Лекція 24

Червоно-чорні дерева

Бінарні дерева пошуку висоти h реалізують усі базові операції над динамічними множинами (Search, Predecessor, Successor, Minimum, Maximum, Insert і Delete) за час $O(h)$. Таким чином, операції виконуються тим швидше, чим менше висота дерева. Однак у найгіршому випадку продуктивність бінарного дерева пошуку виявляється нітрохи не краще, ніж продуктивність зв'язаного списку. Червоно-чорні дерева являють собою одну із збалансованих схем дерев пошуку, що гарантують час виконання операцій над динамічною множиною $O(h)$ навіть у найгіршому випадку.

Властивості червоно-чорних дерев

Червоно-чорне дерево являє собою бінарне дерево пошуку з одним додатковим бітом ***кольору*** в кожнім вузлі. Колір вузла може бути або червоним, або чорним. Відповідно до обмежень, що накладаються на вузли дерева, шлях у червоно-чорному дереві не відрізняється від іншого по довжині більш раза у два рази, червоно-чорні ***дерева*** є приблизно збалансованими.

Кожен вузол дерева містить поля *color*, *key*, *left*, *right* і *p*. Якщо не існує дочірнього чи батьківського вузла стосовно даного, відповідний вказівник приймає значення NULL. Ми будемо розглядати ці значення NULL як вказівники на зовнішні вузли (листи) бінарного дерева пошуку. При цьому всі “нормальні” вузли, що містять поле ключа, стають внутрішніми вузлами дерева.

Бінарне дерево пошуку є червоно-чорним деревом, якщо воно задовольняє наступним ***червоно-чорним властивостям***.

1. Кожен вузол є червоним чи чорним.
2. Корінь дерева є чорним.
3. Кожен лист дерева (NULL) є чорним.
4. Якщо вузол — червоний, то обоє його дочірніх вузла — чорні.
5. Для кожного вузла всі шляхи від нього до листів, що є нащадками даного вузла, містять те саме кількість чорних вузлів.

На рис. 24.1*а* показаний приклад червоно-чорного дерева. На малюнку чорні вузли показані темним кольором, червоні — світлим. Біля кожного вузла показана його "чорна" висота. У всіх листів вона, саме собою зрозуміло, дорівнює 0.

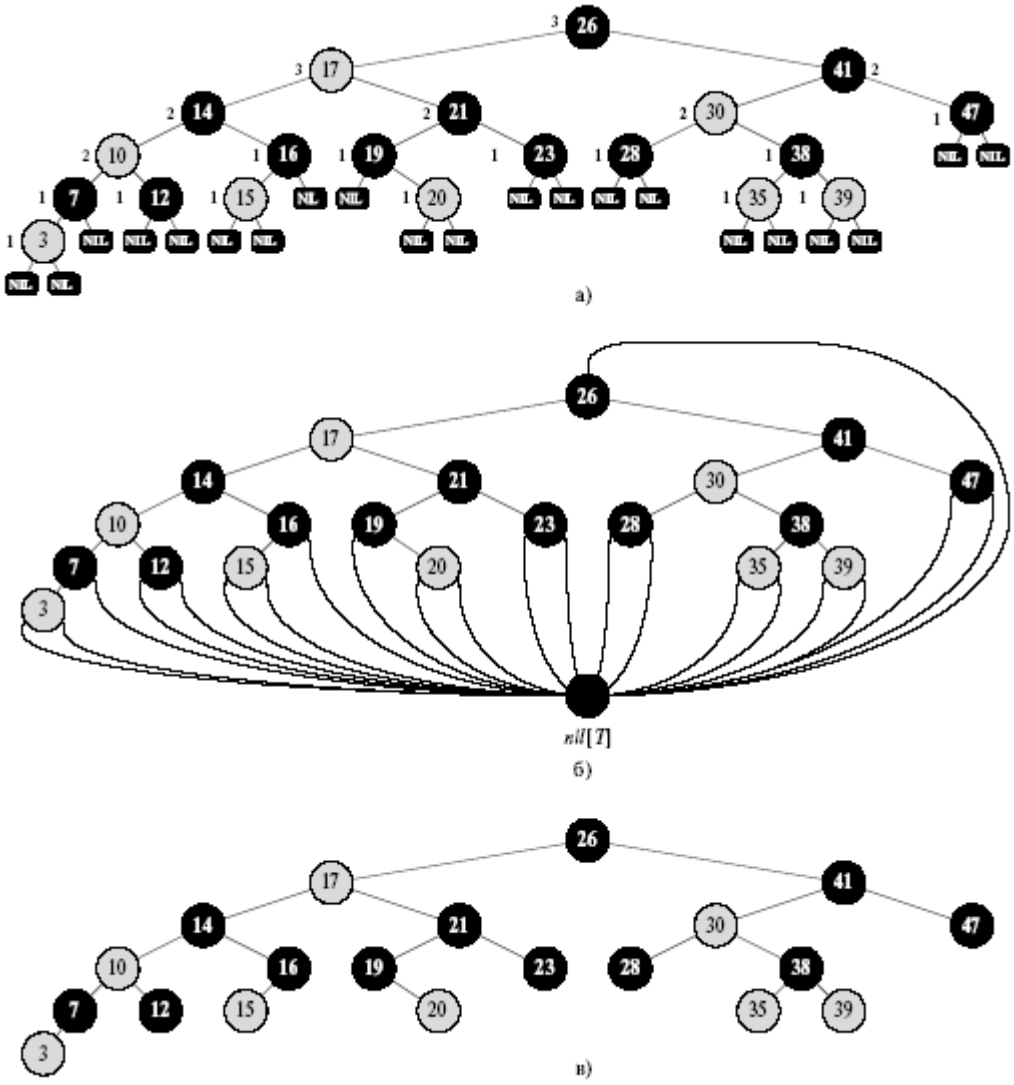

Рис. 24.1. Приклад червоно-чорного дерева

Для зручності роботи з червоно-чорним деревом ми замінимо всі листи одним обмежуючим вузлом, що представляє значення NULL. У червоно-чорному дереві T обмежник $nil[T]$ являє собою об'єкт із тими ж полями, що і звичайний вузол дерева. Значення *color* цього вузла дорівнює BLACK (чорний), а всі інші поля можуть мати довільні значення. Як показано на рис. 24.1*б*, усі вказівники на NULL заміняються вказівником на обмежник $nil[T]$.

Використання обмежника дозволяє нам розглядати дочірній стосовно вузла x NULL як звичайний вузол, батьком якого є вузол x. Хоча можна було б використовувати

різні обмежники для кожного значення NULL (що дозволило б точно визначати з батьківські вузли), цей підхід привів би до невиправданої перевитрати пам'яті. Замість цього ми використовуємо єдиний обмежник для представлення всех NULL — як листів, так і батьківського вузла кореня. Величини полів *p*, *left*, *right* і *key* обмежника не грають ніякої ролі, хоча для зручності ми можемо привласнити їм те чи інші значення.

У цілому ми обмежимо наш інтерес до червоно-чорних дерев лише їх внутрішніми вузлами, оскільки лише вони зберігають значення ключів. У частині даної глави, що залишилася, при зображенні червоно-чорних дерев усі листи опускаються, як це зроблено на рис. 24.1.*в*.

Кількість чорних вузлів на шляху від вузла x (не вважаючи сам вузол) до листа будемо називати ***чорною висотою*** вузла (black-height) і позначати як $\mathrm{bh}(x)$. У відповідності з властивістю 5 червоно-чорних дерев, чорна висота вузла — точно обумовлене значення. Чорною висотою дерева будемо вважати чорну висоту його кореня.

Наступна лема показує, чому червоно-чорні дерева добре використовувати як дерева пошуку.

Лема 24.1. Червоно-чорне дерево з n *внутрішніми вузлами має висоту не більше ніж* $2\lg(n+1)$.

Безпосереднім наслідком леми є те, що такі операції над динамічними множинами, як Search, Minimum, Maximum, Predecessor і Successor, при використанні червоно-чорних дерев виконуються за час $O(\lg h)$, оскільки час роботи цих операцій на дереві пошуку висотою h складає $O(h)$, а будь-яке червоно-чорне дерево з n вузлами є деревом пошуку висотою $O(\lg n)$. Хоча алгоритми Tree_Insert і Tree_Delete з лекції 9 і характеризуються часом роботи $O(\lg n)$, якщо використовувати їх для вставки і видалення з червоно-чорного дерева, безпосередньо використовувати їх для виконання операцій Insert і Delete не можна, оскільки вони не гарантують збереження червоно-чорних властивостей.

Повороти

Операції над деревом пошуку Tree_Insert і Tree_Delete, застосовані до червоно-чорного дерева з n ключами, виконуються за час $O(\lg n)$. Оскільки вони змінюють дерево, у результаті їх роботи можуть порушуватися червоно-чорні властивості. Для відновлення цих властивостей ми повинні змінити колір деяких вузлів дерева, а також структуру його вказівників.

Зміни в структурі вказівників будуть виконуватися за допомогою ***поворотів*** (rotations), що являють собою локальні операції в дереві пошуку, що зберігають властивість бінарного дерева пошуку. На рис. 24.2 показані два типи поворотів — лівий і правий (тут α, β і γ — довільні піддерева). При виконанні лівого повороту у вузлі x передбачається, що його правий дочірній вузол y не є листом $nil[T]$. Лівий поворот виконується навколо зв'язку між x і y, роблячи y новим коренем поддерева, лівим дочірнім вузлом якого стає x, а колишній лівий нащадок вузла y — правим нащадком x.

У псевдокоді процедури Left_Rotate передбачається, що $right[x] \neq nil[T]$, а батько кореневого вузла — $nil[T]$.

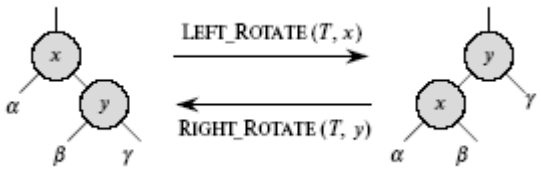

Рис. 24.2. Операції повороту в бінарному дереві пошуку

```
Left_Rotate(T,x)
 1  y ← right[x]        // Установлюємо y
 2  right[x] ← left[y] // Ліве піддерево y стає
                       // правим піддеревом x
 3  p[left[y]] ← x
 4  p[y] ← p[x]         // Перенос батька x у y
 5  if p[x] = NULL[T]
 6       then root[T] ← y
 7       else if x = left[p[x]]
 8            then left[p[x]] ← y
 9            else right[p[x]] ← y
```

```
10  left[y] ← x        // x - лівий дочірній y
11  p[x] ← y
```

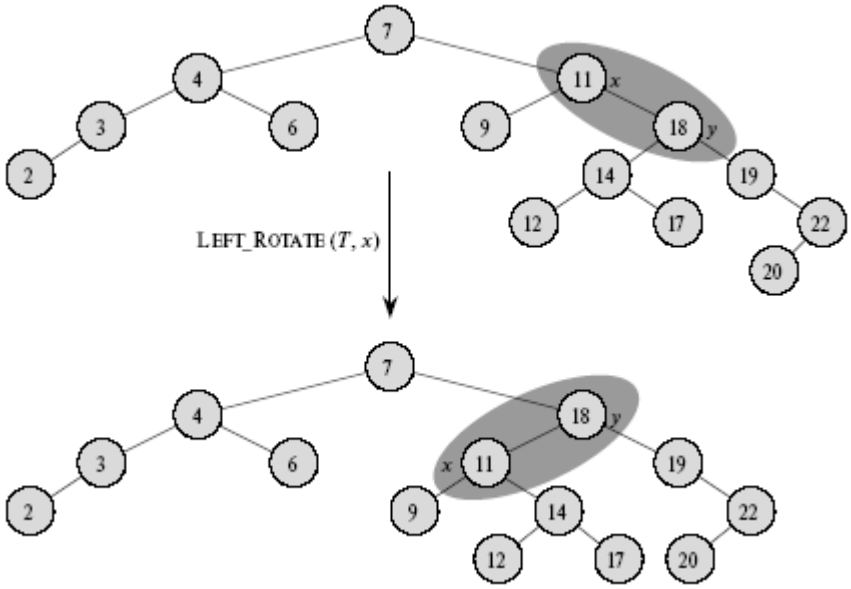

Рис. 24.3. Приклад виконання процедури Left_Rotate

На рис. 24.3 показаний конкретний приклад виконання процедури Left_Rotate. Код процедури Right_Rotate симетричний коду Left_Rotate. Обидві ці процедури виконуються за час $O(1)$. При повороті змінюються тільки вказівники, всі інші поля зберігають своє значення.

Вставка

Вставка вузла в червоно-чорне дерево з n вузлами може бути виконана за час $O(\lg n)$. Для вставки вузла z у дерево T ми використовуємо дещо модифіковану версію процедури Tree_Insert, що вставляє вузол у дерево, як ніби це було звичайне бінарне дерево пошуку, а потім зафарблює його в червоний колір. Для того щоб вставка зберігала червоно-чорні властивості дерева, після неї викликається допоміжна процедура RB_Insert_Fixup, що перефарбовує вузли і виконує повороти. Виклик RB_Insert(T,z) вставляє в червоно-чорне дерево T вузол z із заповненим полем *key*:

```
RB_Insert(T,z)
 1  y ← NULL[T]
 2  x ← root[T]
 3  while x ≠ NULL[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 = NULL[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] ← NULL[T]
15   right[z] ← NULL[T]
16   color[z] ← RED
17   RB_Insert_Fixup(T,z)
```

Є чотири відмінності процедури Tree_Insert від процедури RB_Insert. По-перше, усі NULL у Tree_Insert замінені на $nil[T]$. По-друге, для підтримки коректності структури дерева в рядках 14–15 процедури RB_Insert виконується присвоєння $nil[T]$ вказівникам $left[z]$ і $right[z]$. У третіх, у рядку 16 ми призначаємо вузлу z червоний колір. І нарешті, оскільки червоний колір z може викликати порушення одного з червоно-чорних властивостей, у рядку 17 викликається допоміжна процедура RB_Insert_Fixup(T, z), призначення якого — відновити червоно-чорні властивості дерева:

```
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       // Випадок 1
 6                          color[y] ← BLACK          // Випадок 1
 7                          color[p[p[z]]] ← RED      // Випадок 1
 8                          z ← p[p[z]]               // Випадок 1
 9                     else if z = right[p[z]]
10                               then z ← p[z]         // Випадок 2
11                                    Left_Rotate(T,z) // Випадок 2
12                          color[p[z]] ← BLACK       // Випадок 3
13                          color[p[p[z]]] ← RED      // Випадок 3
14                          Right_Rotate(T,p[p[z]]) // Випадок 3
15           else (то ж, що й у "then", із заміною
```

```
                left на right і навпаки)
16  color[root[T]] ← BLACK
```

Для того щоб зрозуміти, як працює процедура RB_Insert_Fixup, ми розіб'ємо розгляд коду на три основні частини. Спочатку ми визначимо, які з червоно-чорних властивостей порушуються при вставленні вузла *z* і фарбуванні його в червоний колір. Потім ми визначимо призначення циклу **while** у рядках 1–15. Після цього ми вивчимо кожний із трьох випадків (не взаємовиключних), що зустрічаються в цьому циклі, і подивимося, яким чином досягається мета в кожнім випадку. На рис. 24.3 наведений приклад виконання процедури RB_Insert_Fixup.

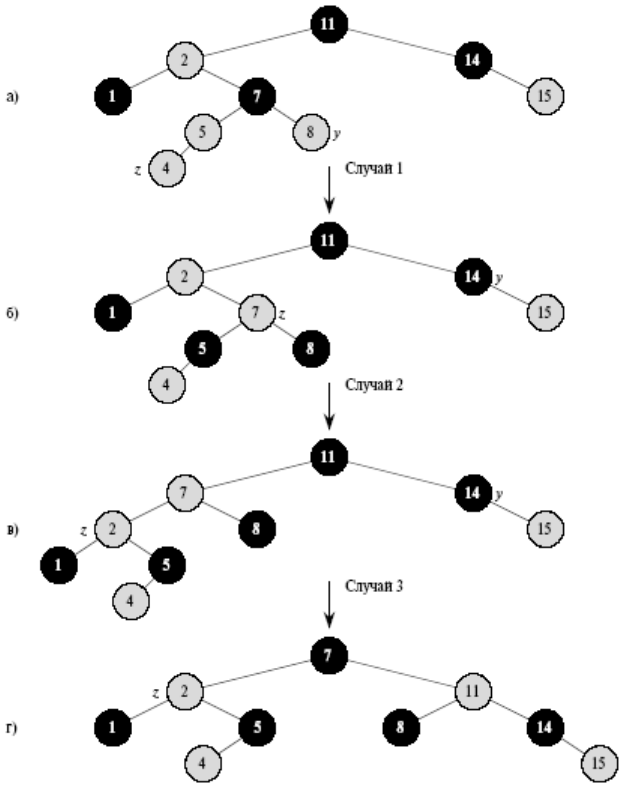

Рис. 24.4. Робота процедури RB_Insert_Fixup

Які з червоно-чорних властивостей можуть бути порушені перед викликом RB_Insert_Fixup? Властивість 1 виконується, як і властивість 3, тому що обоє дочірніх вузла, що вставляються, є обмежниками $nil[T]$. Властивість 5, відповідно до якої для кожного вузла всі шляхи від нього до листів, що є нащадками даного вузла, містять ту саму кількість чорних вузлів, також залишається в силі, оскільки вузол *z* заміщає (чорний) обмежник, будучи при цьому червоним і маючи чорні дочірні вузли. Таким чином, може порушуватися тільки властивість 2, що вимагає, щоб корінь

червоно-чорного дерева була чорним, і властивість 4, відповідно до якої червоний вузол не може мати червоного нащадка. Обоє порушення можливі внаслідок того, що вузол z після вставки зафарблюється в червоний колір. Властивість 2 виявляється порушеною, якщо вузол z стає коренем, а властивість 4 — якщо батьківський стосовно z вузол є червоним. На рис. 24.4*а* показане порушення властивості 4 після вставки вузла z.

Цикл **while** у рядках 1–15 зберігає наступний інваріант, що складається з трьох частин.

На початку кожної ітерації циклу:

à) вузол z червоний;

á) якщо $p[z]$ — корінь дерева, то $p[z]$ — чорний вузол;

â) якщо мається порушення червоно-чорних властивостей, то це порушення лише одне — або порушення властивості 2, або властивості 4. Якщо порушена властивість 2, то це викликано тим, що коренем дерева є червоний вузол z; якщо порушене властивість 4, то в цьому випадку червоними є вузли z і $p[z]$.

Частина *в)*, у якій говориться про можливі порушення червоно-чорних властивостей, найбільш важлива для того, щоб показати, що процедура RB_Insert_Fixup відновлює червоно-чорні властивості. Частини *а*) і *б)* просто пояснюють ситуацію. Оскільки ми зосереджуємо свій розгляд лише на вузлі z і вузлах, що знаходяться в дереві поблизу його, корисно знати, що вузол z — червоний (частина *а*). Частина *б)* використовується для того, щоб показати, що вузол $p[p[z]]$, до якого ми звертаємося в рядках 2, 3, 7, 8, 13 і 14, існує.

Згадаємо, що ми повинні показати, що інваріант циклу виконується перед першою ітерацією циклу, що будь-яка ітерація циклу зберігає інваріант і що інваріант циклу забезпечує виконання необхідної властивості по закінченні роботи циклу.

Почнемо з розгляду ініціалізації і завершення роботи циклу, а потім, докладніше розглянувши роботу циклу, ми доведемо, що він зберігає інваріант циклу. Попутно ми покажемо, що є тільки два можливих варіанти дій у кожній ітерації циклу —

вказівник z переміщається нагору чи в дереві виконуються деякі повороти і цикл завершується.

Ініціалізація. Перед виконанням першої ітерації циклу маємо червоно-чорне дерево без будь-яких порушень червоно-чорних властивостей, до якого ми додаємо червоний вузол z. Покажемо, що всі частини інваріанта циклу виконуються до моменту виклику процедури RB_Insert_Fixup.

à) У момент виклику процедури RB_Insert_Fixup вузол z — вставлений у дерево червоний вузол.

á) Якщо $p[z]$ — кореневий вузол дерева, то він є чорним і не змінюється до виклику процедури RB_Insert_Fixup.

â) Ми уже переконалися в тім, що червоно-чорні властивості 1, 3 і 5 зберігаються до моменту виклику процедури RB_Insert_Fixup.

Якщо порушується властивість 2, то червоний корінь повинний бути доданим у дерево вузлом z, що при цьому є єдиним внутрішнім вузлом дерева. Оскільки і батько, і обоє нащадка z є обмежниками, властивість 4 не порушується. Таким чином, порушення властивості 2 — єдине порушення червоно-чорних властивостей у всім дереві.

Якщо ж порушена властивість 4, то оскільки дочірні стосовно z вузли є чорними обмежниками, а до вставки z ніяких порушень червоно-чорних властивостей у дереві не було, порушення полягає в тім, що і z, і $p[z]$ — червоні. Крім цього, інших порушень червоно-чорних властивостей немає.

Завершення. Як видно з коду, цикл завершує свою роботу, коли $p[z]$ стає чорним (якщо z — кореневий вузол, то $p[z]$ являє собою чорний обмежник $nil[T]$). Таким чином, властивість 4 при завершенні циклу не порушується. Відповідно до інваріанта циклу, єдиним порушенням червоно-чорних властивостей може бути порушення властивості 2. У рядку 16 ця властивість відновлюється, так що по завершенні роботи процедури RB_Insert_Fixup усі червоно-чорні властивості дерева виконуються.

Збереження. При роботі циклу **while** варто розглянути шістьох різних випадків, однак три з них симетричні другим трьом; різниця лише в тім, чи є батько $p[z]$ лівим

чи правим дочірнім вузлом стосовно свого батька $p\left[p[z]\right]$, що і з'ясовується в рядку 2 (ми навели код лише для ситуації, коли $p\left[z\right]$ є лівим нащадком). Вузол $p\left[p[z]\right]$ існує, оскільки, відповідно до частини *б)* інваріанта циклу, якщо $p\left[z\right]$ — корінь дерева, то він чорний. Оскільки цикл починає роботу, лише якщо $p\left[z\right]$ — червоний, то $p\left[z\right]$ не може бути коренем. Отже, $p\left[p[z]\right]$ існує.

Випадок 1 відрізняється від випадків 2 і 3 кольором "брата" батьківського стосовно z вузла, тобто "дядька" вузла z. Після виконання рядка 3 вказівник y указує на дядька вузла z — вузол $right\left[p\left[p[z]\right]\right]$, і в рядку 4 з'ясовується його колір. Якщо y — червоний, виконується код для випадку 1; у супротивному випадку виконується код для випадків 2 і 3. У будь-якому випадку, вузол $p\left[p[z]\right]$ — чорний, оскільки вузол $p\left[z\right]$ — червоний, а властивість 4 порушується тільки між z і $p\left[z\right]$.

Випадок 1: вузол *y* червоний.

На рис. 24.5 показана ситуація, що виникає у випадку 1 (рядка 5–8), коли і $p\left[z\right]$, і y — червоні вузли. Оскільки $p\left[p[z]\right]$ — чорний, ми можемо виправити ситуацію, пофарбувавши і $p\left[z\right]$, і y у чорний колір (після чого колір червоного батька вузла z стає чорним, і порушення між z і його батьком зникає), а $p\left[p[z]\right]$ — у червоний колір, для того щоб виконувалася властивість 5. Після цього ми повторюємо цикл **while** з вузлом $p\left[p[z]\right]$ як новий вузол z. Вказівник z переміщається, таким чином, на два рівні нагору.

На рис. 24.5*а* z — правий дочірній вузол, а на рис. 24.5*б* — лівий. Як бачите, що починаються в обох випадках ті самі дії приводять до однакового результату. Усі піддерева (α , β, γ і δ) мають чорні корені й однакове значення чорної висоти. Після перефарбування властивість 5 зберігається: усі спадні шляхи від вузла до листів містять однакове число чорних вузлів. Після виконання ітерації новим вузлом z стає вузол $p\left[p[z]\right]$, і порушення властивості 4 може бути тільки між новим вузлом z і його батьком (який також може виявитися червоним).

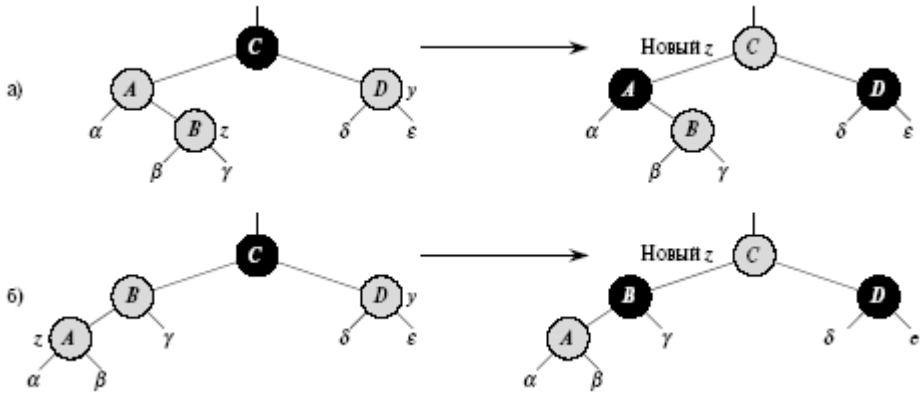

Рис. 24.5. Випадок 1 процедури RB_Insert_Fixup

Тепер покажемо, що у випадку 1 інваріант циклу зберігається. Позначимо через z вузол z у поточній ітерації, а через $z' = p\left[p[z]\right]$ — вузол z, що перевіряється в рядку 1 при наступній ітерації.

à) Оскільки в даній ітерації колір вузла $p\left[p[z]\right]$ стає червоним, на початку наступної ітерації вузол z' — червоний.

á) Вузол $p\left[z'\right]$ у поточній ітерації — $p\left[p\left[p[z]\right]\right]$, і колір даного вузла в межах даної ітерації не змінюється. Якщо це кореневий вузол, то його колір до початку даної ітерації був чорним і залишається таким на початку наступної ітерації.

â) Ми уже довели, що у випадку 1 властивість 5 зберігається; крім того, зрозуміло, що при виконанні ітерації не виникає порушення властивостей 1 чи 3.

Якщо вузол z' на початку чергової ітерації є коренем, то код, що відповідає випадку 1, коректує єдине порушення властивості 4. Оскільки вузол z' — червоний і кореневий, єдиною порушеною стає властивість 2, до того ж це порушення зв'язане з вузлом z' .

Якщо вузол z' на початку наступної ітерації не є коренем, то код, що відповідає випадку 1, не викликає порушення властивості 2. Цей код коректує єдине порушення властивості 4, наявне перед виконанням ітерації. Корекція виражається в тім, що вузол z' стає червоним, у той час

як колір вузла $p[z']$ не змінюється. Якщо вузол $p[z']$ був чорним, то властивість 4 не порушується; якщо ж цей вузол був червоним, то фарбування вузла z' у червоний колір приводять до порушення властивості 4 між вузлами z' і $p[z']$.

Випадок 2: вузол *y* чорний і *z* — правий нащадок.

Випадок 3: вузол *y* чорний і *z* — лівий нащадок.

У випадках 2 і 3 колір вузла *y*, що є "дядьком" вузла *z*, чорний. Ці два випадки відрізняються друг від друга тим, що *z* є лівим чи правим дочірнім вузлом стосовно батьківського. Рядка 10–24 псевдокоду відповідають випадку 2, що показаний на рис. 24.5 разом з випадком 3. У випадку 2 вузол *z* є правим нащадком свого батьківського вузла. Ми використовуємо лівий поворот для перетворення сформованої ситуації у випадок 3 (рядка 12–14), коли *z* є лівим нащадком. Оскільки і *z*, і $p[z]$ — червоні вузли, поворот не впливає ані на чорну висоту вузлів, ані на виконання властивості 5. Коли ми приходимо до випадку 3 (або безпосередньо, або поворотом з випадку 2), вузол *y* має чорний колір (оскільки інакше ми б одержали випадок 1). Крім того, обов'язково існує вузол $p[p[z]]$, тому що ми довели, що цей вузол існував при виконання рядків 2 і 3, а також що після переміщення вузла *z* на один вузол нагору в рядку 10 з наступним опусканням у рядку 24 вузол $p[p[z]]$ залишається незмінним. У випадку 3 ми виконуємо ряд змін кольору і правих поворотів, що зберігають властивість 5. Після этого, тому що в нас немає двох червоних вузлів, що йдуть поспіль, робота процедури завершується. Більше тіло циклу **while** не виконується, тому що вузол $p[z]$ тепер чорний.

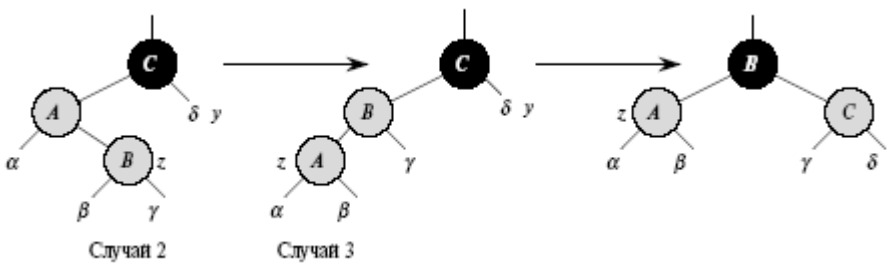

Рис. 24.6. Випадки 2 і 3 процедури RB_Insert_Fixup

Тепер покажемо, що випадки 2 і 3 зберігають інваріант циклу. (Як ми щойно довели, перед наступною перевіркою в рядку 1 вузол $p[z]$ буде чорним і тіло циклу більше виконуватися не буде.)

à) У випадку 2 виконується присвоєння, після якого z указує на червоний вузол $p[z]$. Ніяких інших змін z чи його кольору у випадках 2 і 3 не виконується.

á) У випадку 3 вузол $p[z]$ робиться чорним, так що якщо $p[z]$ на початку наступної ітерації є коренем, то цей корінь — чорний.

â) Як і у випадку 1, у випадку 2 і 3 властивості 1, 3 і 5 зберігаються.

Оскільки вузол z у випадках 2 і 3 не є коренем, порушення властивості 2 невозможно. Випадки 2 і 3 не можуть приводити до порушення властивості 2, оскільки при повороті у випадку 3 червоний вузол стає дочірнім стосовно чорного.

Таким чином, випадки 2 і 3 приводять до корекції порушення властивості 4, при этом не вносячи ніяких нових порушень червоно-чорних властивостей.

Показавши, що при будь-якій ітерації інваріант циклу зберігається, ми тим самим показали, що процедура RB_Insert_Fixup коректно відновлює червоно-чорні властивості дерева.

Аналіз

Чому дорівнює час роботи процедури RB_Insert? Оскільки висота червоно-чорного дерева з n вузлами дорівнює $O(\lg n)$, виконання рядків 1—16 процедури RB_Insert вимагає $O(\lg n)$ часу. У процедурі RB_Insert_Fixup цикл **while** повторно виконується тільки у випадку 1, і в цьому випадку вказівник z переміщається нагору по дереву на два рівні. Таким чином, загальна кількість можливих виконань тіла циклу **while** дорівнює $O(\lg n)$. Таким чином, загальний час роботи процедури RB_Insert дорівнює $O(\lg n)$. Цікаво, що в ній ніколи не виконується більше двох поворотів, оскільки цикл **while** у випадках 2 і 3 завершує роботу.

Видалення

Як і інші базові операції над червоно-чорними деревами з n вузлами, видалення вузла виконується за час $O(\lg n)$. Видалення виявляється дещо більш складною задачею, ніж вставка.

Процедура RB_Delete являє собою трохи змінену процедуру Tree_Delete. Після видалення вузла в ній викликається допоміжна процедура RB_Delete_Fixup, що змінює кольори і виконує повороти для відновлення червоно-чорних властивостей дерева:

```
RB_Delete(T,z)
 1  if left[z] = NULL[T] or right[z] = NULL[T]
 2       then y ← z
 3       else y ← Tree_Successor(z)
 4  if left[y] ≠ NULL[T]
 5       then x ← left[y]
 6       else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = NULL[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            Копіюємо супутні дані y у z
 16  if color [y] = BLACK
 17       then RB_Delete_Fixup(T,x)
 24  return y
```

Існує три розходження між процедурами Tree_Delete і RB_Delete. По-перше, усі посилання на NULL у Tree_Delete замінені в RB_Delete посиланнями на обмежник $nil[T]$. По-друге, вилучена перевірка в рядку 7 процедури Tree_Delete (чи дорівнює x NULL), і присвоєння $p[x] \leftarrow p[y]$ виконується в процедурі RB_Delete безумовно. Таким чином, якщо x є обмежником $nil[T]$, то його вказівник на батька вказує на батька витягнутого з дерева вузла y. По-третє, у рядках 16–17 процедури RB_Delete у випадку, якщо вузол y — чорний, виконується виклик допоміжної процедури

RB_Delete_Fixup. Якщо вузол y — червоний, червоно-чорні властивості при витягу y з дерева зберігаються в силу наступних причин:

- жлдна чорна висота в дереві не змінюється;
- жодні червоні вузли не стають сусідніми;
- оскільки y не може бути коренем внаслідок свого кольору, корінь залишається чорним.

Вузол x, що передається як параметр у допоміжну процедуру RB_Delete_Fixup, є одним із двох вузлів: або це вузол, що був єдиним нащадком y перед витягом останнього з дерева (якщо в y був дочірній вузол, що не є обмежником), або, якщо у вузла y немає дочірніх вузлів, x є обмежником $nil[T]$. В останньому випадку безумовне присвоєння в рядку 7 гарантує, що батьківським стосовно x вузлом стає вузол, що раніше був батьком y, незалежно від того, чи є x внутрішнім вузлом з реальним чи ключем обмежником $nil[T]$.

Тепер ми можемо подивитися, як допоміжна процедура RB_Insert_Fixup розв'язує свою задачу відновлення червоно-чорних властивостей дерева пошуку:

```
RB_Delete_Fixup(T,x)
 1  while x ≠ root[T] and 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  // Випадок 1
 6                          color[p[x]] ← RED // Випадок 1
 7                          Left_Rotate(T,p[x])   // Випадок 1
 8                          w ← right[p[x]]  // Випадок 1
 9                if color[left[w]] = BLACK and
                     color[right[w]] = BLACK
10                       then color[w] ← RED   // Випадок 2
24                            x ← p[x]        // Випадок 2
12                       else if color[right[w]] = BLACK
13                            then color[left[w]] ← BLACK // Випадок 3
14                                 color[w] ← RED    // Випадок 3
15                                 Right_Rotate(T,w) // Випадок 3
16                                 w ← right[p[x]]  // Випадок 3
17                            color[w] ← color[p[x]]     // Випадок 4
```

```
24                      color[p[x]] ← BLACK     // Випадок 4
19                      color[right[w]] ← BLACK     // Випадок 4
20                      Left_Rotate(T,p[x])     // Випадок 4
21                      x ← root[T]       // Випадок 4
22          else (то ж, що й у "then", із заміною
                 left на right і навпаки)
23  color[x] ← BLACK
```

Якщо з дерева в процедурі RB_Delete витягається чорний вузол y, то можуть виникнути три проблеми. По-перше, якщо y був коренем, а тепер коренем став червоний нащадок y, порушується властивість 2. По-друге, якщо і x, і $p[y]$ (який тепер є також $p[x]$) були червоними, то порушується властивість 4. І, по-третє, видалення y приводить до того, що всі шляхи, що проходили через y, тепер мають на один чорний вузол менше. Таким чином, для всіх предків y виявляється порушеною властивість 5. Ми можемо виправити ситуацію, затверджуючи, що вузол x — понадчорний, тобто при розгляді будь-якого шляху, що проходить через x, додавати додаткову одиницю до кількості чорних вузлів. При такій інтерпретації властивість 5 залишається дійсною. При видаленні чорного вузла y ми передаємо його "чорність" його нащадку. Проблема полягає в тім, що тепер вузол x не є ни чорним, ни червоним, що порушує властивість 1. Замість цей вузол x пофарбований або "двічі чорним", або "червоно-чорним" кольором, що дає при підрахунку чорних вузлів на шляху, що містить x, внесок, рівний, відповідно, 2 чи 1. Атрибут *color* вузла x при цьому залишається дорівнює або BLACK (якщо вузол двічі чорний), або RED (якщо вузол червоно-чорний). Іншими словами, колір вузла не відповідає його атрибуту *color*.

Процедура RB_Delete_Fixup відновлює властивості 1, 2 і 4. Дана процедура відновлює властивості 2 і 4, так що в частині даного розділу, що залишилася, ми сконцентруємо свою увагу на властивості 1. Цель циклу **while** у рядках 1–22 полягає в тому, щоб перемістити додаткову чорність нагору по дереву доти, поки не виконається одна з перерахованих умов.

1. x указує на червоно-чорний вузол — у цьому випадку ми просто робимо вузол x "однократно чорним" у рядку 23.
2. x указує на корінь — у цьому випадку ми просто забираємо зайву чорність.

3. Можна виконати деякі повороти і перефарбування, після яких подвійна чорність буде усунута.

Усередині циклу **while** *x* завжди вказує на двічі чорну вершину, що не є коренем. У рядку 2 ми визначаємо, чи є *x* лівим чи правим дочірнім вузлом свого батька $p[x]$. Далі приведений докладний код для ситуації, коли *x* — лівий нащадок. Для правого нащадка код аналогічний і симетричний, і схований за описом у рядку 22. Вказівник *w* указує на другого нащадка батька *x*. Оскільки вузол *x* двічі чорний, вузол *w* не може бути $nil[T]$ — у противному випадку кількість чорних вузлів на шляху від $p[x]$ до (одкократно чорного) листу було б менше, ніж кількість чорних вузлів на шляху від $p[x]$ до *x*.

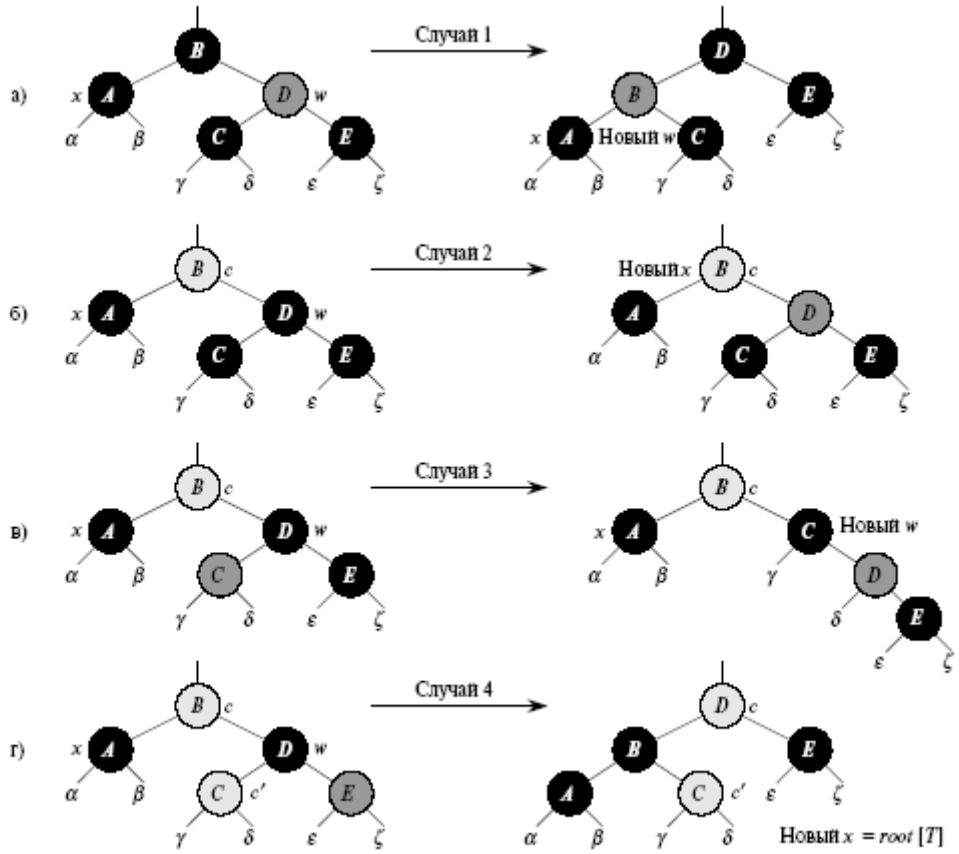

Рис. 24.7. Можливі ситуації, що виникають у циклі ***while*** *процедури* RB_Delete_Fixup*. Темні вузли мають колір* BLACK*, темно-сірі —* RED*, а світлі можуть мати будь-який колір (на малюнку показаний як* c *і* c′*). Грецькими буквами показані довільні піддерева*

Чотири різних можливих випадки (не взаємовиключних) показані на рис. 24.7. Перед тим як приступити до детального розгляду каждого випадку, переконаємося, що в кожнім з випадків перетворення зберігається властивість 5. Ключова ідея полягає в необхідності переконатися, що при перетвореннях у кожнім випадку кількість чорних вузлів (включаючи додаткову чорність у x) на шляху від кореня включно до кожного з піддерев α, β, …, ζ залишається незмінним. Таким чином, якщо властивість 5 виконувалося до перетворення, то воно виконується і після нього. Наприклад, на рис. 24.7*а*, що ілюструє випадок 1, кількість чорних вузлів на шляху від кореня до піддерев α і β дорівнює 3, як до, так і після перетворення (не забудьте про те, що вузол x — двічі чорний). Аналогично, кількість чорних вузлів на шляху від кореня до кожного з піддерев γ, δ, ε і ζ дорівнює 2, як до, так і після перетворення. На рис. 24.7*б* підрахунок повинний включати значення c, рівне значенню атрибута *color* кореня показаного поддерева, що може бути або RED, або BLACK. Так, на шляху від кореня до поддерева α кількість чорних вузлів дорівнює 2 плюс величина, рівна 1, якщо c дорівнює BLACK, і 0, якщо c дорівнює RED; ця величина однакова до і після виконання перетворень. У такій ситуації після перетворення новий вузол x має атрибут *color*, рівний c, але реально це або червоно-чорний вузол (якщо c=RED), або двічі чорний (якщо c=BLACK). Інші випадки можуть бути перевірені аналогічним способом (див. вправа 13.4-5).

Випадок 1: вузол *w* червоний.

Випадок 1 (рядка 5–8 процедури RB_Delete_Fixup і рис. 24.7*а*) виникає, коли вузол w (“брат” вузла x) — червоний. Оскільки w повинний мати чорних нащадків, можна обміняти кольору w і $p[x]$, а потім виконати лівий поворот навколо $p[x]$ без порушення яких-небудь червоно-чорних властивостей. Новий “брат” x, до повороту колишній одним з нащадків w, тепер чорний. Таким путем випадок 1 приводиться до випадку 2, 3 чи 4.

Випадки 2, 3 і 4 виникають при чорному вузлі w і відрізняються друг від друга цветами дочірніх стосовно w вузлів.

Випадок 2: вузол *w* чорний, обоє його дочірніх вузла чорні.

У цьому випадку (рядка 10–24 процедури RB_Delete_Fixup і рис. 24.7*б)* обоє дочірніх вузла *w* чорні. Оскільки вузол *w* також чорний, ми можемо забрати чорне фарбування у *x* и *w*, зробивши *x* одкократно чорним, а *w* — червоним. Для того щоб компенсувати видалення чорного фарбування у *x* и *w*, ми можемо додати додатковий чорний колір вузлу $p[x]$, що до цього міг бути як червоним, так і чорним. Після цього буде виконана наступна ітерація циклу, у якій роль *x* буде грати поточний вузол $p[x]$. Помітимо, що якщо ми переходимо до випадку 2 від випадку 1, новий вузол *x* — червоно-чорний, оскільки вихідний вузол $p[x]$ був червоним. Отже, значення *c* атрибута *color* нового вузла *x* дорівнює RED і цикл завершується при перевірці умови циклу. Після этого новий вузол *x* офарблюється в звичайний чорний колір у рядку 23.

Випадок 3: вузол *w* чорний, його лівий дочірній вузол червоний, а правий — чорний.

У цьому випадку (рядки 13–16 процедури RB_Delete_Fixup і рис. 24.7*в)* вузол *w* чорний, його лівий дочірній вузол червоний, а правий — чорний. Ми можемо обміняти кольору *w* і $left[w]$, а потім виконати правий поворот вокруг *w* без порушення яких-небудь червоно-чорних властивостей. Новим “братом” вузла *x* після цього буде чорний вузол з червоним правим дочірнім вузлом, і в такий спосіб ми привели випадок 3 до випадку 4.

Випадок 4: вузол *w* чорний, його правий дочірній вузол червоний.

У цьому випадку (рядка 17–21 процедури RB_Delete_Fixup і рис. 24.7*г)* вузол *w* чорний, а його правий дочірній вузол — червоний. Виконуючи обмін кольорів і лівий поворот навколо $p[x]$, ми можемо усунути зайву чорність у *x*, роблячи його просто чорним, без порушення яких-небудь червоно-чорних властивостей. Присвоєння *x* вказівника на корінь дерева приводить до завершення роботи при перевірці умови циклу при наступній ітерації.

Аналіз

Чому дорівнює час роботи процедури RB_Delete? Оскільки висота дерева з *n* вузлами дорівнює $O(\lg n)$, загальний час роботи процедури без виконання допоміжної процедури RB_Delete_Fixup дорівнює $O(\lg n)$. У процедурі

RB_Delete_Fixup у випадках 1, 3 і 4 завершення роботи відбувається після виконання постійного числа змін кольору і не більш трьох поворотів. Випадок 2 — єдиний, після якого можливе виконання чергової ітерації циклу **while**, причому вказівник x переміщається нагору по дереву не більш чем $O(\lg n)$ раз, і ніякі повороти при цьому не виконуються. Таким чином, час роботи процедури RB_Delete_Fixup складає $O(\lg n)$, причому вона виконує не більш трьох поворотів. Загальний час роботи процедури RB_Delete, саме собою зрозуміло, також дорівнює $O(\lg n)$.

Література

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Вильямс, 2005. — глава 12, с. 316–335.