Курс kiev-clrs – Лекция 10. Красно-чёрные деревья

Олег Смирнов

13 июня 2009 г.

Содержание

1 Цель лекции

- Красно-чёрные деревья поиска
- Изометрия между RB и 2-3-4 деревьями

2 Сбалансированные поисковые деревья

Сбалансированным называется дерево, высота которого гарантировано не превышает $\lg n$ для n элементов. Таким образом операции поиска, вставки и удаления на нём работают за $O(\lg n)$ итераций.

Существуют разные подходы к организации таких структур:

- AVL-деревья (придуманы в 1962 году в СССР)
- 2-3 деревья
- 2-3-4 деревья
- AA-деревья
- RB-деревья (Red-Black)
- Списки с пропусками (Skip lists)
- Декартовы деревья (Treaps = Tree + Heap)

2-3 и 2-3-4 являются разновидностями более общей структуры – B-деревьев.

3 Красно-чёрные деревья

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлена дополнительный аттрибут – цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева – чёрные
3. У красного узла родительский узел – чёрный

4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов – black-height(x)

5. Чёрный узел может иметь чёрного родителя

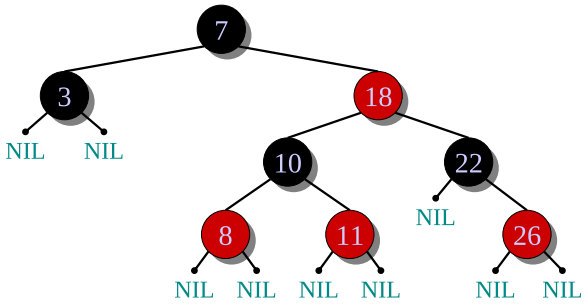

Рис. 1: Пример красно-чёрного дерева

При подсчете количества чёрных узлов считаем листья (конечные узлы) за единицу. Свойства дерева (в основном третье и четвертое) гаранти-

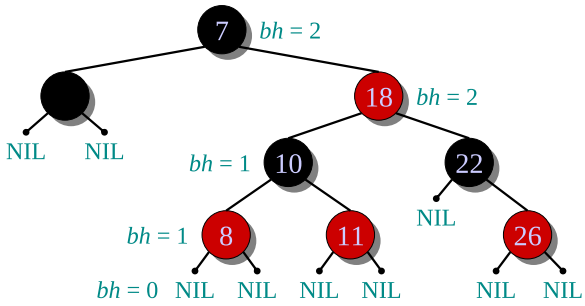

Рис. 2: Свойство black-height(x)

руют его логарифмическую высоту.

Сконструировать красно-чёрное дерево из входного массива довольно просто. Например, можно положить все узлы чёрными. Задача разработать алгоритм вставки (и удаления) элемента, который бы позволил поддерживать свойство балансировки и при этом работал за логарифмическое время.

4 Высота красно-чёрного дерева

Теорема: Красно-чёрное дерево с n ключами имеет высоту

$$h \leqslant 2\lg(n+1) = O(\lg n)$$

Схема доказательства:

1. Совмещаем все красные узлы с родительскими чёрными (они существуют по свойству 2) начиная с корня
2. В результате получим дерево, каждый узел котрого имеет 2, 3 или 4 потомка
3. В следствии свойства 4 красно-чёрного дерева, все листья 2-3-4 дерева будут иметь одинаковую глубину – black-height корня исходного дерева
4. Пусть дерево из п.2 имеет высоту h'. Тогда $h' \geqslant h/2$, т.к. не более половины узлов на каждом пути – красные

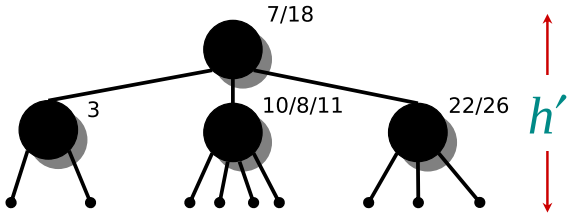

Рис. 3: 2-3-4 дерево

5. Количество листьев не изменяется и равно $n+1$ для обоих деревьев, т.к. в красно-чёрном дереве все внутренние узлы имеют ровно два листа
6. В 2-3-4 дереве высоты h' количество листьев $n+1$ ограничено: $2^{h'} \leqslant n+1 \leqslant 4^{h'}$
7. Тогда:

$$\begin{aligned} n+1 &\geqslant 2^{h'} \\ \lg(n+1) &\geqslant h' \geqslant h/2 \\ h &\leqslant 2\lg(n+1) \end{aligned}$$

5 Алгоритмы на красно-чёрных деревьях

5.1 Поиск и запросы

Алгоритм поиска Search, а также алгоритмы Min, Max, Successor и Predecessor аналогичны алгоритмам для бинарных поисковых деревьев.

5.2 Вставка и удаление

В отличии от запросов, модификация может нарушить свойства красно-чёрных деревьев. Поэтому необходимы дополнительные операции:

- Вставка/удаление аналогично BST
- Изменение цветов узлов
- Реструктуризация связей между узлами с помощью “вращений”

Вращения бывают “правые” и “левые”. Операции вращения сохраняют свойства поискового дерева: $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leqslant A \leqslant b \leqslant B \leqslant c$ и выполняются за $O(1)$ итераций. Идея алгоритма вставки:

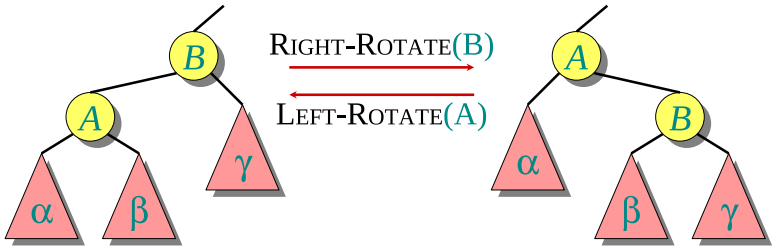

Рис. 4: Правое вращение B и левое вращение A

- Добавить элемент x помощью BST Tree_Insert
- Установить цвет x – красный
- Если третье свойство “красно-чёрности” нарушено, “передвигать” нарушение вверх по дереву, выполняя операции изменения цвета и поворотов

- Цвета должны изменяться таким образом, чтоб не нарушить четвертое свойство
- В конце работы цвет корня дерева – чёрный

На первом шаге (рис. 5) новый элемент (15) красного цвета. Нарушено третье свойство, т.к. его родитель (11) также красный. Если новый элемент сделать чёрным, то нарушится четвертое свойство, т.к. тогда из 11 левый путь будет содержать один чёрный элемент, а правый – два.

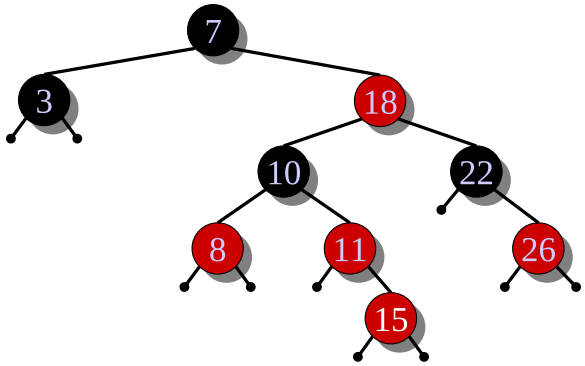

Рис. 5: Пример, шаг 1: добавляем элемент $x = 15$

“Прародитель” нового узла (10) чёрного цвета (рис. 6), а два его потомка – красные. Это значит, что можно изменить цвета, сместив нарушение на уровень 10.

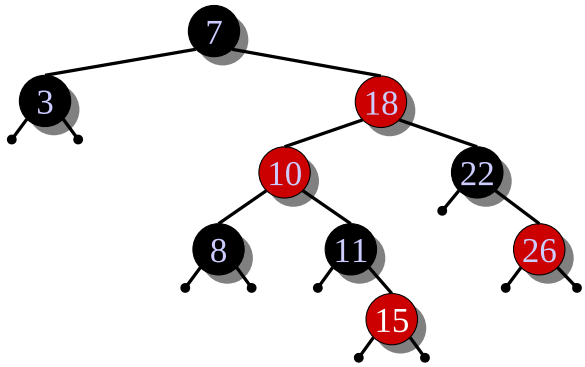

Рис. 6: Пример, шаг 2: изменяем цвет 8, 10 и 11

Изменить цвет элемента 7 и его потомков 3 и 18 нельзя, т.к. они разных цветов (рис. 7). Вместо этого выполняем правое вращение относительно элемента 18.

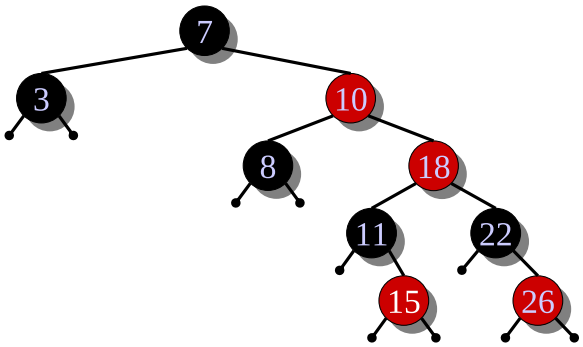

Рис. 7: Пример, шаг 3: правое вращение относительно 18

После вращение относительно 7, дерево снова становится сбалансированным (рис. 8). Изменяем цвет нового корня (10) на чёрный.

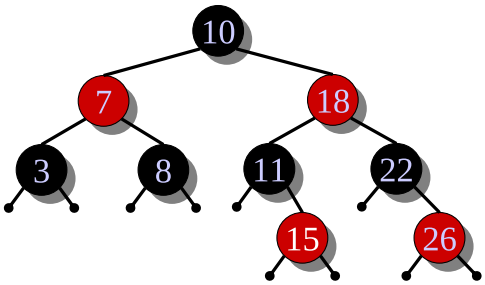

Рис. 8: Пример, шаг 4: левое вращение относительно 7

Алгоритм выполняет не более двух вращений и не более $\lg n$ шагов, пока не будет достигнут корень дерева.

Итак, алгоритм рассматривает три возможных случая:

- В случае 1, когда прародитель узла x – чёрный, а второй потомок прародителя (“дядя”) – красный или в симметричном (когда потомки A поменяны местами), чёрный цвет C “опускается” на A и D (на родителя и “дядю”), а прародитель C становится красным. После этого нужно продолжить балансировку, т.к. родитель C тоже может быть красным

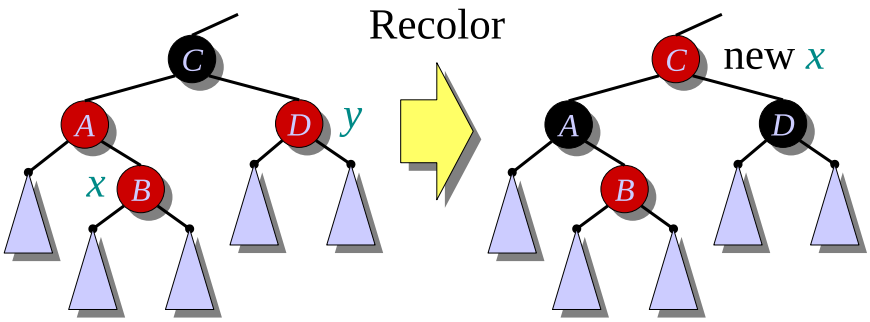

Рис. 9: Случай 1

- В случае 2 “дядя” узла x – корень поддерева y – чёрный. Цвет менять нельзя, т.к. это нарушит четвертое свойство. Левое вращение относительно A сводит ситуацию к случаю 3

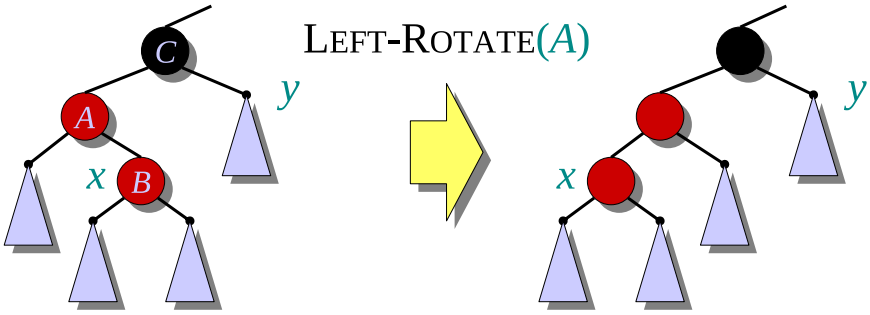

Рис. 10: Случай 2

- В случае 3 правое вращение относительно элемента B исправляет нарушения. Больше модификаций не требуется независимо от того, какой цвет у родителя C (B после вращения)

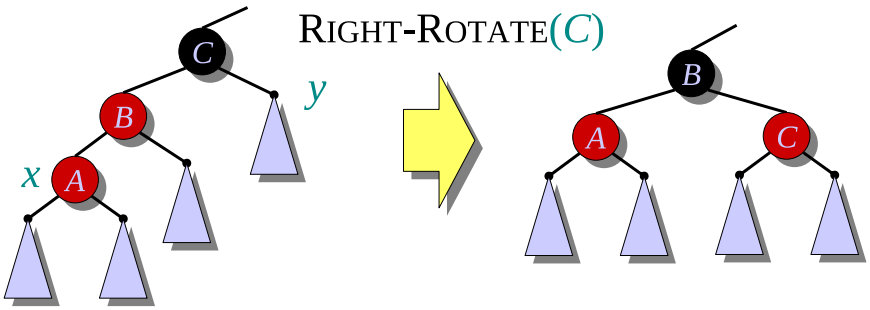

Рис. 11: Случай 3

RB_INSERT(T, x)

```
1  Tree_Insert(T, x)
2  color[X] ← RED
3  while x ≠ root[T] and color[x] = RED //Родитель – красный
4        do if parent[x] = left[parent[parent[x]]]
5              then y ← right[parent[parent[x]]] //y ← "дядя"
6                   if color[y] = RED //Можно менять цвет
7                      then ... //<Случай 1>
8                      else  if x = right[parent[x]]
9                                then ... //<Случай 2>
10                            ... //<Случай 3>
11             else  ... //Симметрично, рассматриваем left вместо right
12 color[root[T]] ← BLACK
```

Крис Окасаки [Oka99] предложил альтернативную реализацию красно-чёрных деревьев в функциональной парадигме. В отличии от императивного подхода, имея в распоряжении алгебраические типы данных и механизм pattern matching, можно рассмотреть всего четыре случая, требующие модификации дерева.

Алгоритм Окасаки на Haskell:

```
data Color                  = R | B
data Tree elt               = E | T Color (Tree elt) elt (Tree elt)
type Set a                  = Tree a
empty                       :: Set el
empty                       = E
member                      :: Ord elt ⇒ elt → Set elt → Bool
member x E                  = False
```

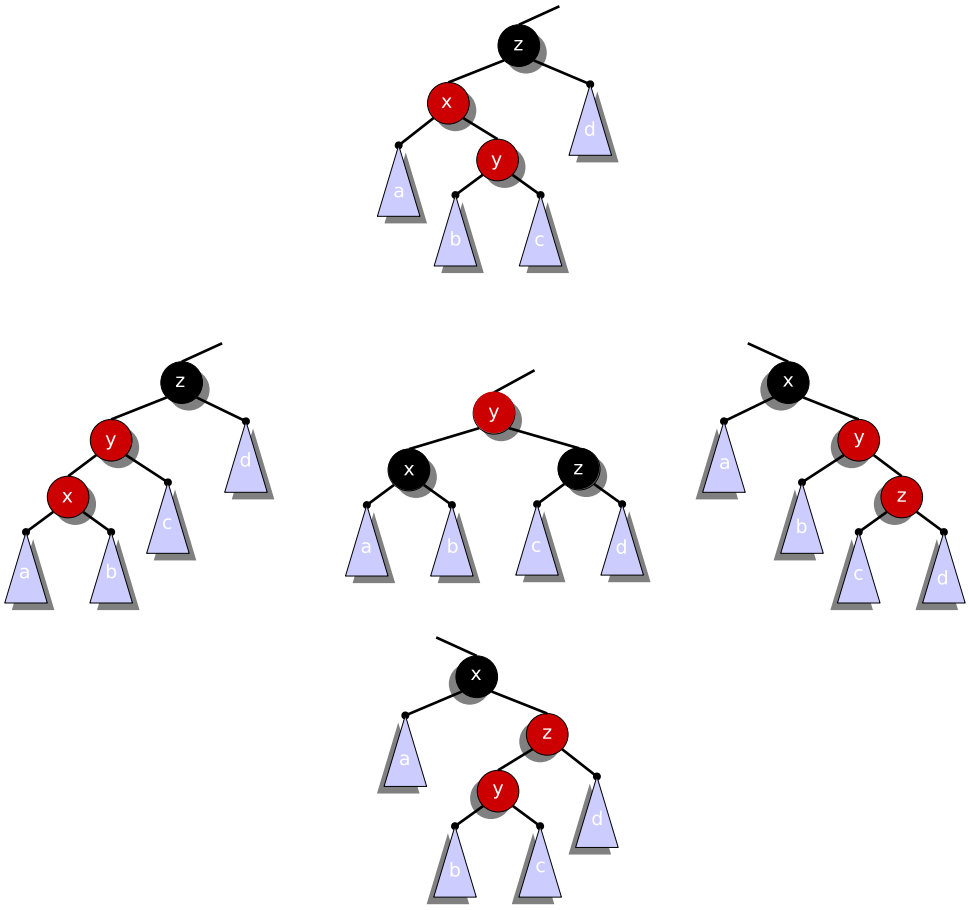

Рис. 12: Алгоритм Окасаки

```
member x (T _ a y b)          | x < y  = member x a
                              | x == y = True
                              | x > y  = member x b
insert                        :: Ord elt ⇒ elt → Set elt → Set elt
insert x s                    = makeBlack (ins s)
  where
    ins E                     = T R E x E
    ins (T color a y b)       | x < y  = balance color (ins a) y b
                              | x == y = T color a y b
                              | x > y  = balance color a y (ins b)
    makeBlack (T _ a y b) = T B a y b
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance color a x b           = T color a x
```

Альтернативная реализация функции balance рассматривает те же случаи, что и в императивном алгоритме:

```
balance B (T R a (T R _ _ _) x b) y (T R c z d)
| B (T R a x b (T R _ _ _)) y (T R c z d)
| B (T R a x b) y (T R c (T R _ _ _) z d)
| B (T R a x b) y (T R c z d (T R _ _ _))      = T R (T B a x b) y (T B c z d)
  — color flip balance B (T R a (T R _ _ _) x b) y c = T B a x (T R b y c)
balance B a x (T R b y c (T R _ _ _))         = T B (T R a x b) y c
  — single rotation
balance B (T R a x (T R b y c)) z d
| B a x (T R (T R b y c) z d)                 = T B (T R a x b) y (T R c z d)
  — double rotation
balance color a x b                           = T color a x b
```

Разница между реализациями в количестве выполняемых операций. Например, смена цвета в императивном алгоритме выполняется в три присваивания, а соответствующая трансформация в функциональном – за семь или больше операций с цветом и с указателями.

Императивный алгоритм выполняет операцию Insert в две фазы: спуск по дереву сверху-вниз со вставкой элемента, затем *балансировка* дерева снизу-вверх. Балансировка может прекратиться до того, как будет достигнут корень дерева.

Функциональный алгоритм выполняет поиск сверху-вниз, а затем *конструкцию* дерева снизу вверх. Процедура конструирования, в отличии от балансировки, не может быть прервана досрочно.

6 Заключительные замечания

Одно из основных преимуществ красно-чёрных деревьев заключается в том, что процедуру балансировки практически всегда можно выполнять параллельно с процедурами поиска, т.к. алгоритм поиска не зависит от аттрибута цвета узлов. Вращение поддеревьев не может выполнятся одновременно с поиском, но при вставке выполняется не более $O(1)$ вращений.

Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнеры библиотеки STL (map, set, multiset, multimap) основаны на красно-чёрных деревьях.

Легко видеть, что красно-чёрные деревья изометричны 2-3-4 B-деревьям. Каждый чёрный узел можно объединить с его красными потомками. Результирующий узел будет иметь не более трех ключей и не более четырех потомков.

Список литературы

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.