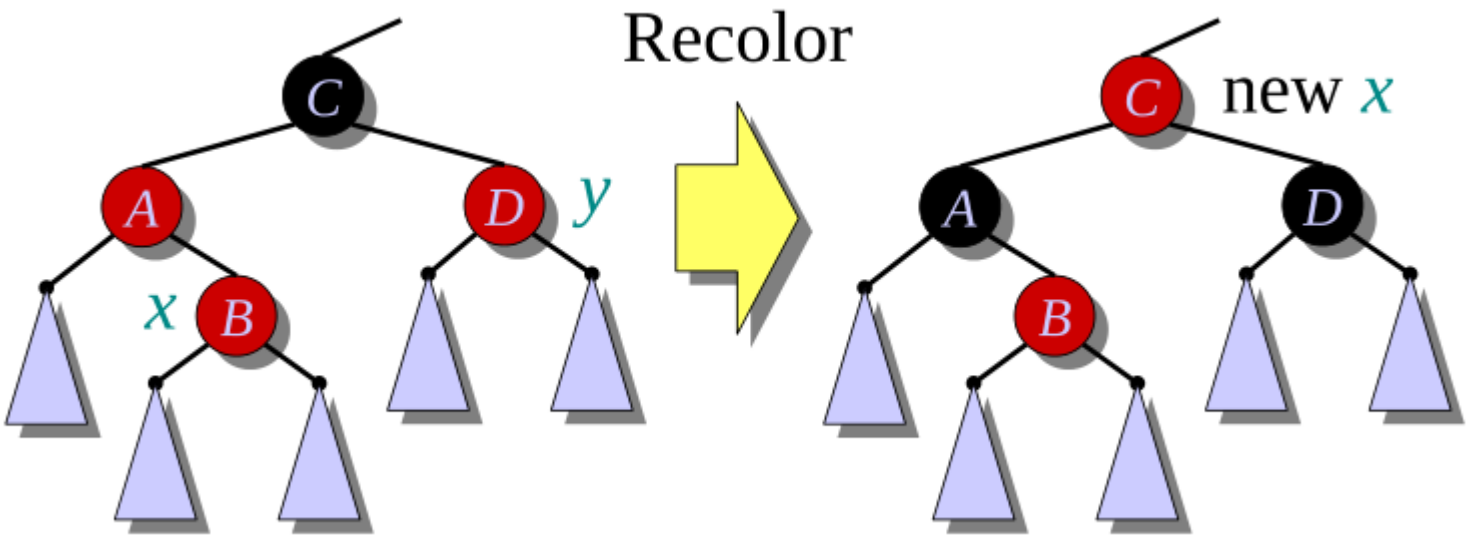
Відновлення властивостей (загалом не більше 2 обертань та lg *n* кроків):

```
RB_INSERT_FIXUP(T, z)
 1  while color[p[z]] = RED
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = RED
 5                      then color[p[z]] ← BLACK                    ▷ Случай 1
 6                           color[y] ← BLACK                       ▷ Случай 1
 7                           color[p[p[z]]] ← RED                   ▷ Случай 1
 8                           z ← p[p[z]]                            ▷ Случай 1
 9                      else if z = right[p[z]]
10                              then z ← p[z]                       ▷ Случай 2
11                                   LEFT_ROTATE(T, z)              ▷ Случай 2
12                              color[p[z]] ← BLACK                 ▷ Случай 3
13                              color[p[p[z]]] ← RED                ▷ Случай 3
14                              RIGHT_ROTATE(T, p[p[z]])            ▷ Случай 3
15              else (то же, что и в "then", с заменой
                      left на right и наоборот)
16  color[root[T]] ← BLACK
```

Вставка вузла

- Випадок 1 (або симетричний відносно А)

«Дідусь» чорний, «дядько» у червоний.

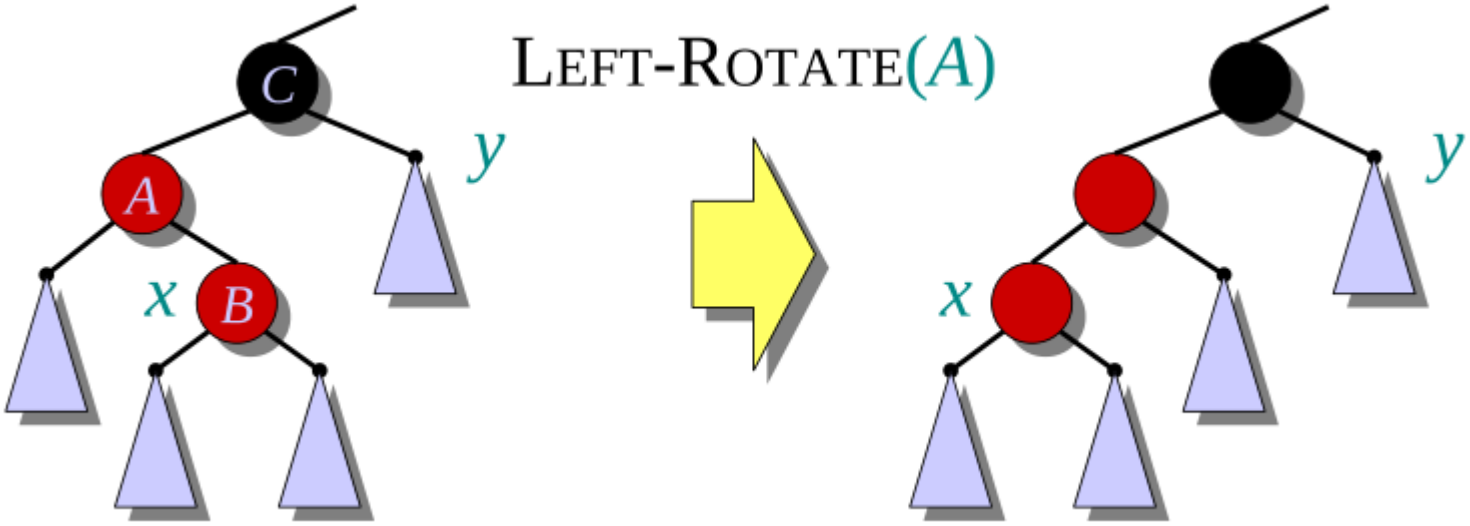


«Батько» і «дядько» стають чорними, «дідусь» червоним. Перевірка продовжується, бо предок С може виявитися червоним.

Вставка вузла

- Випадок 2

«Дядько» чорний, x є правим потомком.



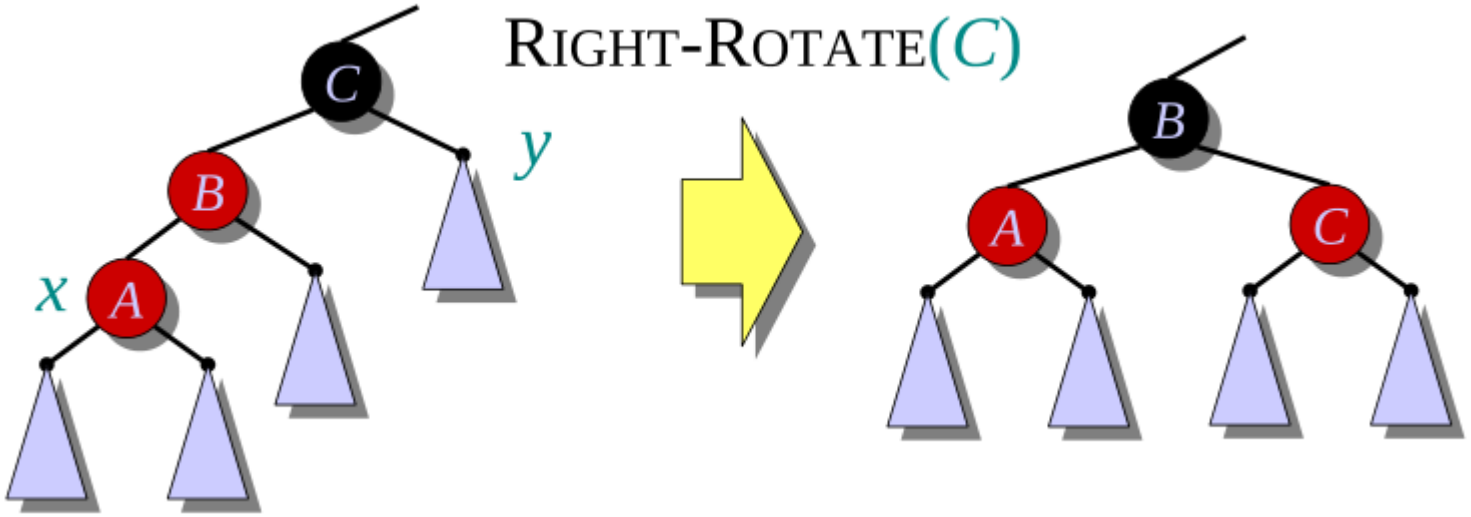
Лівий поворот відносно A.

Зведення до випадку 3.

Вставка вузла

- Випадок 3

«Дядько» чорний, x є лівим потомком.

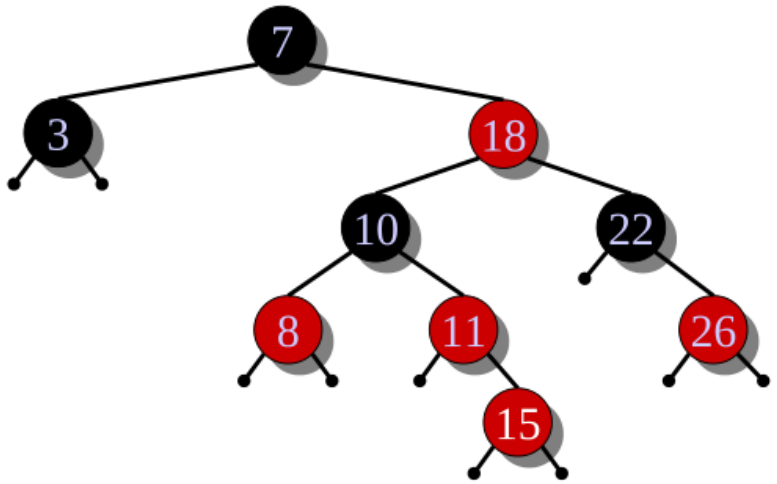


Правий поворот відносно С та зміна кольорів.

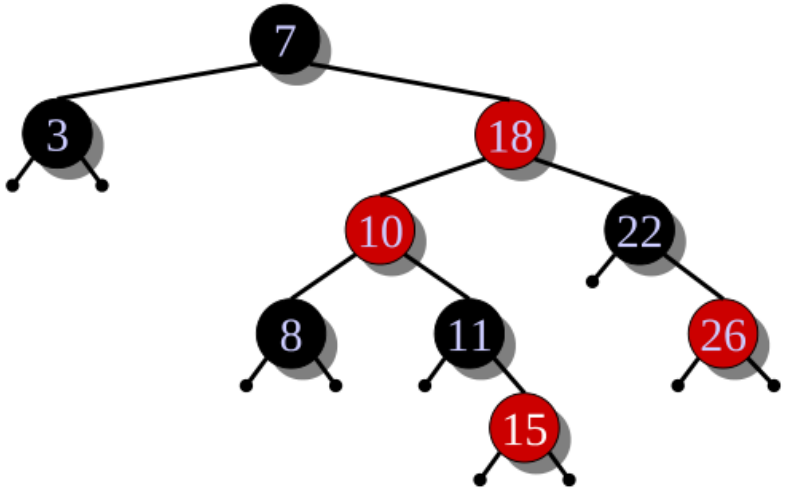
Вихід з алгоритму

Приклад вставки вузла

Додали елемент x=15.

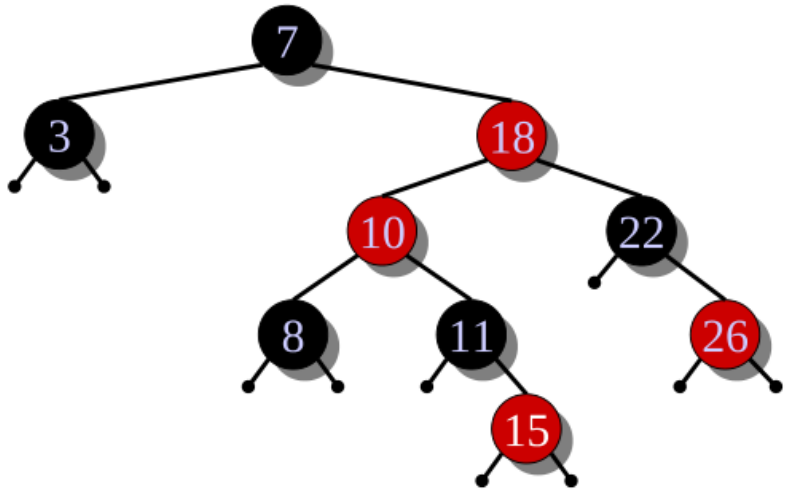


Випадок 1: міняємо кольори.

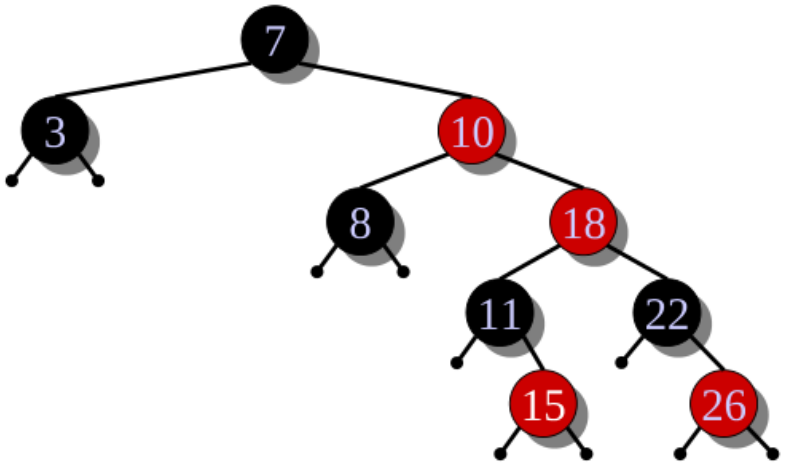


Приклад вставки вузла

Маємо:

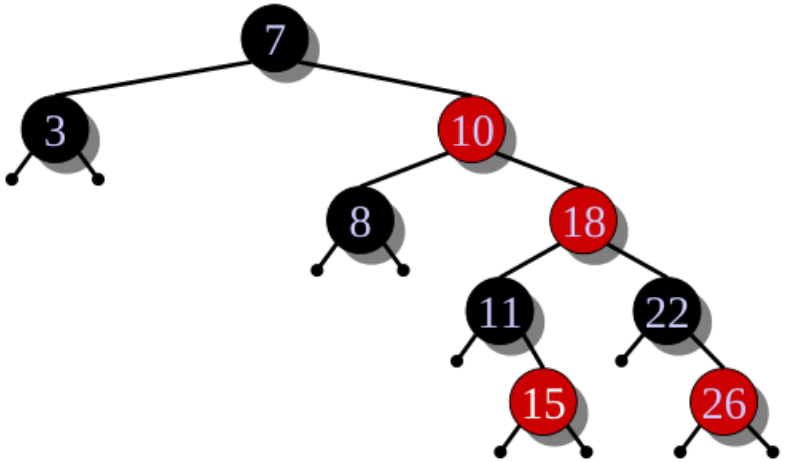


Випадок 2 (симетричний): правий поворот відносно 18.

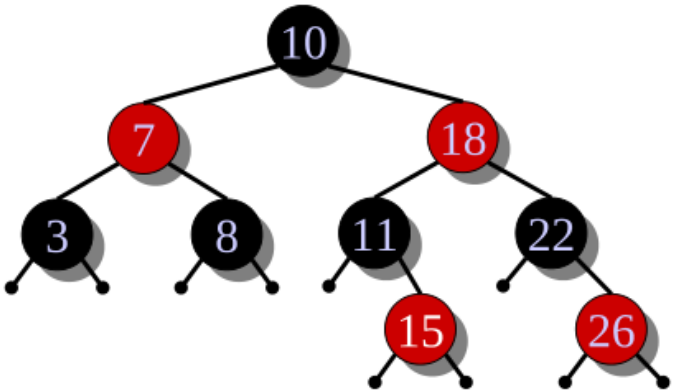


Приклад вставки вузла

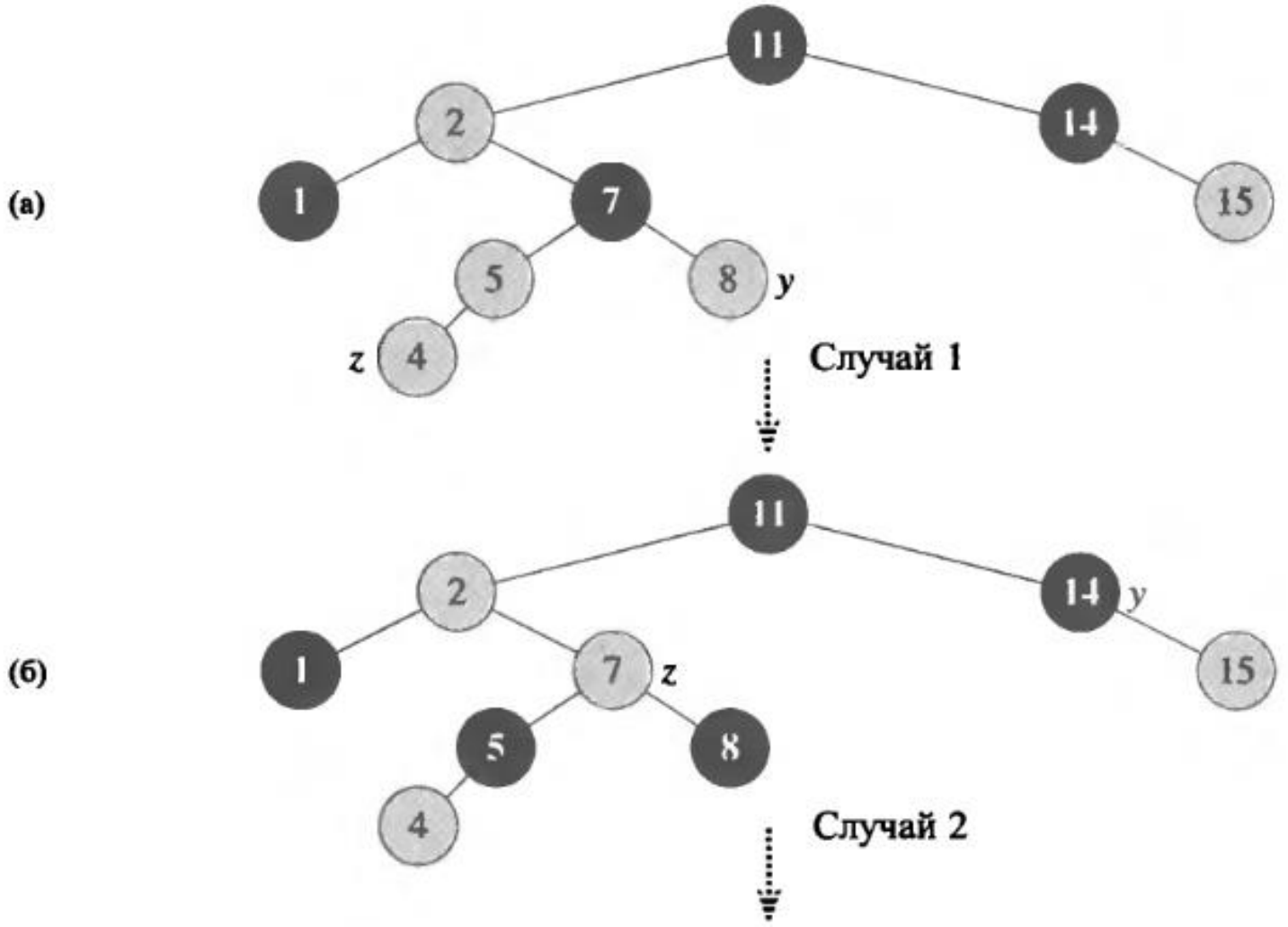
Маємо:



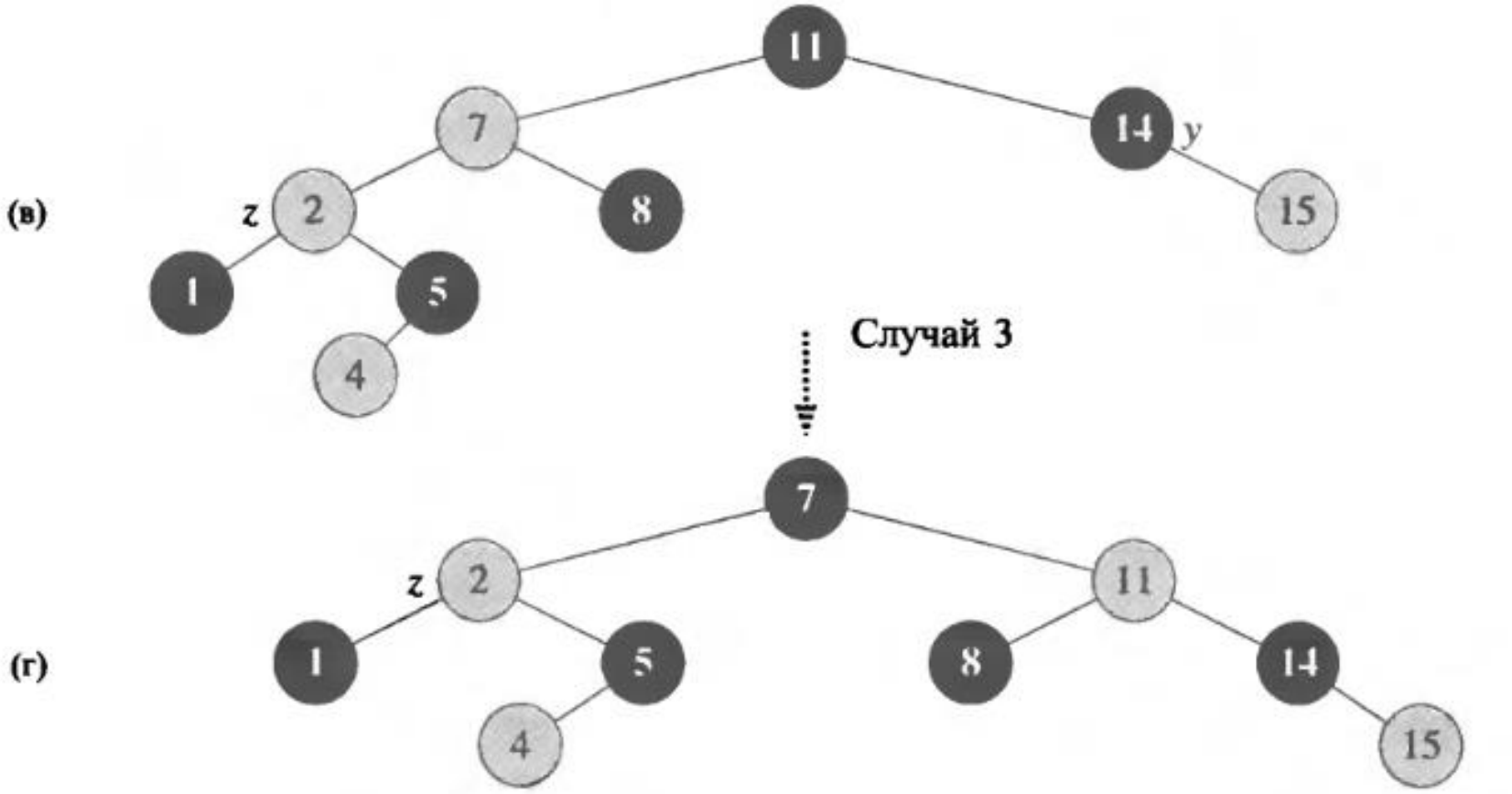
Випадок 3 (симетричний): лівий поворот відносно 7 та зміна кольору кореня на чорний.



Приклад вставки вузла



Приклад вставки вузла



Видалення вузла

```
RB_DELETE(T, z)
 1  if left[z] = nil[T] или 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           Копируем сопутствующие данные y в z
16  if color[y] = BLACK
17      then RB_DELETE_FIXUP(T, x)
18  return y
```

Видалення вузла

Можливі три випадки залежно від наявності потомків:

- Якщо у вершини немає дітей, то змінюється вказівник на неї у батька на NIL.
- Якщо у неї тільки один потомок, то у батька робиться посилання на нього замість цієї вершини.
- Якщо ж є обидва потомки, то знаходимо вершину з наступним значенням ключа. У такої вершини немає лівого дитини. Видаляємо вже цю вершину описаним вище способом, скопіювавши її ключ в початкову.

При видаленні червоної вершини червоно-чорні властивості не порушуються, тому перебалансування проводиться при видаленні чорного вузла.

В процесі видалення може бути не більше 3 обертань.

Видалення вузла

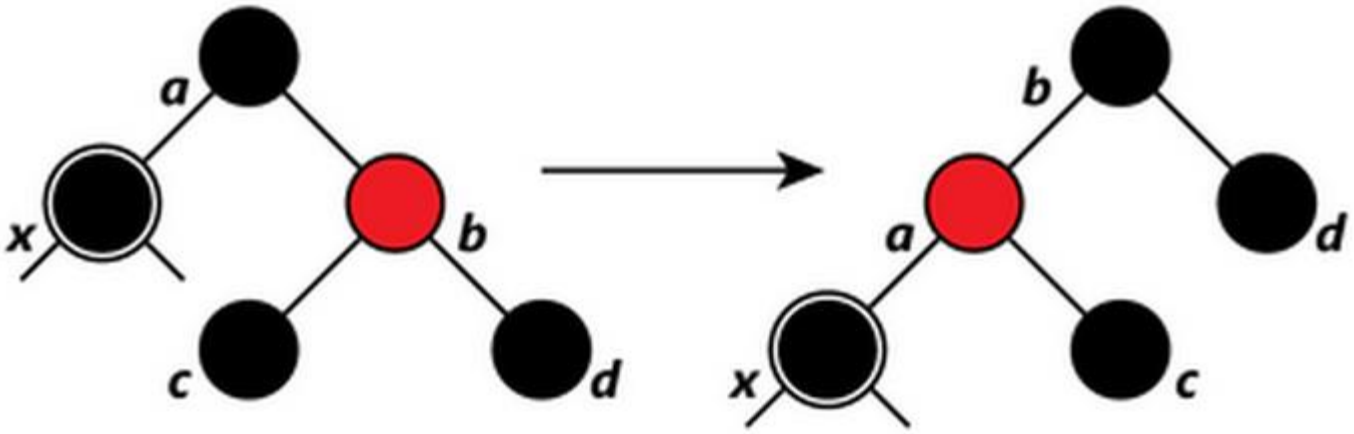
```
RB_DELETE_FIXUP(T, x)
 1  while x ≠ root[T] и color[x] = BLACK
 2        do if x = left[p[x]]
 3              then w ← right[p[x]]
 4                   if color[w] = RED
 5                      then color[w] ← BLACK                          ▷ Случай 1
 6                           color[p[x]] ← RED                         ▷ Случай 1
 7                           LEFT_ROTATE(T, p[x])                      ▷ Случай 1
 8                           w ← right[p[x]]                           ▷ Случай 1
 9                   if color[left[w]] = BLACK и color[right[w]] = BLACK
10                      then color[w] ← RED                            ▷ Случай 2
11                           x ← p[x]                                  ▷ Случай 2
12                      else if color[right[w]] = BLACK
13                              then color[left[w]] ← BLACK            ▷ Случай 3
14                                   color[w] ← RED                    ▷ Случай 3
15                                   RIGHT_ROTATE(T, w)                ▷ Случай 3
16                                   w ← right[p[x]]                   ▷ Случай 3
17                           color[w] ← color[p[x]]                    ▷ Случай 4
18                           color[p[x]] ← BLACK                       ▷ Случай 4
19                           color[right[w]] ← BLACK                   ▷ Случай 4
20                           LEFT_ROTATE(T, p[x])                      ▷ Случай 4
21                           x ← root[T]                               ▷ Случай 4
22              else (то же, что и в "then", с заменой left на right и наоборот)
23  color[x] ← BLACK
```

Видалення вузла

Розглянемо потомка x видаленого вузла.

- Випадок 1.

«Брат» вершини x червоний.



Ліве обертання відносно *a* та перефарбування.

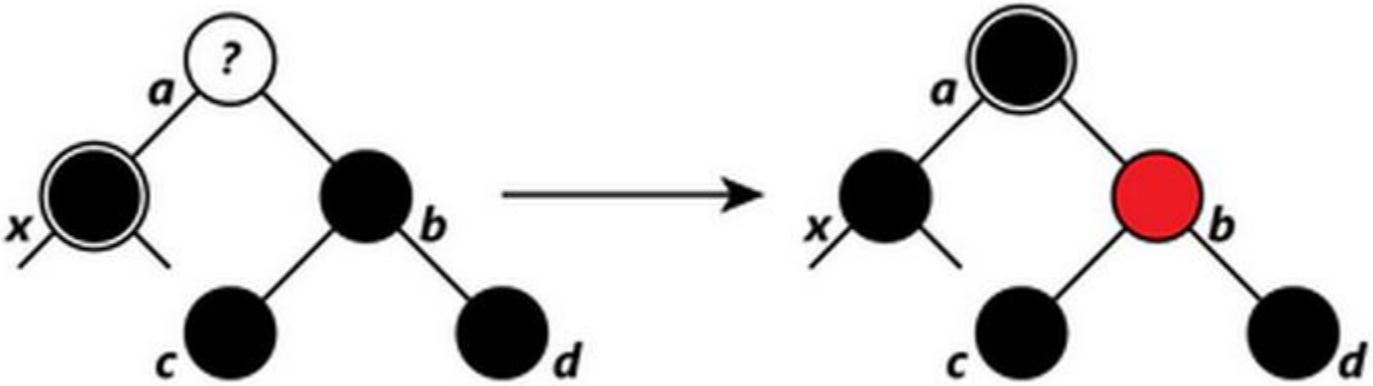
Зведення до випадків 2, 3, 4.

Видалення вузла

Випадок 2.

«Брат» вершини x чорний.

Обидва потомки «брата» чорні.



Перефарбовуємо «брата».

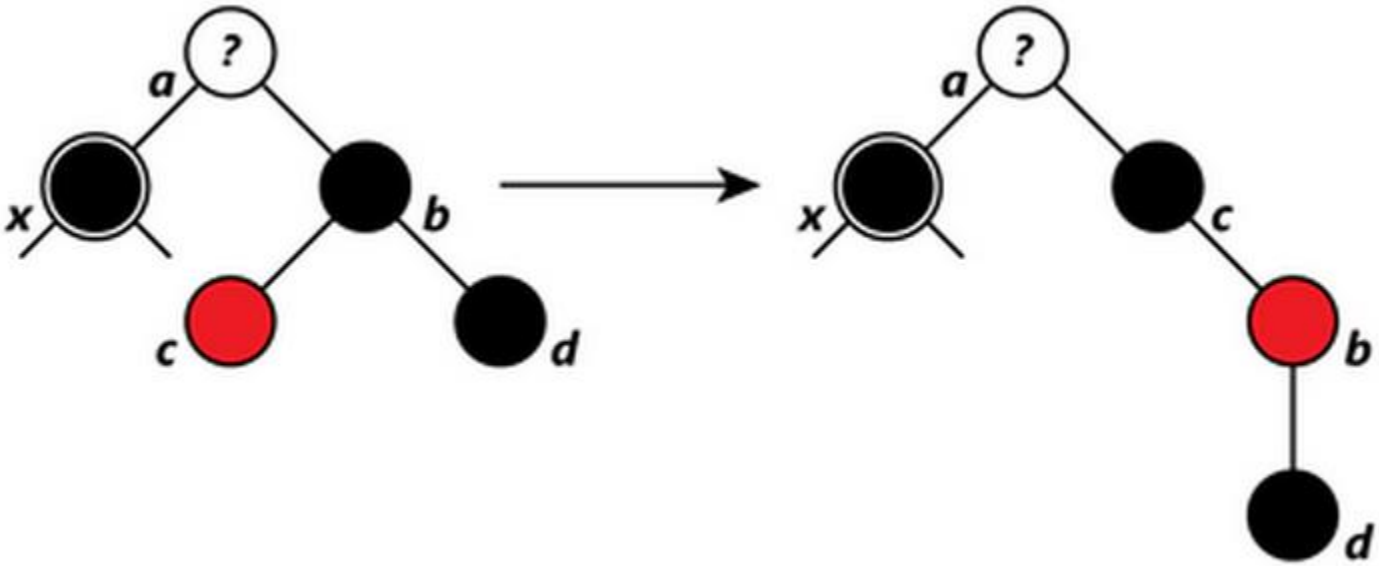
Далі розглядаємо вузла-батька.

Видалення вузла

Випадок 3.

«Брат» вершини x чорний.

Лівий потомок «брата» червоний, правий чорний.



Перефарбовуємо «брата» і його лівого сина.

Правий поворот відносно *b*.

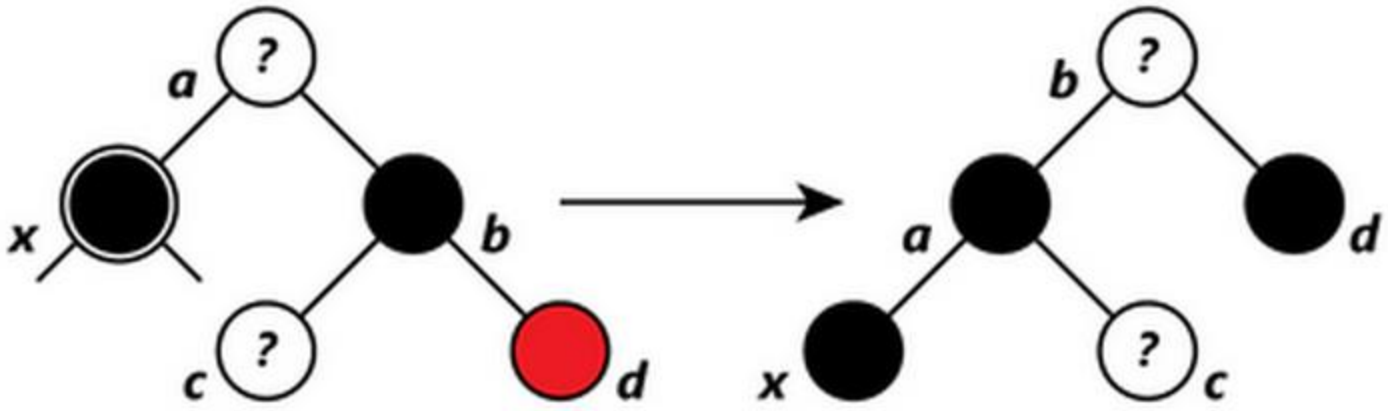
Зведення до випадку 4.

Видалення вузла

Випадок 4.

«Брат» вершини x чорний.

Правий потомок «брата» червоний.



Перефарбовуємо і робимо лівий поворот відносно a.

Вихід з алгоритму.

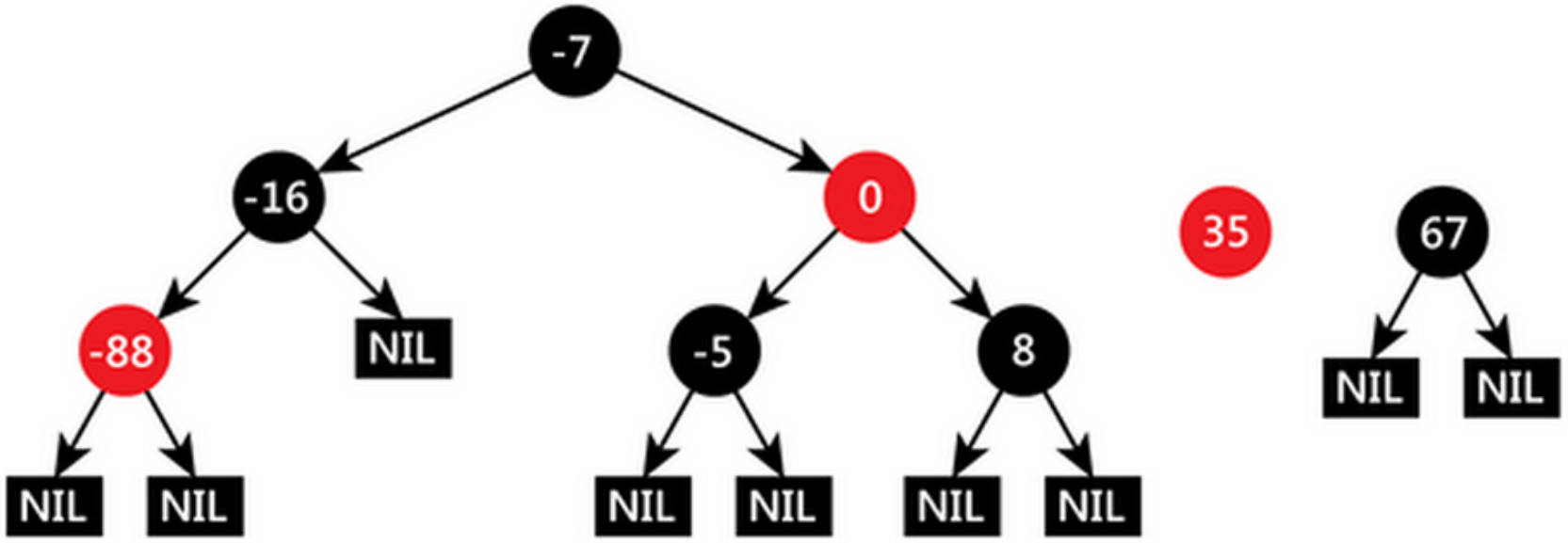
Об’єднання червоно-чорних дерев

- Операція об’єднання застосовується до двох динамічних множин T_1 і T_2 та елемента x такого, що для довільних $x_1 \in T_1$, $x_2 \in T_2$ виконується

$$key[x_1] \le key[x] \le key[x_2].$$

- Результатом операції є множина $T = T_1 \cup \{x\} \cup T_2$.
- Припустимо, що $bh(T_1) \ge bh(T_2)$. В дереві T_1 серед чорних вершин з чорною висотою $bh(T_2)$ шукаємо вузол y з найбільшим ключем.
- Нехай T_y – піддерево з коренем y. Об’єднуємо T_y з T_2 в одне дерево з червоним коренем x. Тобто батьком x стає колишній батько y.
- Відновлюємо властивості червоно-чорного дерева, як при вставці вузла.
- Всі операції виконуються за час O(lg n).

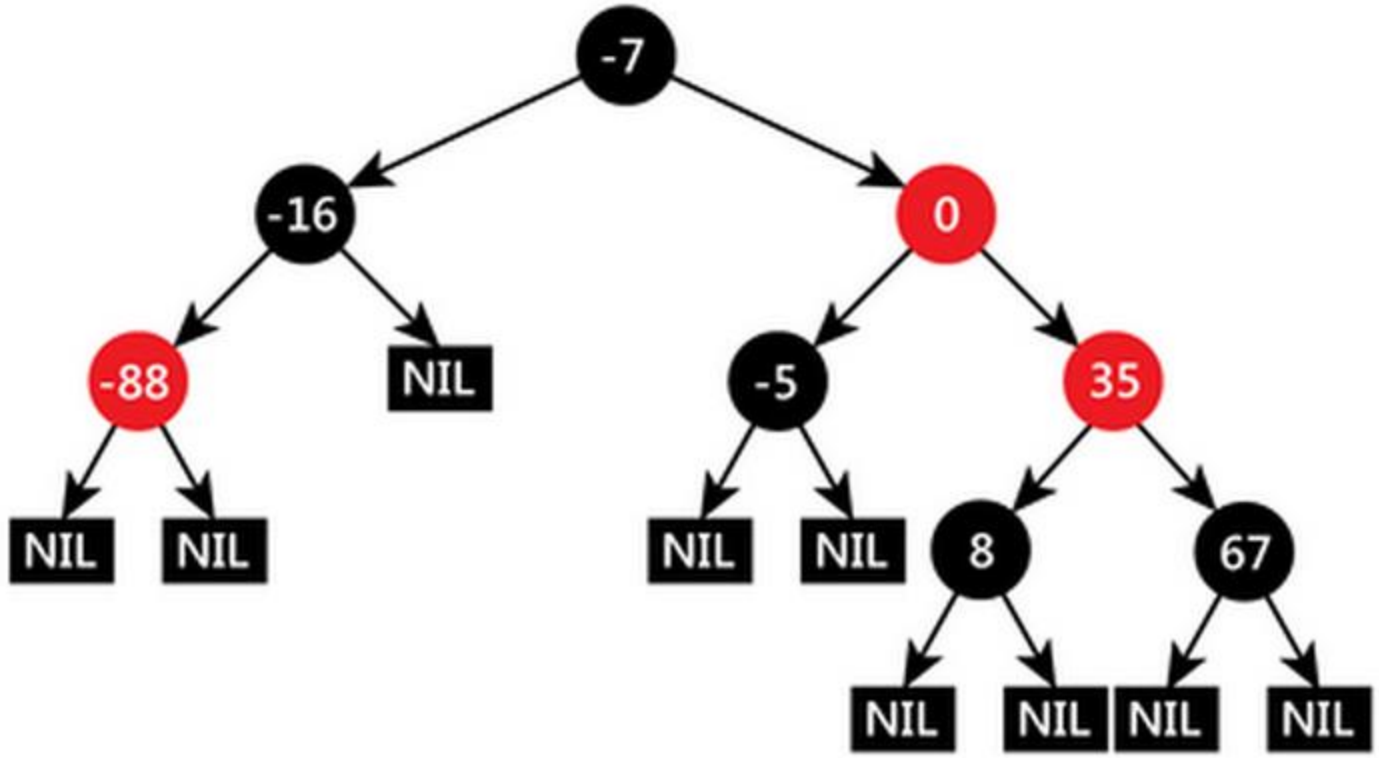
Приклад об’єднання червоно-чорних дерев



Чорна висота лівого дерева більша. Шукаємо в ньому найбільшу чорну вершину з чорною висотою 1. Це буде вузол 8.

Приклад об’єднання червоно-чорних дерев

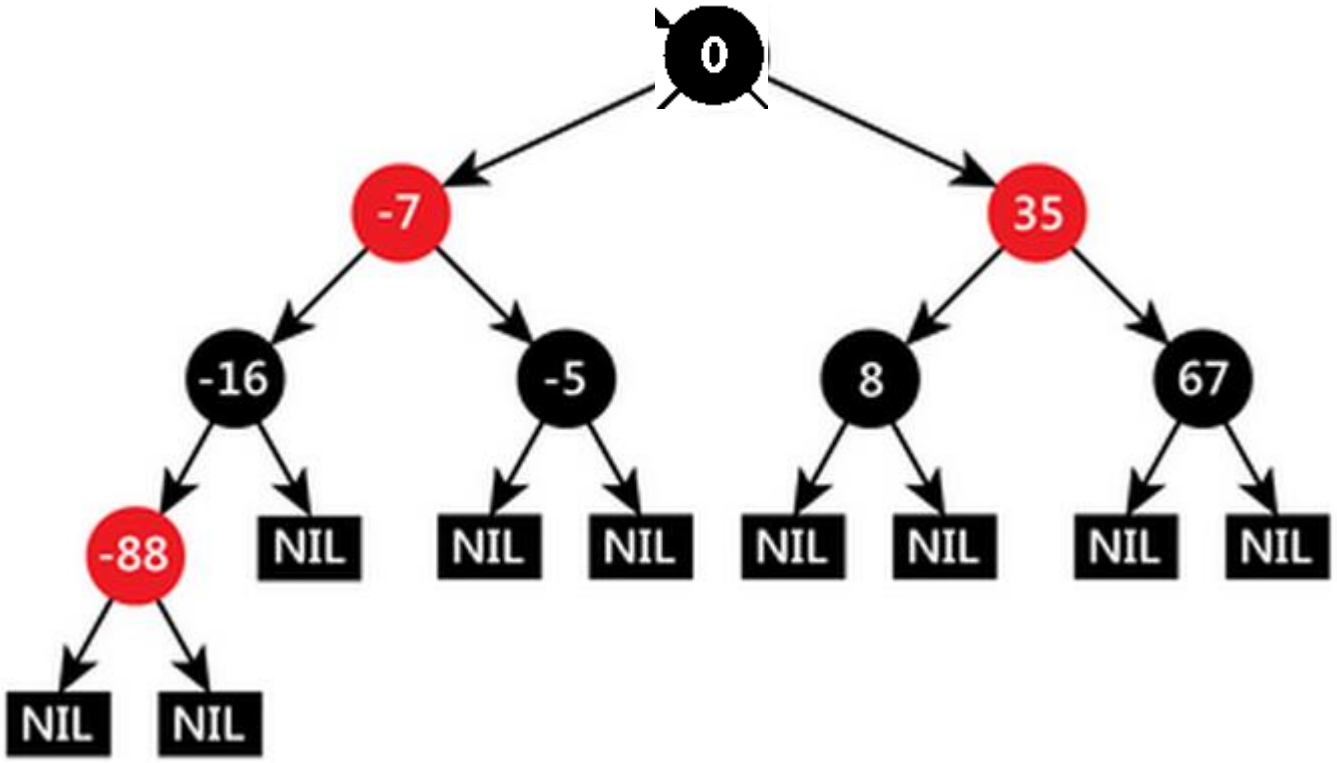
Об’єднаємо дерева і вершину:



Чи порушились червоно-чорні властивості?

Приклад об’єднання червоно-чорних дерев

Лівий поворот і перефарбування відновлять червоно-чорні властивості:



Запитання і завдання

- Розробіть нерекурсивний алгоритм симетричного обходу бінарного дерева (без використання стеку – але з перевіркою на рівність двох вказівників).
- Симетричний обхід бінарного дерева пошуку з n вузлами можна здійснити знайшовши мінімальний елемент через TREE_MINIMUM з подальшим (n–1)-м викликом процедури TREE_SUCCESSOR. Доведіть, що час роботи алгоритму Θ(n).
- Нехай T – бінарне дерево пошуку з унікальними ключами, x – лист цього дерева, y – його батьківський вузол. Покажіть, що *key*[y] або найменший ключ в дереві, що перевищить *key*[x], або найбільший ключ в T, що менший за *key*[x].
- Чи є операція видалення вузла з бінарного дерева пошуку комутативною в тому сенсі, що при видаленні спочатку вузла x, а потім y, отримаємо таке ж результуюче дерево, як при видаленні спочатку вузла y, а потім x? Обгрунтуйте відповідь.

Запитання і завдання

- Визначимо ослаблене червоно-чорне дерево як бінарне дерево пошуку, яке задовольняє червоно-чорним властивостям 1, 3, 4 та 5. Іншими словами, корінь може бути як чорним, так і червоним. Візьмемо ослаблене червоно-чорне дерево, корінь якого червоний. Якщо ми перефарбуємо корінь на чорний, чи отримаємо червоно-чорне дерево?
- Покажіть, що найдовший шлях від вершини x до листа червоно-чорного дерева має довжину, що не більше ніж удвічі перевищує найкоротший шлях від x до листа.
- Чому дорівнює найбільше можливе число внутрішніх вузлів у червоно-чорному дереві з чорною висотою k? А найменше можливе число?

Запитання і завдання

- Опишіть червоно-чорне дерево з n ключами з найбільшим можливим відношенням кількості червоних внутрішніх вузлів до кількості чорних внутрішніх вузлів. Чому дорівнює це відношення? Яке дерево має найменше вказане відношення і чому дорівнює його величина?
- Покажіть, що довільне бінарне дерево пошуку з n вузлами може бути перетворене в будь-яке інше бінарне дерево пошуку з n вузлами з використанням O(n) поворотів. (Спочатку покажіть, що (n–1) правих поворотів достатньо для перетворення дерева в правий ланцюжок.)
- Розглянемо червоно-чорне дерево, утворене вставкою n вузлів за допомогою процедури RB_INSERT. Доведіть, що якщо n>1, то в дереві є як мінімум один червоний вузол.
- Припустимо, що вузол x вставлений в червоно-чорне дерево за допомогою процедури RB_INSERT, після чого одразу видалений. Чи буде отримане в результаті вставки і видалення червоно-чорне дерево таким самим, як і вихідне? Обгрунтуйте відповідь.