

## Лекции 23–24

### Красно-черное дерево

Красно-чёрное дерево (англ. Red-Black-Tree, RB-Tree) – это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов.

Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева – «цвета». Этот атрибут может принимать одно из двух возможных значений – «красный» или «чёрный». Асимптотическая сложность алгоритмов включения/удаления ключей красно-черного дерева есть  $O(\log N)$  (при  $N \rightarrow +\infty$ ).

Красно-чёрное дерево изобретено в 1972 году немцем Рудольфом Байером.

Принято считать, что листовые узлы красно-черных деревьев не содержат данных. Такие листья не нуждаются в явном выделении памяти – нулевой указатель на потомка может фактически означать, что этот потомок – листовой узел.

В ряде источников утверждается, что при работе с красно-черными деревьями использование явных листовых узлов (с выделением им полноценной памяти) может привести к упрощению алгоритма. Имея личный опыт программирования красно-черных деревьев, позволю себе с этим утверждением категорически не согласиться. Кроме того, «явные листовые узлы» занимают в памяти больше места, чем все прочие узлы, и это совершенно неоправданная жертва.

Красно-чёрное дерево должно подчиняться следующим пяти традиционным требованиям:

1. Узел либо красный, либо чёрный.
2. Корень – чёрный.
3. Все листья – черные.
4. Оба потомка каждого красного узла – черные.
5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов.

Эти требования реализуют важное свойство красно-черных деревьев: путь от корня до самого дальнего листа не более чем в два раза длиннее пути от корня до ближайшего листа.

«Традиционность» требований 3 и 4 опирается на «явность» листовых узлов. Можно согласиться с этими требованиями чисто «внешне», то есть при визуализации дерева на элементе PictureBox. При этом каждое связное поле узла (pLeft или pRight), имеющее значение **null**, будет обозначаться маленьким черным прямоугольником под изображением узла (как показано на Рис. 1).

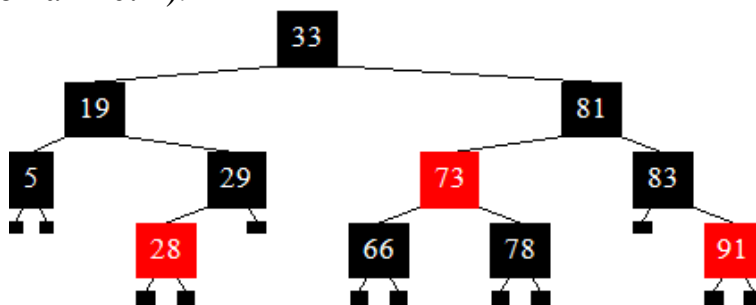


Рис. 1

С точки зрения программиста требование 3 можно отменить, а требование 4 можно заменить требованием

4'. Красный узел либо имеет двух черных (полноценных) потомков, либо является листовым (терминальным).

Здесь и всюду далее слово «потомок» означает *непосредственного* потомка узла. От применения слова «ребенок» лучше воздержаться.

Реализация красно-черного дерева представлена в проекте производным классом `ClassRBTTree`.

```
class ClassRBTTree : ClassBaseTree
{
    private MyNodeType[] mPath;
    private int iDepthBad;
    private bool bBadTree;

    public ClassRBTTree(PictureBox pictureBoxParam)
    {
        ImageGraph = pictureBoxParam;
        ...
    }
    ...
    private void IncludeKey(ref MyNodeType pN, int iKey, int iDepth)
    {
        Array.Resize(ref mPath, iDepth + 1);
        mPath[iDepth] = pN;
        ...
    }
    public override void Include(int iKey)
    {
        if (pRoot == null) // IncludeCase1(), Отдельный метод писать нет смысла.
        {
            NewNode(ref pRoot, iKey);
            pRoot.AuxField.bRed = false;
            return;
        }
        iDepthBad = -1;
        IncludeKey(ref pRoot, iKey, 0);

        while (iDepthBad >= 0)
        {
            if (!mPath[iDepthBad].AuxField.bRed || !mPath[iDepthBad - 1].AuxField.bRed) break;
            // IncludeCase1(), Отдельный метод писать нет смысла.

            // IncludeCase2() содержится в самом IncludeKey().
        }
    }
}
```

```

    if (IncludeCase3(iDepthBad))
    {
        if (iDepthBad < 0) break;
        iDepthBad -= 2;
        continue;
    }

    IncludeCase4(iDepthBad);
    IncludeCase5(iDepthBad);
}

}
protected void ExcludeKey(ref MyNodeType pFrom, int iKey, int iDepth)
{
    if (pFrom == null)
    {
        MessageBox.Show("Ключа " + iKey.ToString() + " нет в RB дереве!");
        return;
    }
    Array.Resize(ref mPath, iDepth + 1);
    mPath[iDepth] = pFrom;
    ...
}
public void Exclude(int iKey)
{
    ExcludeKey(ref pRoot, iKey, 0);
    ...
    if (iDepthBad < 0) mPath = null;

    while (iDepthBad >= 0)
    {
        if (ExcludeCase1(iDepthBad)) break;
// Перебалансировка более не нужна. Дерево уже хорошее.

        ExcludeCase2(iDepthBad);
// iDepthBad на единицу вырос.
// Требуется ExcludeCase4, ExcludeCase5, ExcludeCase6

        if (ExcludeCase3(iDepthBad)) continue;
// Перебалансировка нужна. Дерево еще плохое.
// По отношению к mPath[iDepthBad] требуется перебалансировка,
// начинающаяся со случая ExcludeCase1.

        if (ExcludeCase4(iDepthBad)) break;
// Перебалансировка более не нужна. Дерево уже хорошее.

```

```

        if (ExcludeCase6(iDepthBad)) break;
// Перебалансировка более не нужна. Дерево уже хорошее.
    }

    SettleWithNodePositions(null);
    SettleWithColors();
}
}

```

Методы Include(int iKey) {...} и Exclude(int iKey) {...} класса ClassRBTree являются переопределёнными. Прочие методы характерны только для красно-черного дерева.






В литературе можно найти примеры кодов для красно-черного дерева, использующих в каждом узле дополнительное связное поле – ссылку pParent на родительский узел. Тем, кто желает получить положительную оценку на экзамене, не следует пользоваться таким *нерациональным* подходом.

В приложении WFABinaryTrees используется другой подход. При проникновении вглубь дерева (как при включении, так и при удалении ключа) в массиве mPath, состоящем из элементов типа MyNodeType, запасается цепочка ссылок на узлы, лежащие на пути к искомому месту для нового узла либо к узлу с удаляемым ключом.

Разумеется, для хранения массива mPath требуется память. Однако, количество элементов этого массива есть  $O(\log N)$ , тогда как количество дополнительных ссылок pParent было бы  $O(N)$ , что существенно больше (при  $N \rightarrow +\infty$ ).

Для удобства всюду далее число черных узлов на всяком пути, о котором идет речь в требовании 5, будет называться «Длиной черного пути». Для дерева, подчиняющегося всем пяти требованиям, по сложившейся традиции применяются словосочетание «Дерево действительно». Правда, не все согласны с уместностью такой традиции.

При внесении изменений в красно-черное дерево могут складываться различные ситуации, пояснение которых и действия в которых требуют рисования схем. Далее в этих схемах будут применяться следующие обозначения:

	Красный узел.
	Черный узел.
	Узел произвольного цвета. Цвет, возможно, потребуется запомнить и позже применить.
	Поддерево, возглавляемое черным узлом.
	Поддерево, возглавляемое узлом произвольного цвета.

## Включение в красно-черное дерево

Метод Include(...) реализует внесение нового ключа и, если требуется, последующую перестройку дерева.

Внесение в дерево нового узла с новым ключом выполняется так же, как для обычного двоичного дерева поиска. Новый узел с новым ключом становится терминальным.

От придания ему в подчинение двух фиктивных терминальных узлов мы воздержимся. Возможны следующие ситуации с деревом.

Случай 1. Новый узел – первый в дереве, то есть корневой. Этот узел окрашивается в черный цвет. Дерево действительно. Этот случай обрабатывается первыми операторами метода `Include(...)`.

Во всех остальных случаях, которые обрабатываются методом `IncludeKey(...)`, новый узел окрашивается в красный цвет.

Глобальная переменная (точнее, поле экземпляра класса) `iDepthBad` своим значением, полученным в ходе исполнения метода `IncludeKey(...)`, выражает глубину узла, на котором замечено нарушение требования 4 красно-черного дерева: рассматриваемый красный узел является потомком красного узла. Если дерево действительно, то `iDepthBad=-1`.

Случай 2. Непосредственный предок нового узла – черный. Новый красный узел не увеличивает длины черных путей. Дерево действительно. Этот случай обрабатывается первыми операторами метода `IncludeKey(...)`.

Частный случай случая 2: новый узел – непосредственный потомок корневого узла. Если у него есть брат, второй потомок корневого узла, то он тоже красный, иначе не выполнялось бы требование 5. Если брата нет, к качеству этого брата может быть рассмотрен фиктивный черный терминальный узел (которого, в действительности, в дереве не будет).

Во всех случаях, иных, чем случаи 1 и 2, у нового узла обязательно есть дедушка, который в последующих схемах будет означен символом `G`. Сам новый узел и его родитель обозначаются как `N` и `P` соответственно. Второй потомок узла `G` (даже если им является фиктивный терминальный узел), то есть дядя, обозначается символом `U`.

Каждый из последующих случаев имеет две разновидности, являющиеся «зеркальным отражением» друг друга. Идентификация и обработка случаев осуществляется методами `IncludeCase3(...)`, `IncludeCase4(...)`, `IncludeCase5(...)`, которые вызываются методом `Include(...)` уже после отработки рекурсивного метода `IncludeKey(...)`. Исходная и итоговая конфигурации показаны на пояснительных рисунках слева и справа соответственно.

Случай 3. Узлы `P` и `U` – красные, `G` – черный. Переменная `iDepthBad` показывает глубину узла `N`, который, по требованию 4, должен быть черным, но таковым не является. Узел `N` будем называть виновником нарушения.

Узлы `P` и `U` перекрашиваются в черный, `G` перекрашивается в красный. Длины черных путей не изменились, требование 4 для узлов `N` и `P` теперь выполнено. Но возможно одно из двух осложнений.

Первое, пустяковое: `G` – корень дерева. После перекрашивания `G` снова в черный все длины черных путей увеличиваются на единицу, то есть требование 5 выполняется. Дерево действительно.

Второе, неприятное: родитель узла `G` – красный. Нарушено требование 4. Значение переменной `iDepthBad` уменьшается на 2, после чего Случай 3 применяется вновь, но виновником нарушения объявляется узел `G`. Не исключено, что Случай 3 будет применен несколько раз вплоть до достижения корня дерева.

Реализация случая 3:

```
private bool IncludeCase3(int iDepth)
{
```

```
    if (iDepth < 2) return false;
```

```
    MyNodeType pN = mPath[iDepth];
    if (!pN.AuxField.bRed) return false;
```

```
    MyNodeType pP = mPath[iDepth - 1];
    if (!pP.AuxField.bRed) return false;
```

```
    MyNodeType pG = mPath[iDepth - 2];
```

```
    MyNodeType pU;
```

```
    if (LeftSon(iDepth - 1))
```

```
        pU = pG.pRight;
```

```
    else
```

```
        pU = pG.pLeft;
```

```
    if (pU == null) return false;
```

```
    if (!pU.AuxField.bRed) return false;
```

```
    // Случай 3 – это когда дядя есть, и он красный!
```

```
    pP.AuxField.bRed = false;
```

```
    pG.AuxField.bRed = true;
```

```
    pU.AuxField.bRed = false;
```

```
    if (pRoot == pG) // Первое из осложнений.
```

```
    {
```

```
        pRoot.AuxField.bRed = false;
```

```
        iDepthBad = -1; // Дерево действительно.
```

```
    }
```

```
    return true; // Случай 3 состоялся.
```

```
}
```

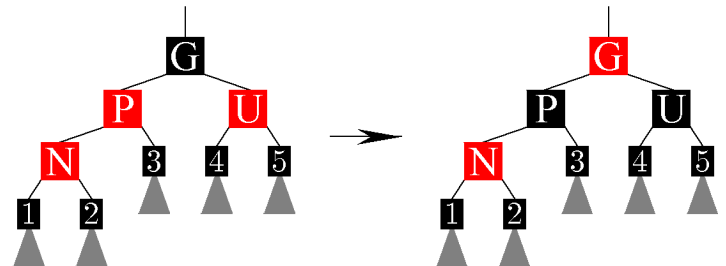


Рис. 2а

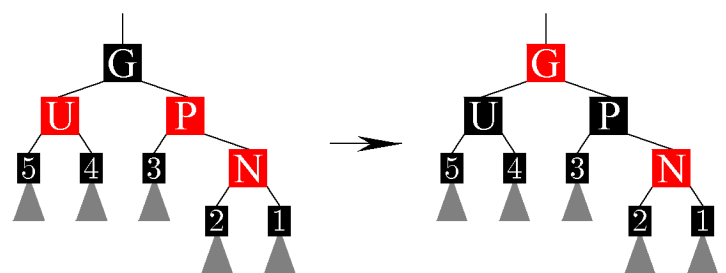


Рис. 2б

В методе имеется обращение к методу LeftSon(int iDepth), возвращающему значение true, если узел на глубине iDepth является левым потомком своего родителя, и false, если правым потомком. На глубине iDepth=0 метод LeftSon(...) не вызывается.

Случай 4. Узлы G и U – черные, P – красный. P – левый потомок G, а N – правый потомок P (зеркально-симметричный случай: P – правый потомок G, а N – левый потомок P). Производится перестройка дерева по схеме, показанной на Рис. 3а или Рис. 3б.

Длины черных путей не изменяются. Требование 4 все еще не выполнено. Однако, полученная конфигурация соответствует случаю 5 (при обмене именами узлов Р и N), рассматриваемому далее.

Реализация случая 4:

```
private void IncludeCase4(int iDepth)
{
    if (iDepth < 2) return;
    MyNodeType pN = mPath[iDepth];
    if (!pN.AuxField.bRed) return;
    MyNodeType pP = mPath[iDepth - 1];
    if (!pP.AuxField.bRed) return;
    MyNodeType pG = mPath[iDepth - 2];
    MyNodeType pU, p2;

    if (LeftSon(iDepth - 1))
    {
        if (pN == pP.pLeft) return;
        // Case4 – это когда pN == pP.Right

        pU = pG.pRight;
        if (pU != null)
            if (pU.AuxField.bRed) return;

        p2 = pN.pLeft;
        pN.pLeft = pP;
        pG.pLeft = pN;
        pP.pRight = p2;
    }
    else
    {
        if (pN == pP.pRight) return;
        // Case4 Зеркальный – это когда pN == pP.Right

        pU = pG.pLeft;
        if (pU != null)
            if (pU.AuxField.bRed) return;

        p2 = pN.pRight;
        pN.pRight = pP;
        pG.pRight = pN;
        pP.pLeft = p2;
    }
    mPath[iDepth - 1] = pN;
    mPath[iDepth] = pP;
}
```

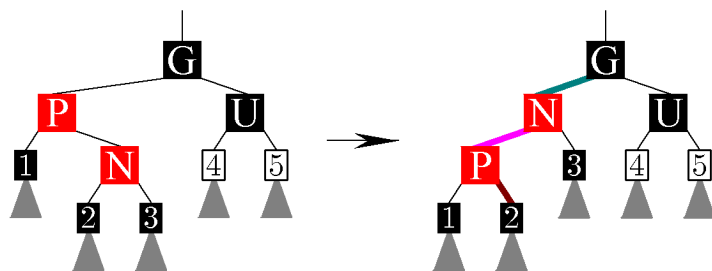


Рис. 3а

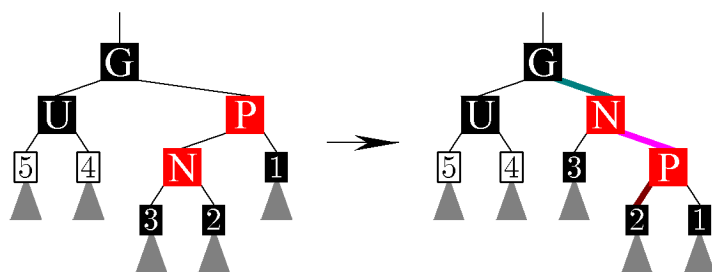


Рис. 3б

```

}

```

Случай 5. Узлы G и U – черные, P – красный. P – левый потомок G, а N – левый потомок P (зеркально-симметричный случай: P – правый потомок G, а N – правый потомок P). Производится перестройка дерева по схеме, показанной на Рис. 4а или Рис. 4б. Титул главы поддереву отнимается у узла G и передается узлу P. Длины черных путей не изменяются. Но требование 4 уже выполнено. Дерево действительно.

Реализация случая 5:

```

private void IncludeCase5(int iDepth)
{
    if (iDepth < 2) return;
    MyNodeType pN = mPath[iDepth];
    if (!pN.AuxField.bRed) return;
    MyNodeType pP = mPath[iDepth - 1];
    if (!pP.AuxField.bRed) return;
    MyNodeType pG = mPath[iDepth - 2];
    MyNodeType pU, p3;

    if (LeftSon(iDepth - 1))
    {
        pU = pG.pRight;
        if (pU != null)
            if (pU.AuxField.bRed) return;

        p3 = pP.pRight;
        pP.pRight = pG;
        pG.pLeft = p3;
    }
    else
    {
        pU = pG.pLeft;
        if (pU != null)
            if (pU.AuxField.bRed) return;

        p3 = pP.pLeft;
        pP.pLeft = pG;
        pG.pRight = p3;
    }
}

```

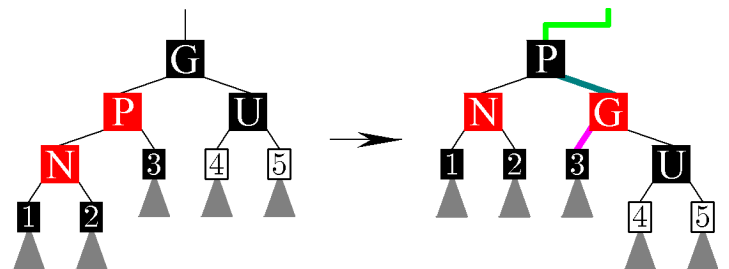


Рис. 4а

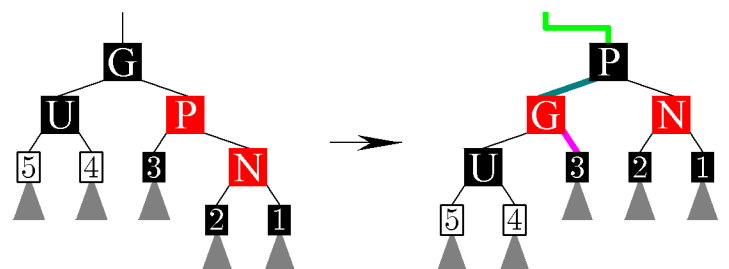


Рис. 4б

```

pP.AuxField.bRed = false;
pG.AuxField.bRed = true;

```



```

if (pG == pRoot)
{
    pRoot = pP;
    return;
}

if (LeftSon(iDepth - 2))
    mPath[iDepth - 3].pLeft = pP;
else
    mPath[iDepth - 3].pRight = pP;
}

```

Рассмотренные ситуации покрывают весь спектр возможных случаев, возникающих при внесении нового ключа в красно-черное дерево.

### Исключение из красно-черного дерева

Метод Exclude(...) реализует удаление ключа и, если требуется, последующую перестройку дерева.

Удаление ключа из дерева выполняется так же, как для обычного двоичного дерева поиска. Если узел с удаляемым ключом имеет одно направление потомков (или ни одного), ссылка на него с узла-родителя перебрасывается на узел-потомок (или обнуляется). Если удаляемый ключ занимает узел с двумя потомками, этот узел обменивается ключами с узлом, являющимся самым правым из его левых потомков, который имеет не более одного направления своих потомков.

При внесении нового ключа в красно-черное дерево могут возникнуть проблемы из-за «лишнего» красного узла. При удалении ключа могут возникнуть проблемы из-за «недостающего» черного узла.

После удаления красного узла дерево остается действительным.

Пусть удаляется черный узел. Для его единственного потомка (на которого перебрасывается ссылка с родителя) далее применяется обозначение N. Если потомка нет, N будет подразумевать «фиктивный терминальный узел», а ссылка на него будет иметь значение **null**.

Узел условимся называть **проблемным**, если все длины черных путей, проходящих через него, на единицу меньше, чем все длины черных путей, **не** проходящих через него.

После удаления черного узла все длины черных путей, проходящих через N, сокращаются на единицу, в отличие от путей, не проходящих через N. Требование 5 перестает соблюдаться, дерево становится недействительным. Узел N является **проблемным**.

Просто исправляется ситуация, когда узел N является красным: достаточно перекрасить его в черный цвет.

Сложнее может оказаться ситуация, когда узел N – черный. В этой ситуации может иметь место один из случаев, обсуждаемых далее. Каждому случаю посвящен отдельный метод кода. Случаи рассматриваются в порядке возрастания их номеров. Иногда такой порядок нарушается, и управление передаётся методу, рассматривающему случай с меньшим номером.

Случай 1. N – новый корень. Все длины черных путей, проходящих через корень, сократились на единицу, а значит, остались равными друг другу. Дерево осталось действительным.

Для прочих случаев вводятся обозначения: родитель и брат N обозначаются как P и S соответственно, левый и правый непосредственные потомки брата обозначаются как SL и SR соответственно.

Случай 2. S – красный, тогда как SL, SR – черные. При этом P может быть только черным. Если бы он был красным, его потомок S не мог бы быть красным согласно требованию 4. Производится перестройка дерева по схеме, показанной на Рис. 5а или Рис. 5б. Титул главы поддереву отнимается у узла P и передается узлу S. Узел N сохраняет статус *проблемного*. Дерево не становится действительным. Но создается основа для обработки случаев 4, 5 или 6.

Реализация случая 2:

```
private bool ExcludeCase2(int iDepth)
{
    MyNodeType pN = mPath[iDepth];
    MyNodeType pP = mPath[iDepth - 1];
```

```
    MyNodeType pS;
```

```
    bool bLeft = LeftSon(iDepth);
```

```
    if (bLeft)
```

```
        pS = pP.pRight;
```

```
    else
```

```
        pS = pP.pLeft;
```

```
    if (pS == null) return false;
```

```
    if (!pS.AuxField.bRed) return false;
```

```
    if (bLeft)
```

```
    {
        MyNodeType pSL = pS.pLeft;
        pP.pRight = pSL;
        pS.pLeft = pP;
    }
```

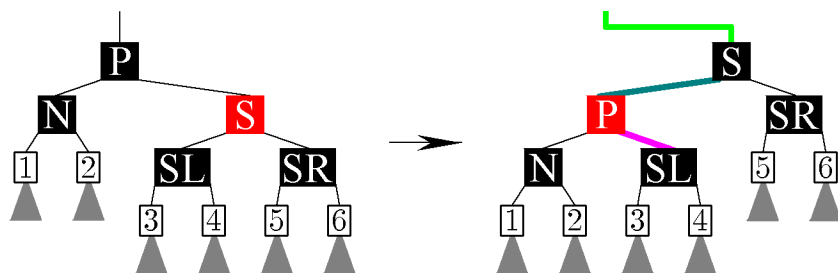


Рис. 5а

```
    else
```

```
    {
        MyNodeType pSR = pS.pRight;
        pP.pLeft = pSR;
        pS.pRight = pP;
    }
```

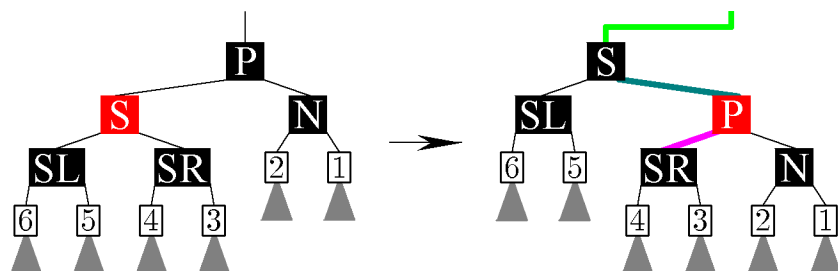


Рис. 5б

```

pP.AuxField.bRed = true;
pS.AuxField.bRed = false;

if (pRoot == pP)
    pRoot = pS;
else
{
    if (LeftSon(iDepth - 1))
        mPath[iDepth - 2].pLeft = pS;
    else
        mPath[iDepth - 2].pRight = pS;
}

mPath[iDepth - 1] = pS;
mPath[iDepth] = pP;

iDepthBad = iDepth + 1;

Array.Resize(ref mPath, iDepthBad + 1);
mPath[iDepthBad] = pN;

return true; // Случай 2 состоялся.
}

```

Случай 3. P, S, SL, SR – черные. Узел S перекрашивается в красный цвет (Рис. 6а, Рис. 6б). Длины черных путей, проходящих через узел N, не изменились. Дерево не стало действительным. Но уменьшились на единицу длины черных путей, проходящих через узел S – второго потомка P. Следовательно, статус *проблемного* теперь перехватил более высокий узел P. Для исправления новой ситуации следует возобновить перебор случаев, начиная со случая 1, но теперь по отношению к *проблемному* узлу P.

Реализация случая 3:

```

private bool ExcludeCase3(int iDepth)
{
    MyNodeType pN = mPath[iDepth];
    if (pN != null)
        if (pN.AuxField.bRed) return false;

    MyNodeType pP = mPath[iDepth - 1];
    if (pP.AuxField.bRed) return false;

    MyNodeType pS;

    if (LeftSon(iDepth))
        pS = pP.pRight;
}

```

```

else
    pS = pP.pLeft;

if (pS == null)
{
    iDepthBad = iDepth - 1;
    return true;
}

if (pS.AuxField.bRed) return false;

```

```

MyNodeType pSL = pS.pLeft;
if (pSL != null)
    if (pSL.AuxField.bRed)
        return false;

```

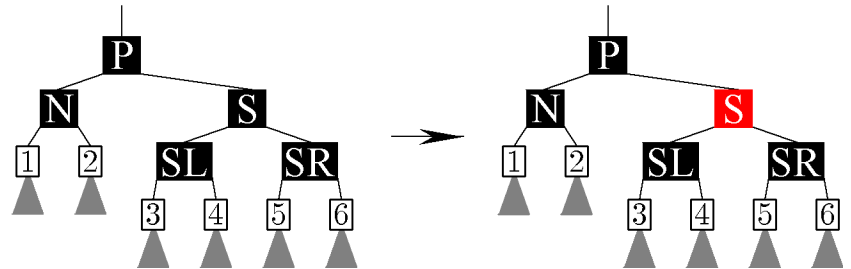


Рис. 6а

```

MyNodeType pSR = pS.pRight;
if (pSR != null)
    if (pSR.AuxField.bRed)
        return false;

```

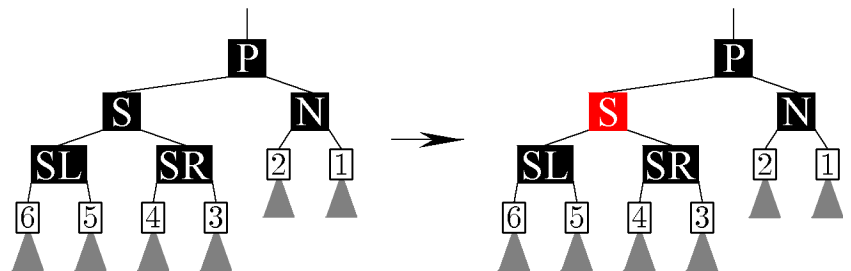


Рис. 6б

```

pS.AuxField.bRed = true; // Перекрашивание состоялось.
iDepthBad = iDepth - 1;

return true; // Случай 3 состоялся.
}

```

Случай 4. P – красный, S, SL, SR – черные. Узел S перекрашивается в красный цвет, P перекрашивается в черный (Рис. 7а, Рис. 7б). Длины черных путей, проходящих через узел S, не изменились. Длины черных путей, проходящих через узел N, увеличились на единицу. Дерево стало действительным. Перестройка дерева завершена, рассмотрение случаев прекращается.

Реализация случая 4:

```

bool ExcludeCase4(int iDepth)
{
    MyNodeType pN = mPath[iDepth];
    if (pN != null)
        if (pN.AuxField.bRed) return false;
}

```

```

MyNodeType pP = mPath[iDepth - 1];
if (!pP.AuxField.bRed) return false;

```

```

MyNodeType pS;

```

```

if (LeftSon(iDepth))
    pS = pP.pRight;
else
    pS = pP.pLeft;

```

```

if (pS == null)
{
    pP.AuxField.bRed = false;
    iDepthBad = -1;
    return true;
}

```

```

if (pS.AuxField.bRed) return false;

```

```

MyNodeType pSL = pS.pLeft;
if (pSL != null)
    if (pSL.AuxField.bRed)
        return false;

```

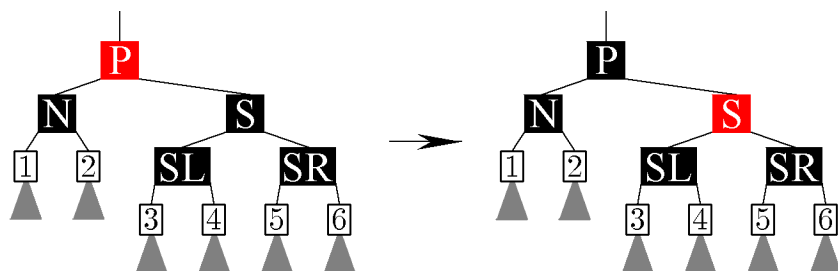


Рис. 7а

```

MyNodeType pSR = pS.pRight;
if (pSR != null)
    if (pSR.AuxField.bRed)
        return false;

```

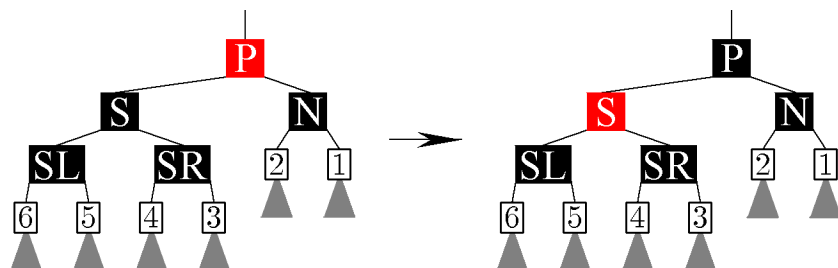


Рис. 7б

```

pS.AuxField.bRed = true;
pP.AuxField.bRed = false;

```

```

iDepthBad = -1; // Дерево стало действительным. Корректировать больше не надо.

```

```

return true; // Случай 4 состоялся.

```

```

}

```

Случай 5а. N есть левый потомок P, S – черный, его левый потомок SL – красный, тогда как правый потомок SR – черный. Цвета P и N не имеют значения. Производится перестройка дерева по схеме, показанной на Рис. 8а.

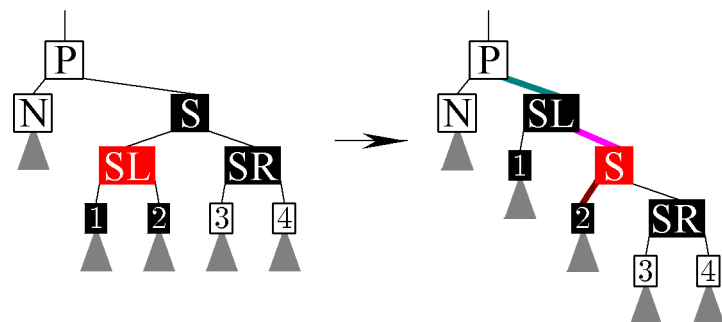


Рис. 8а

Случай 5б (симметричный случаю 5а). N есть правый потомок P, S – черный, его правый потомок SR – красный, тогда как левый потомок SL – черный. Цвета P и N не имеют значения. Производится перестройка дерева по схеме, показанной на Рис. 8б.

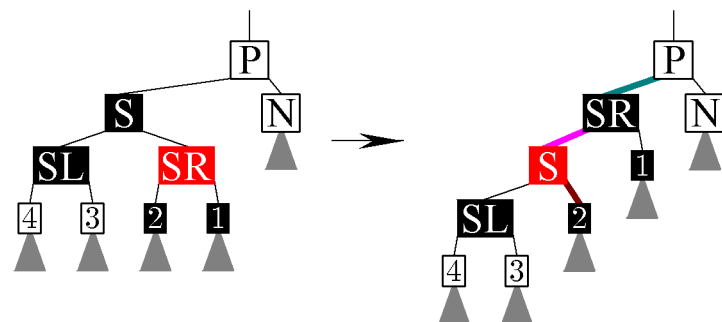


Рис. 8б

Исходные цвета узлов P и N не имеют значения.

После перестройки узел N остается **проблемным**. Длины черных путей, проходящих через узлы N, S, SL, SR не изменяются. Однако, создается основа для обработки случая 6.

Случай 6а. N – левый потомок узла P, S – черный, его правый потомок SR – красный. Производится перестройка дерева по схеме, показанной на Рис. 9а. P и SR перекрашиваются в черный.

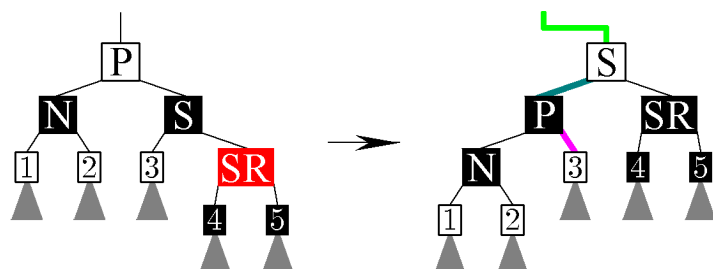


Рис. 9а

Случай 6б (симметричный случаю 6а). N – правый потомок узла P, S – черный, его правый потомок SL – красный. Производится перестройка дерева по схеме, показанной на Рис. 9б. P и SL перекрашиваются в черный.

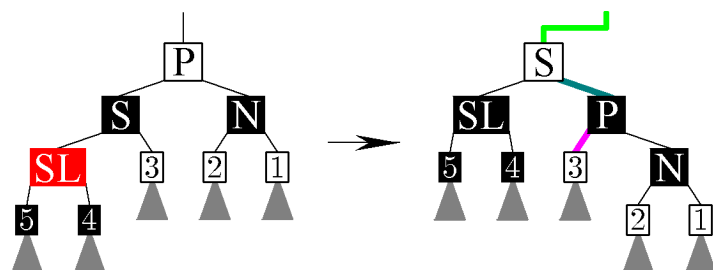


Рис. 9б

Исходный цвет узла P не имеет значения, однако, этот цвет передается узлу S. Титул главы поддеревы отнимается у узла P и передается узлу S.

Длины черных путей, проходящих через узлы S, SL, SR не изменяются. Длины черных путей, проходящих через узел N, увеличиваются на единицу. Дерево стало действительным. Перестройка дерева завершена, рассмотрение случаев прекращается.

Реализация случаев 5 и 6 собрана в одном методе:

```
bool ExcludeCase6(int iDepth)
{
    MyNodeType pN = mPath[iDepth];
    if (pN != null)
        if (pN.AuxField.bRed) return false;

    MyNodeType pP = mPath[iDepth - 1];

    bool bRedP = pP.AuxField.bRed; // Запоминается исходный цвет узла Р для передачи S.
    bool bLeft = LeftSon(iDepth);

    MyNodeType pS;

    if (bLeft)
        pS = pP.pRight;
    else
        pS = pP.pLeft;

    if (pS == null) return false;
    if (pS.AuxField.bRed) return false;

    MyNodeType pSL = pS.pLeft;
    MyNodeType pSR = pS.pRight;

    MyNodeType p2, p3;

    if (bLeft)
    {
        if (pSR == null || pSR != null && !pSR.AuxField.bRed)
        {
            if (pSL == null) return false;
            if (!pSL.AuxField.bRed) return false;
        }
        // Сюда попали => это Случай 5а.

        p2 = pSL.pRight;

        pSL.pRight = pS;
        pS.pLeft = p2;
        pP.pRight = pSL;

        pSL.AuxField.bRed = false;
        pS.AuxField.bRed = true;
    }
}
```

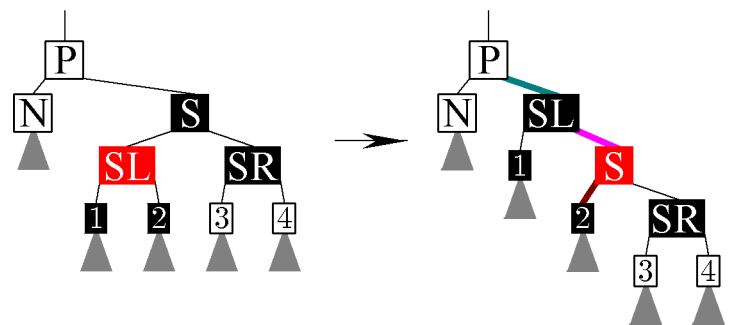


Рис. 8а

```

    pSR = pS;
    pS = pSL;
    // Подготовлены брат узла N и правый потомок брата
    // для последующего распознавания конфигурации случая 6а.
  }
  else
    if (!pSR.AuxField.bRed) return false;

```

// Сюда попали => это Случай 6а.

```

p3 = pS.pLeft;
pS.pLeft = pP;
pP.pRight = p3;

```

```

pSR.AuxField.bRed = false;

```

```

}
else
{
  if (pSL == null || pSL != null && !pSL.AuxField.bRed)
  {
    if (pSR == null) return false;
    if (!pSR.AuxField.bRed) return false;

```

// Сюда попали => это Случай 5б (зеркально симметричный случаю 5а).

```

p2 = pSR.pLeft;

```

```

pSR.pLeft = pS;
pS.pRight = p2;
pP.pLeft = pSR;

```

```

pSR.AuxField.bRed = false;
pS.AuxField.bRed = true;

```

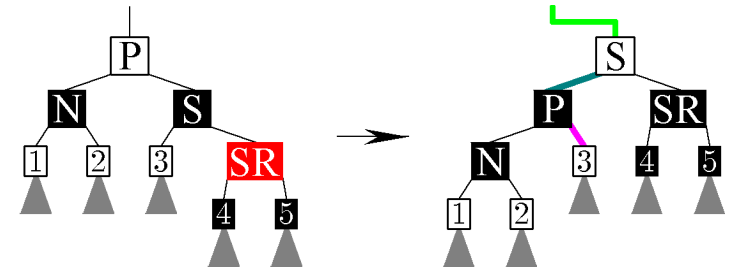


Рис. 9а

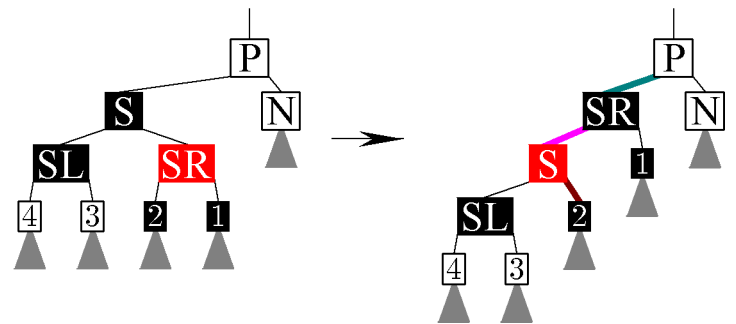


Рис. 8б

```

pSL = pS;
pS = pSR;

```

// Подготовлены брат узла N и левый потомок брата  
 // для последующего распознавания конфигурации случая 6б.

```

}
else
  if (!pSL.AuxField.bRed) return false;

```



// Сюда попали => это Случай 6б.

```
p3 = pS.pRight;
pS.pRight = pP;
pP.pLeft = p3;
```

```
pSL.AuxField.bRed = false;
```

```
}
```

```
if (pRoot == pP)
```

```
    pRoot = pS;
```

```
else
```

```
    if (LeftSon(iDepth - 1))
```

```
        mPath[iDepth - 2].pLeft = pS;
```

```
    else
```

```
        mPath[iDepth - 2].pRight = pS;
```

```
pP.AuxField.bRed = false;
```

```
pS.AuxField.bRed = bRedP;
```

```
iDepthBad = -1;
```

```
return true; // Случай 6 состоялся.
```

// Дерево стало Действительным. Корректировать больше не надо.

```
}
```

С прочими подробностями класса [ClassRBTree](#) можно ознакомиться в прилагаемом к данному документу проекте WFABinaryTrees.

Здесь имеет смысл привести текст одно уже использованного метода:

```
private bool LeftSon(int i)
```

```
{
```

```
    if (i <= 0)
```

```
    {
```

```
        ItIsUnimpossible("Точка RB1.");
```

```
        return false;
```

```
    }
```

```
    if (mPath[i - 1].pLeft == mPath[i])
```

```
        return true;
```

```
    else
```

```
    if (mPath[i - 1].pRight == mPath[i])
```

```
        return false;
```

```
    else
```

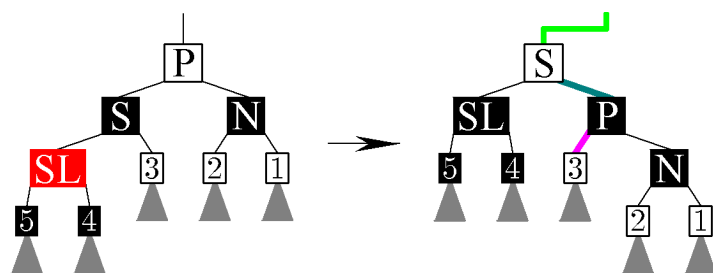


Рис. 9б

```
{  
    ItIsUnimpossible("Точка RB2.");  
    return false;  
}
```

Метод `ItIsUnimpossible(...)`, объявленный в классе `ClassBaseTree`, прекращает работу приложения, предварительно давая знать, из какой точки произошел выброс из-за ошибки. Метод написан «на всякий случай». Пока что он ни разу не понадобился.