МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет
вычислительной математики
и кибернетики

О.В. Сенюкова

СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ ПОИСКА

МОСКВА

2014

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

О.В. Сенюкова

СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ ПОИСКА

Учебно-методическое пособие

МАКС ПРЕСС

МОСКВА – 2014

УДК 004.021:519.163(075.8)
ББК 22.12:22.176я73
С31

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ имени М.В. Ломоносова

Рецензенты:
доцент *А.А. Белеванцев*
(факультет вычислительной математики и кибернетики МГУ имени М.В. Ломоносова);
доцент *Ю.С. Корухова*
(факультет вычислительной математики и кибернетики МГУ имени М.В. Ломоносова)

Сенюкова О.В.

С31 **Сбалансированные деревья поиска:** Учебно-методическое пособие. – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова (лицензия ИД N 05899 от 24.09.2001 г.); МАКС Пресс, 2014. – 68 с.
ISBN 978-5-89407-528-0
ISBN 978-5-317-04873-0

Методическое пособие посвящено сбалансированным деревьям поиска. В начале пособия рассматриваются деревья поиска общего вида. Далее рассматриваются три вида сбалансированных деревьев поиска: АВЛ-деревья, красно-черные деревья и самоперестраивающиеся деревья. Теоретический материал сопровождается иллюстрациями и примерами реализации операций над деревьями на псевдокоде. В конце каждого раздела предлагается набор задач теоретического характера для самостоятельного решения. Последний раздел пособия посвящен сравнению рассмотренных видов деревьев и приведены примеры их практического использования.

Пособие предназначено для студентов и преподавателей лекционного курса «Алгоритмы и алгоритмические языки» и поддерживающего его курса «Практикум на ЭВМ».

УДК 004.021:519.163(075.8)
ББК 22.12:22.176я73

ISBN 978-5-89407-528-0
ISBN 978-5-317-04873-0

Оглавление

1. Введение

Данное методическое пособие посвящено определенному виду структур данных – сбалансированным деревьям поиска. В начале пособия вводится понятие дерева поиска, на примерах рассматриваются его свойства, разбираются основные операции над ним: поиск, вставка и удаление узла. Далее подробно излагается теоретический материал по трем видам сбалансированных деревьев поиска: АВЛ-деревьям, красно-черным деревьям и самоперестраивающимся (splay) деревьям. Основные операции над этими деревьями рассматриваются по шагам, с комментариями и иллюстрациями. Приводятся оценки сложности операций с доказательствами. Приводится описание реализации этих операций на псевдокоде. В конце каждого раздела предлагается набор задач теоретического характера для самостоятельного решения. Последний раздел пособия посвящен сравнению рассмотренных видов деревьев и приведены примеры их практического использования.

Пособие предназначено для студентов и преподавателей лекционного курса «Алгоритмы и алгоритмические языки» и поддерживающего его курса «Практикум на ЭВМ».

Автор приносит глубокую благодарность рецензентам за ценные замечания, а также Кириллу Батузову за помощь при подготовке пособия.

2. Двоичные деревья поиска

С развитием компьютерной техники проблема хранения и обработки больших объемов данных становилась все более актуальной. Возникла необходимость организации хранилища для больших объемов данных, которое предоставляет возможность быстро находить и модифицировать данные. Один из способов организации такого хранилища — двоичные деревья поиска. Эту структуру данных можно описать на любом языке программирования, в котором есть составные типы данных.

Двоичное дерево представляет собой в общем случае неупорядоченный набор узлов, который

- либо пуст (пустое дерево)
- либо разбит на три непересекающиеся части:
  - узел, называемый корнем;
  - двоичное дерево, называемое левым поддеревом;
  - двоичное дерево, называемое правым поддеревом.

Таким образом, двоичное дерево — это рекурсивная структура данных.

Каждый узел двоичного дерева можно представить в виде структуры данных, состоящей из следующих полей:

- данные, обладающие ключом, по которому их можно идентифицировать;
- указатель на левое поддерево;
- указатель на правое поддерево;
- указатель на родителя *(необязательное поле).*

Значение ключа уникально для каждого узла.

Описание узла двоичного дерева на языке программирования может выглядеть следующим образом:

C	Pascal

struct BTNode
{
key_type key;
struct BTNode *left;
struct BTNode *right;
struct BTNode *parent;
};
	type
	Tree = ^BTNode;
	BTNode = record
	key: key_type;
	left: Tree;
	right: Tree;
	parent: Tree;
	end;

Дерево поиска – это двоичное дерево, в котором узлы упорядочены определенным образом по значению ключей: для любого узла x

значения ключей всех узлов его левого поддерева меньше значения ключа x, а значения ключей всех узлов его правого поддерева больше значения ключа x. Поэтому для key_type, типа значений ключей, должна быть определена операции сравнения *меньше*.

Рассмотрим выполнение основных операций над деревьями поиска: поиск узла в дереве, вставка узла в дерево, удаление узла из дерева.

Далее, для простоты будем считать, что ключ и данные – это одно и то же, и ключ имеет целочисленный тип.

2.1 Поиск узла в двоичном дереве поиска

Процедура поиска узла по ключу заключается в том, что на каждом шаге значение искомого ключа сравнивается со значением ключа рассматриваемого узла, начиная с корня. Если значение искомого ключа меньше, чем значение ключа рассматриваемого узла, то поиск продолжается в левом поддереве, если больше — то в правом поддереве. И так, пока не будет найден узел с искомым ключом (см. Рис. 1(а)) или пока поиск не достигнет того узла, ниже которого этот узел не может находиться. Если при поиске мы обнаруживаем, что узел далее надо искать, например, в правом поддереве, а оно пусто (как на Рис. 1(б)), следовательно, мы можем сделать вывод, что искомого ключа в дереве нет.

Таким образом, алгоритм следует из определения дерева поиска.

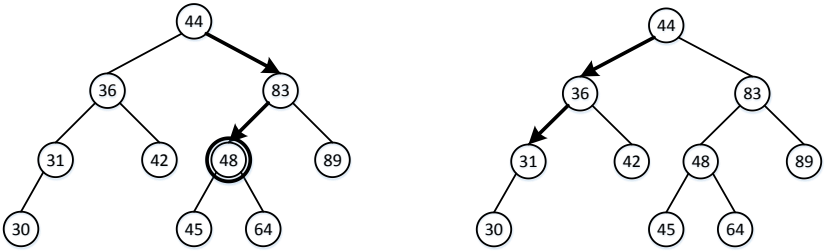


Рис. 1. Поиск узла по ключу. Направление поиска показано стрелками: (а) поиск узла 48; (б) поиск узла 32

Приведем рекурсивную реализацию операции поиска узла по ключу в двоичном дереве поиска.

TREE-SEARCH(T, k)

/ На вход подается дерево T, в котором производится поиск, и k – значение ключа. Возвращается узел дерева, в котором находится искомый ключ, или NULL, если узла с искомым ключом в дереве нет. */*

```
1 x ← root[T]
2 if x = NULL или k = key[x] then /* нерекурсивная ветвь */
3       return x
4 if k < key[x] then
5       return TREE-SEARCH(left[x], k) /* вызываем функцию для левого поддерева */
6 else
7       return TREE-SEARCH(right[x], k) /* вызываем функцию для правого поддерева */
```

Более эффективная по времени и используемой памяти итеративная реализация операции поиска:

INTERATIVE-TREE-SEARCH (T, k)

/ На вход подается дерево T, в котором производится поиск, и k – значение ключа. Возвращается узел дерева, в котором находится искомый ключ, или NULL, если узла с искомым ключом в дереве нет. */*

```
1 x ← root[T]
2 while x ≠ NULL и k ≠ key[x] do
/* Просматриваем текущий узел, спускаясь по дереву вниз, пока не найдем искомый ключ или не дойдем до пустого поддерева. */
3       if k < key[x] then
4               x ← left[x] /* присваиваем x его левого сына */
5       else
6               x ← right[x] /* присваиваем x его правого сына */
7 return x
```

Эффективность по времени и памяти по сравнению с рекурсивной реализацией достигается за счет того, что не нужно на каждом шаге заново вызывать функцию и хранить информацию, связанную с текущим ее вызовом.

Определение 1: *Высота дерева – это максимальная длина пути от корня до листа.*

При поиске узла на каждой итерации мы спускаемся на один уровень вниз, поэтому время поиска узла в двоичном дереве поиска, то есть, количество шагов, ограничено сверху высотой этого дерева. Используя O-символику, можно обозначить время поиска в худшем случае как O(h), где h – высота дерева.

Если требуется найти в дереве узел с минимальным (максимальным) ключом, то, учитывая свойства дерева поиска, можно заметить, что это будет самый левый (самый правый) узел в дереве. Найти этот узел можно, двигаясь от корня только налево (направо) до тех пор, пока у рассматриваемого узла существует левый (правый) сын. На Рис. 1 минимальный ключ 30 находится в самом левом узле, а максимальный ключ – 89 – в самом правом узле. Поэтому операции поиска будут выглядеть следующим образом:

TREE-MINIMUM(T)

/ На вход подается дерево T, и возвращается узел с минимальным значением ключа в дереве. */*
1 x ← root[T]
2 **while** left[x] ≠ NULL **do** */* ищем самый левый узел в дереве */*
3 x ← left[x]
4 **return** x

TREE-MAXIMUM(T)

/ На вход подается дерево T, и возвращается узел с максимальным значением ключа в дереве. */*
1 x ← root[T]
2 **while** right[x] ≠ NULL **do** */* ищем самый правый узел в дереве */*
3 x ← right[x]
4 **return** x

Время их выполнения также будет O(*h*).

Еще одна процедура поиска, которая может потребоваться, – поиск последователя узла, то есть, узла со следующим по значению ключом, который имеется в дереве. Если правое поддерево узла x с ключом k не пусто, то последователем узла x будет самый левый узел его правого поддерева, потому как это он имеет минимальный ключ среди всех ключей, больших k, имеющихся в дереве. Например, в дереве на Рис. 1 у узла с ключом 44 последователем является узел с ключом 45. Если правое поддерево узла x пусто, то надо искать, поднимаясь наверх, первого предка, для которого узел x окажется в левом поддереве. В примере на Рис. 1 у узла с ключом 42 нет правого поддерева. Поднимаясь наверх, мы видим его родителя – узел с ключом 36, для которого узел с ключом 42 – правый сын, поэтому узел с ключом 36 не может быть последователем. Поднимаясь далее, мы приходим в узел с ключом 44, для которого все предыдущее поддерево – левое. 44 больше и чем 36, и чем 42, значит, последователь найден.

TREE-SUCCESSOR(T, x)

/ На вход подается дерево T и узел x, и возвращается узел, который является его последователем. Если у узла нет последователя, то есть, узел является самым правым в дереве, то возвращается NULL. */*
1 **if** right[x] ≠ NULL **then**
/ Если у узла есть правое поддерево, то возвращаем самый левый узел правого поддерева. */*
2 **return** TREE-MINIMUM(right[x])
/ Если у узла нет правого поддерева, то поднимаемся по родителям наверх, пока не найдем того, для которого рассматриваемый узел – левый сын. */*
3 y ← parent[x]
4 **while** y ≠ NULL и x = right[y] **do**
5 x ← y
6 y ← parent[y]
7 **return** y

Как видно из приведенного выше фрагмента кода, данная задача имеет элегантное решение при наличии в описании узла дерева необязательного поля – указателя на родительский узел.

Время выполнения операции поиска последователя будет O(h), потому как оно ограничено высотой дерева – поиск идет либо только вниз (когда ищем самый левый узел правого поддерева), либо только вверх (когда ищем первого предка, для которого узел x окажется в левом поддереве).

2.2 Вставка узла в двоичное дерево поиска

Вставка узла в двоичное дерево поиска выполняется после того, как найдено место, куда можно вставить новый узел так, чтобы сохранились свойства дерева поиска. Для этого выполняется поиск узла с этим ключом в дереве. Если узел с таким ключом в дереве уже имеется, то вставку выполнять не нужно. В противном случае поиск остановится на некотором узле, к которому впоследствии будет подсоединяться узел слева или справа в зависимости от значения его ключа. Так, в примере на Рис. 1 для вставки узла с ключом 32 нужно запустить процедуру поиска, которая остановится на узле с ключом 31, потому как поиск должен продолжаться в правом поддереве, а оно пусто. Затем узел 32 присоединится к нему справа.

Опишем модифицированную операцию поиска, которая, если узел с искомым ключом не найден, возвращает не NULL, а тот узел, на котором остановился поиск.

TREE-SEARCH-INEXACT(T, k)

/* *На вход подается дерево T, в котором производится поиск, и k – значение ключа. Возвращается узел дерева, в котором находится искомый ключ, или тот узел, на котором остановился поиск.* */
1 y ← NULL
2 x ← root[T]
3 **while** x ≠ NULL и k ≠ key[x] **do** /* *продолжаем поиск до тех пор, пока не найдем ключ или не дойдем до пустого дерева* */

```
4       y ← x
5       if k < key[x] then
6               x ← left[x]
7       else
8               x ← right[x]
9  if x ≠ NULL /* ключ найден */
10      y ← x /* кладем в y узел x с ключом k вместо его родителя */
11 return y
```

```
TREE-INSERT(T, z)

/* На вход подается дерево T и узел z, который надо добавить в
дерево. */
1   y ← TREE-SEARCH-INEXACT(T, key[z])
2   parent[z] ← y
3 if y = NULL then /* если дерево было пусто, то z станет корнем */
4       root[T] ← z
/* Присоединяем z к y слева или справа в зависимости от значения
ключа. */
5 else if key[z] < key[y] then
6       left[y] ← z
7 else
8       right[y] ← z
```

Вставка в двоичное дерево поиска требует O(h) шагов для выполнения поиска и еще O(1) (константное значение) шагов для выполнения непосредственно операции вставки, поэтому итоговое время выполнения – O(h).

2.3 Удаление узла из двоичного дерева поиска

При удалении узла из двоичного дерева поиска необходимо рассмотреть три случая:

1. У узла нет сыновей – узел является листовым.

В этом случае узел удаляется, и соответствующее поддерево его родителя становится пустым (Рис. 2(а)).

2. У узла только один сын.

Узел удаляется, и его сын переходит к его родителю (Рис. 2(б)).

3. У узла два сына.

Пусть B – удаляемый узел. Ищем его последователя C – узла с минимальным ключом в правом поддереве удаляемого узла. Переносим ключ узла C в узел B и сводим задачу к удалению узла C (Рис. 2(в)). Эта процедура является корректной, потому что после удаления узла B его место должен занять как раз его последователь. Время поиска последователя, как было показано выше, ограничено высотой дерева.

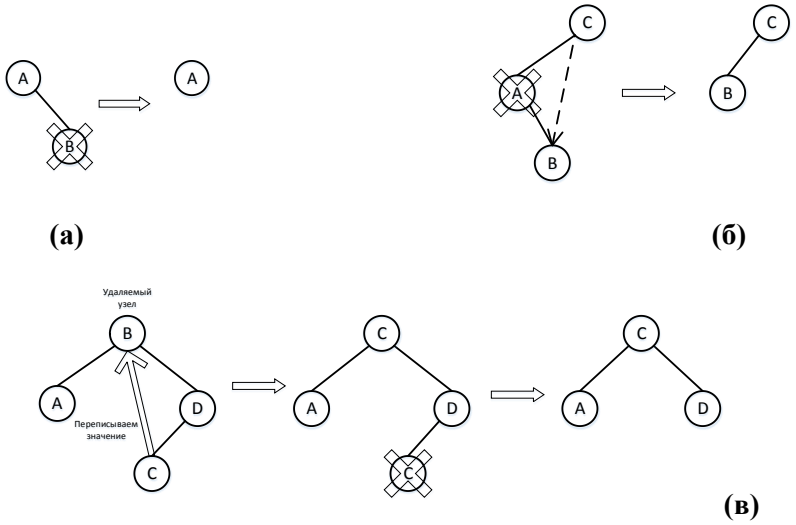


Рис. 2. Удаление узла из двоичного дерева поиска: (а) у узла нет сыновей; (б) у узла один сын; (в) у узла два сына

Ниже приведена реализация функции удаления узла из двоичного дерева поиска. Узел исключается из дерева с помощью связывания между собой его сына (если есть) и отца (если есть). Память, занимаемая узлом, не освобождается, и этот узел возвращается функцией, чтобы затем его можно было использовать в других структу-

рах. Если узел больше использоваться не будет, то память можно освободить.

TREE-DELETE(T, z)

```
/* На вход подается дерево T и узел z, который надо удалить. Возвращается удаленный узел. */
1  if left[z] = NULL или right[z] = NULL then
/* Если у узла z нет сыновей или один сын, то удаляемым узлом y будет он сам. */
2        y ← z
3  else
/* Если у узла z два сына, то удаляемым узлом y будет его последователь. У узла y может быть только правый сын, потому что это самый левый элемент правого поддерева узла z. Таким образом, у узла y в любом случае только один сын.*/
4        y ← TREE-SUCCESSOR(z)
/* В строках 5–16 прикрепляем x (сына y, если есть), к отцу y, либо делаем x корнем дерева. */
5  if left[y] ≠ NULL then /* неприменимо, если y – последователь z */
6        x ← left[y]
7  else
8        x ← right[y]
9  if x ≠ NULL then
10       parent[x] ← parent[y]
11 if parent[y] = NULL then
12       root[T] ← x
13 else if y = left[parent[y]] then
14       left[parent[y]] ← x
15 else
16       right[parent[y]] ← x
17 if y ≠ z then
/* Если удаляемым узлом оказался не z, а его последователь, то копируем в z ключ из y. */
18       key[z] ← key[y]
19 return y
```

Удаление из двоичного дерево поиска требует O(h) шагов для поиска удаляемого узла, также может потребоваться дополнительно O(h) шагов для поиска его последователя, и O(1) шагов для выполнения непосредственно операции удаления узла. Поэтому итоговое время выполнения – O(h).

2.4 Балансировка двоичного дерева поиска

Как было показано выше, время выполнения базовых операций в дереве поиска линейно зависит от его высоты. Но из одного и того же набора ключей можно построить разные деревья поиска, как показано на Рис. 3.

В приведенном примере на Рис. 3 оба дерева построены из одного и того же набора ключей, но высота первого дерева больше, поэтому время выполнения операций над ним будет больше. Например, при поиске узла с ключом 6 в случае (а) требуется просмотреть все шесть узлов, а в случае (б) только три узла: 3->5->6. Второе дерево лучше *сбалансировано*, чем первое. В этом случае хорошо видно преимущество двоичного дерева поиска над обычным двоичным деревом. Если бы дерево на Рис. 3(б) не было деревом поиска, то при поиске узла с ключом 6 надо было бы просмотреть от 4 до 6 узлов, в зависимости от порядка обхода дерева. А в дереве поиска при поиске узла движение происходит только вниз, поэтому количество шагов ограничено высотой дерева. Чем больше узлов в дереве, тем более очевидно преимущество дерева поиска над обычным двоичным деревом при условии, что дерево поиска хорошо сбалансировано.

Как построить дерево поиска минимальной высоты? Если набор ключей известен заранее, то его надо упорядочить. Корнем поддерева становится узел с ключом, значение которого – медиана этого набора. Для упорядоченного набора, содержащего нечетное количество ключей, – это ключ, находящийся ровно в середине набора. Для набора 1,2,3,4,5,6 из примера на Рис. 3, который содержит четное количество ключей, в качестве медианы может быть вы-

бран ключ как со значением 3, так и со значением 4. Для определенности выберем 3. Узел с этим ключом будет корнем дерева. Далее, ключи, меньшие 3, попадут в левое поддерево, а ключи, большие 3, попадут в правое поддерево. Для построения левого и правого поддеревьев проделываем ту же процедуру на соответствующих наборах ключей. И так до тех пор, пока все ключи не будут включены в дерево. На Рис. 3(б) для каждого узла указан набор ключей, для которых строится дерево с корнем в этом узле, и выделена медиана этого набора, которая и попадает в корень. Дерево, изображенное на Рис. 3(б) является *идеально сбалансированным*, то есть, для каждого его узла количество узлов в левом и правом поддеревьях различается не более, чем на 1. Такое дерево можно построить для любого набора ключей. Достаточно заметить, что при выборе медианы на каждом шагу при построении дерева набор ключей разбивается на две части, и количество ключей в них может различаться не более, чем на 1.

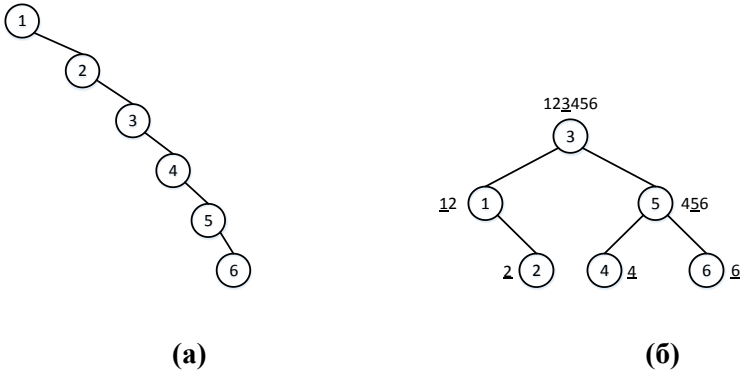


Рис. 3. Разные деревья поиска, построенные из одного и того же набора ключей

Полный набор ключей не всегда известен заранее. Если ключи поступают по очереди, то построение дерева поиска будет зависеть от порядка их поступления. Если, например, ключи будут поступать в порядке 1,2,3,4,5,6, то получится дерево, изображенное на Рис. 3(а). Высота такого дерева максимальна для этого набора

ключей, следовательно, и время выполнения операций над ним также будет максимальным. Поэтому при добавлении очередного узла, возможно, дерево понадобится перестраивать, чтобы уменьшить его высоту, сохраняя тот же набор узлов. Идеальную балансировку поддерживать сложно. Если при добавлении очередного узла количество узлов в левом и правом поддеревьях какого-либо узла дерева станет различаться более, чем на 1, то дерево не будет являться идеально сбалансированным, и его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска. Поэтому обычно требования к сбалансированности дерева менее строгие.

В настоящем пособии рассматриваются три основных вида сбалансированных деревьев поиска: АВЛ-деревья, красно-черные деревья и самоперестраивающиеся деревья (splay-деревья). Для АВЛ-деревьев сбалансированность определяется разностью высот правого и левого поддеревьев любого узла. Если эта разность по модулю не превышает 1, то дерево считается сбалансированным. Данное условие проверяется после каждого добавления или удаления узла, и определен минимальный набор операций перестройки дерева, который приводит к восстановлению свойства сбалансированности, если оно оказалось нарушено. В красно-черных деревьях каждый узел имеет дополнительное свойство – цвет, красный или черный. На дерево наложены ограничения по расположению и количеству узлов в зависимости от цвета, и определен набор операций перестройки дерева в случае нарушения этих ограничений после добавления или удаления узла. Если АВЛ-деревья накладывают достаточно строгие условия на сбалансированность, и при добавлении узлов дерево достаточно часто приходится перестраивать, то в красно-черном дереве у каждого узла высоты левого и правого поддеревьев могут отличаться не более, чем в два раза. И, наконец, третий вид рассматриваемых сбалансированных деревьев поиска – самоперестраивающиеся деревья. В отличие от двух предыдущих видов, у этих деревьев нет никаких ограничений на расположение узлов, а сбалансированность в среднем достигается за счет того, что каждый раз перед выполнением операции над уз-

лом этот узел перемещается в корень дерева. Но есть вероятность того, что дерево может оказаться не сбалансированным, как, например, на Рис. 3(а).

3. АВЛ-деревья

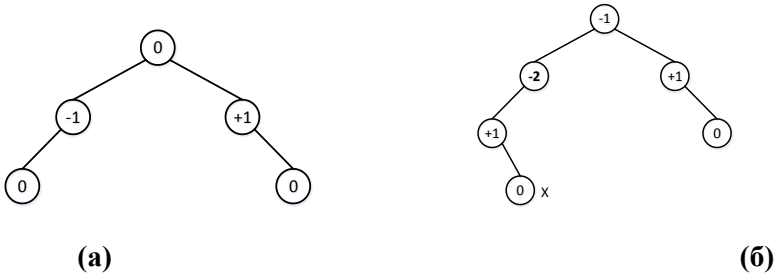
АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на 1. Эта структура данных разработана советскими учеными Адельсон-Вельским Георгием Максимовичем и Ландисом Евгением Михайловичем в 1962 году. Аббревиатура АВЛ соответствует первым буквам фамилий этих ученых. Первоначально АВЛ-деревья были придуманы для организации перебора в шахматных программах. Советская шахматная программа «Каисса» стала первым официальным чемпионом мира в 1974 году.

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранится показатель баланса – разность высот правого и левого поддеревьев. В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо.

C	Pascal

struct AVLNode
{
key_type key;
struct AVLNode *left;
struct AVLNode *right;
struct AVLNode *parent;
int balance; // показатель баланса
};
	type
	AVLTree = ^AVLNode;
	AVLNode = record
	key: key_type;
	left: AVLTree;
	right: AVLTree;
	parent: AVLTree;
	balance: integer;
	end;

На Рис. 4(а) приведен пример АВЛ-дерева. Таким образом, получается, что в АВЛ-дереве показатель баланса balance для каждого узла, включая корень, по модулю не превосходит 1. На Рис. 4(б) приведен пример дерева, которое не является АВЛ-деревом, поскольку в одном из узлов баланс нарушен, т.е. |balance|>1. Здесь и далее для АВЛ-деревьев будем указывать в узле значение показателя баланса.



(а) **(б)**

Рис. 4. (а) пример АВЛ-дерева; (б) пример дерева, не являющегося АВЛ-деревом: в узле X сбалансированность нарушена

3.1 Вставка узла в АВЛ-дерево

Рассмотрим, что происходит с АВЛ-деревом при добавлении нового узла. Сначала узел добавляется в дерево с помощью стандартного алгоритма вставки в двоичное дерево поиска. Показатели баланса в ряде узлов при этом изменятся, и сбалансированность может нарушиться. Сбалансированность считается нарушенной, если показатель баланса по модулю превысил 1 в одном или нескольких узлах.

При добавлении нового узла разбалансировка может произойти сразу в нескольких узлах, но все они будут лежать на пути от этого добавленного узла к корню, как показано на Рис. 5.

Тем не менее, перестраивать будем поддерево с корнем в том из этих узлов, который является ближайшим к добавленному. В примере на Рис. 5 этот узел обведен кружком. Общее правило для всех добавляемых узлов, приводящих к разбалансировке: чтобы найти

корень поддерева, которое понадобится перестраивать, надо подниматься вверх по дереву от вновь добавленного узла до тех пор, пока не найдется *первый* узел, в котором нарушена сбалансированность. Назовем его *опорным узлом*. Таким образом, искать этот узел надо, поднимаясь наверх, а не спускаясь вниз от корня. После того как опорный узел будет найден, будет проведена процедура перестройки поддерева с корнем в этом узле с целью восстановления его сбалансированности. Остальная часть дерева останется в прежнем виде. При этом все дерево также станет сбалансированным — показатель баланса не будет превышать 1 по модулю во *всех* узлах дерева. После демонстрации процедуры балансировки вам будет предложено обосновать этот факт самостоятельно.

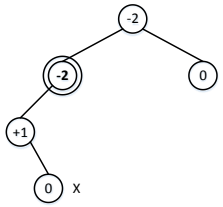


Рис. 5. Добавление узла X в АВЛ-дерево привело к разбалансировке в двух узлах. Но перестраивать будем только поддерево с корнем в выделенном узле

В зависимости от того, в какое поддерево опорного узла был добавлен новый узел, рассматривается четыре случая, которые можно разбить на две пары симметричных друг другу случаев. В каждом из них баланс восстанавливается с помощью одного или двух *поворотов*. На Рисунках 6–9 опорный узел обведен кружком. Вновь добавленный узел обозначен буквой X.

1. Добавление в *левое* поддерево *левого* сына опорного узла.

Необходимо произвести *правый* поворот (R): опорный узел (B) поворачивается направо относительно своего левого сына (A).

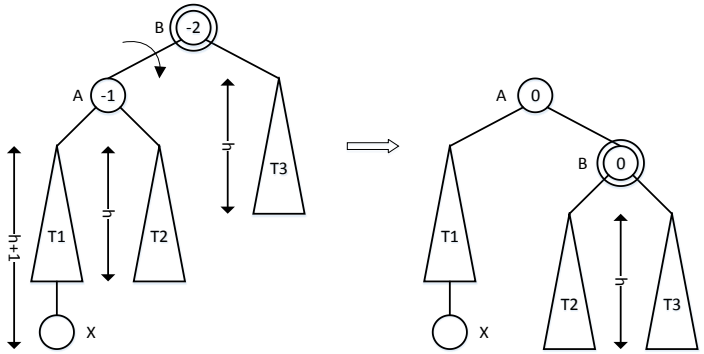


Рис. 6. Добавление узла в левое поддерево левого сына опорного узла и балансировка – правый поворот

Ситуация, требующая правого поворота после добавления нового узла, схематично изображена на Рис. 6(а). Поддеревья T1, T2 и T3 могут быть пустыми, но они обязательно должны иметь одинаковую высоту. Тогда до добавления узла X у опорного узла B высота левого поддерева будет на 1 больше, чем высота правого поддерева, а после добавления X баланс нарушится именно в B, а не в A. После поворота направо (по часовой стрелке) вокруг узла A узел B вместе с поддеревом T3 опустится на два уровня вниз относительно узла A. Самый нижний узел поддерева T3 окажется на одном уровне с узлом X. Так как поддерево с корнем в B станет новым правым сыном A, его бывший правый сын T2 перейдет к B. Высота T2 равна высоте T3, поэтому и в B, и в A показатель баланса станет равен 0.

TREE-ROTATE-R(T, x)

/ На вход подается дерево T и опорный узел x. */*
1 y ← left[x]
/ В строках 2–4 присоединяем T2 к x слева. */*
2 left[x] ← right[y]
3 **if** right[y] ≠ NULL **then**
4 parent[right[y]] ← x
/ В строках 5–11 отсоединяем x от его родителя и присоединяем y вместо x. */*

```
5  parent[y] ← parent[x]
6  if parent[x] = NULL then
7        root[T] ← y
8  else if x = right[parent[x]] then
9        right[parent[x]] ← y
10 else
11       left[parent[x]] ← y
/* Соединяем x и y. */
12 right[y] ← x
13 parent[x] ← y
```

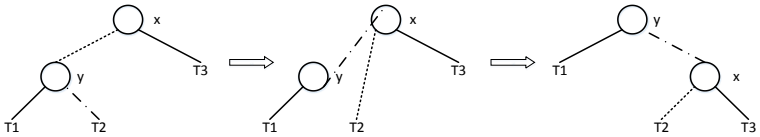
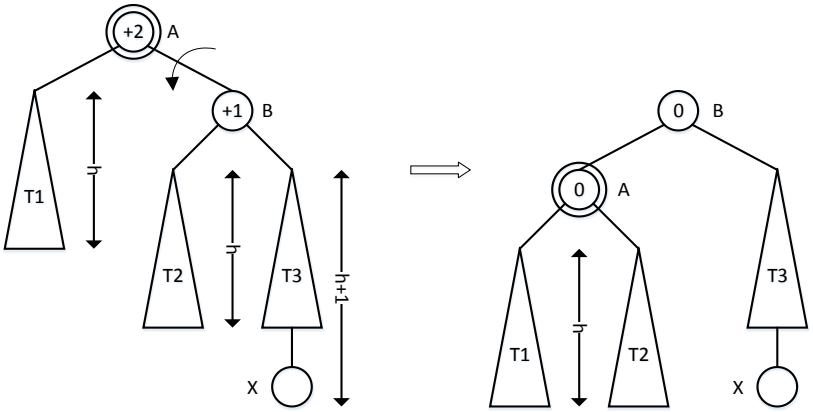


Рис. 13. Иллюстрация функции TREE-ROTATE-R(T, x)

2. Добавление в *правое* поддерево *правого* сына опорного узла.

Случай, симметричный предыдущему.



(а) **(б)**

Рис. 7. Добавление узла в правое поддерево правого сына опорного узла и балансировка – левый поворот

3. Добавление в *правое* поддерево *левого* сына опорного узла.

Необходимо произвести *двойной* поворот — *налево, потом направо* (LR): сначала левый сын опорного узла (A) поворачивается *налево* относительно своего правого сына (B), а затем опорный узел (C) поворачивается *направо* относительно своего нового левого сына (B).

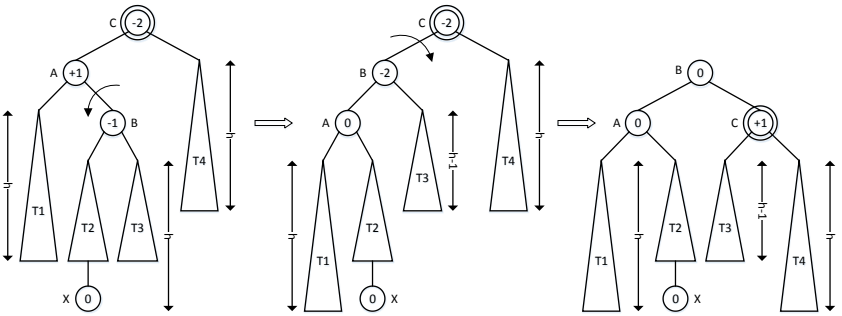


Рис. 8. Добавление узла в правое поддерево левого сына опорного узла и балансировка – левый-правый поворот

На Рис. 8(а) правым поддеревом левого сына опорного узла является поддерево с корнем в B. При этом узел X может быть добавлен как в поддерево T2, так и в поддерево T3 – тип поворота при балансировке от этого не изменится. Если до добавления узла X в правое поддерево узла A это правое поддерево было пусто, то в роли B выступает сам узел X. В результате двойного поворота узел B окажется наверху комбинации узлов ABC. За один поворот это сделать нельзя, потому как в начальный момент узел B является самым нижним в комбинации ABC. Поэтому первый поворот (A относительно B налево) поднимает B на один уровень вверх относительно C, а второй поворот (C относительно B направо) поднимает B еще на один уровень вверх относительно C. В итоге, слева у B окажется A с поддеревом T1, а справа у B окажется C с поддеревом T4. Высоты поддеревьев T1 и T4 совпадают. Поддерево T2 с узлом X и поддерево T3 в соответствии со свойствами дерева поиска прикрепятся к узлам A и C соответственно. Поскольку их вы-

соты отличаются от высот T1 и T4 не более, чем на 1, баланс во всех узлах – A, B, C – по модулю не превысит 1. Баланс в B будет равен 0.

TREE-ROTATE-LR(T, x)
/ На вход подается дерево T и опорный узел x. */*
1 TREE-ROTATE-L(T, left[x])
2 TREE-ROTATE-R(T, x)

4. Добавление в *левое* поддерево *правого* сына опорного узла.

Случай, симметричный предыдущему.

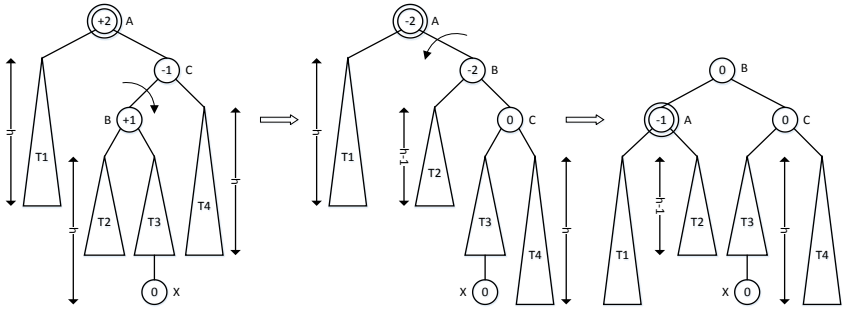
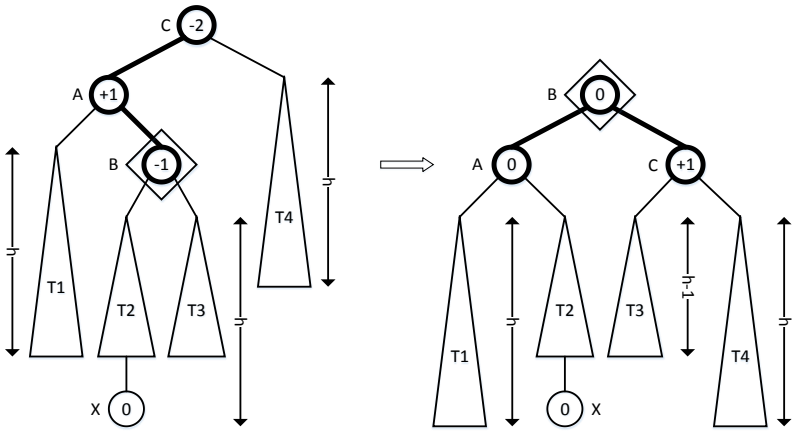
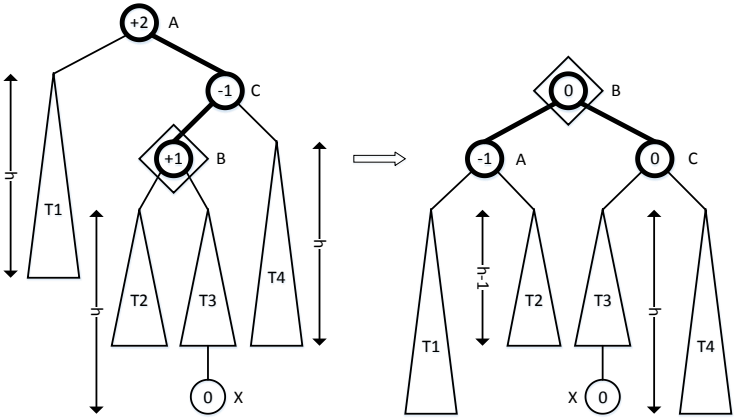


Рис. 9. Добавление узла в левое поддерево правого сына опорного узла и балансировка – правый-левый поворот

Чтобы проще было запомнить, как производится двойной поворот и не выписывать промежуточное дерево, достаточно обратить внимание на вращение комбинации из трех узлов. Эта комбинация включает в себя опорный узел и корни тех поддеревьев, куда добавился новый узел, нарушивший баланс. Если, например, добавляем узел в левое поддерево правого сына опорного узла, то комбинация будет состоять из опорного узла, правого сына опорного узла и левого сына правого сына опорного узла. Тот узел, который был самым нижним в этой комбинации, после двойного поворота становится самым верхним, независимо от того, выполняется правый-левый поворот или левый-правый (см. Рис. 10).



(а)



(б)

Рис. 10. Двойной поворот: самый нижний узел в «треугольнике» становится самым верхним (обозначен ромбиком):
(а) левый-правый поворот; (б) правый-левый поворот

Резюмируя правила поворотов при выполнении операции балансировки АВЛ-деревьев, отметим общее правило: если добавление нового узла, приводящее к разбалансировке, происходит в левое

поддерево левого сына опорного узла или в правое поддерево правого сына опорного узла, т.е. если стороны сына и внука опорного узла одноименны, то необходимо произвести одинарный поворот. Если добавление происходит в правое поддерево левого сына опорного узла или в левое поддерево правого сына опорного узла, т.е. стороны разноименны, то необходимо произвести двойной поворот. Мнемоническое правило для запоминания того, в какую сторону производится поворот:

- добавление в *левое* поддерево *левого* сына опорного узла – *правый* (R);
- добавление в *правое* поддерево *левого* сына опорного узла – *левый-правый* (LR);
- добавление в *левое* поддерево *правого* сына опорного узла – *правый-левый* (RL);
- добавление в *правое* поддерево *правого* поддерева сына опорного узла – *левый* (L).

Определение 2: *Глубина узла равна длине простого пути от корня до этого узла.*

Таким образом, поворот производится в *противоположную* сторону. Визуально это правило выглядит так, что если самый глубокий узел (тот, который был добавлен последним) находится слева или справа, то производится одинарный поворот опорного узла относительно поддерева, содержащего этот узел, в противоположную сторону, чтобы выровнять высоты. Если самый глубокий узел находится посередине, то потребуется двойной поворот.

Ниже приведена реализация операции вставки узла в АВЛ-дерево через вспомогательную процедуру восстановления баланса.

AVL-RESTORE-BALANCE(T, x)

/ На вход подается дерево T и узел x, в котором надо восстановить баланс, если он был нарушен. */*

1 **if** balance[x] < –1**then** */* у узла x высота левого поддерева больше высоты правого поддерева */*

```
2       if height[left[left[x]]] > height[right[left[x]]] then /* самый
глубокий узел слева */
3               TREE-ROTATE-R(T, x)
4       else /* самый глубокий узел посередине */
5               TREE-ROTATE-LR(T, x)
6   if balance[x] > 1 then /* у узла x высота правого поддерева
больше высоты левого поддерева */
7       if height[right[right[x]]] > height[left[right[x]]] then /* самый
глубокий узел справа */
8               TREE-ROTATE-L(T, x)
9       else /* самый глубокий узел посередине */
10              TREE-ROTATE-RL(T, x)
```

```
AVL-INSERT(T, x)

/* На вход подается дерево T и узел x, который надо добавить в
дерево. */
1 TREE-INSERT(T, x)
/* Поскольку мы не знаем, где именно находится опорный узел, в
строках 2–5 проходим по всем узлам, лежащим на пути от x к
корню, и вызываем процедуру восстановления баланса. */
2 current ← x
3 while current ≠ NULL do
4       AVL-RESTORE-BALANCE(T, current)
5       current ← parent[current]
```

3.2 Удаление узла из АВЛ-дерева

Непосредственно удаление узла из АВЛ-дерева происходит так же, как и удаление узла из обычного двоичного дерева поиска, в соответствии с алгоритмом, описанном в Разделе 2.3. Таким образом, если у узла менее двух сыновей, то удаляется сам узел, а если два сына, то удаляемым узлом становится его последователь, информация (ключ) из которого предварительно переписывается в удаляемый узел. Чтобы после удаления сохранились свойства АВЛ-дерева, возможно, понадобится выполнить балансировку. Для это-

го надо подниматься вверх по пути от удаленного узла к корню и проверять в этих узлах баланс. Если в узле баланс нарушен, то надо выполнить соответствующий поворот – одинарный или двойной. Остановить просмотр можно на том узле, в котором показатель баланса не поменялся. Это означает, что высота его поддерева, левого или правого, в котором производилось удаление, не изменилась.

При балансировке после удаления используются те же виды поворотов, что и после вставки узла в дерево. Рассмотрим, как определить тип поворота. На иллюстрациях ниже показано, какой поворот восстановит баланс в ближайшем разбалансированном узле к удаляемому (опорном узле). Используя эти правила, можно будет определять тип поворота и при подъеме наверх по дереву, если баланс в родителях будет нарушаться после текущей балансировки.

Правила выполнения поворотов при удалении узла из АВЛ-дерева следующие.

1. *Левое* поддерево стало *ниже правого* на 2 уровня.

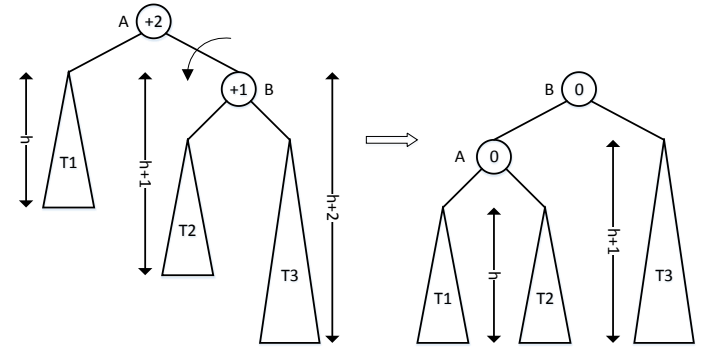
a. **У *правого* сына (B) высота *правого* поддерева *больше, либо равна* высоте *левого* поддерева (height(T3) ≥ height(T2)),**

Необходимо произвести *левый* поворот (L): опорный узел (A) поворачивается *налево* относительно своего правого сына (B).

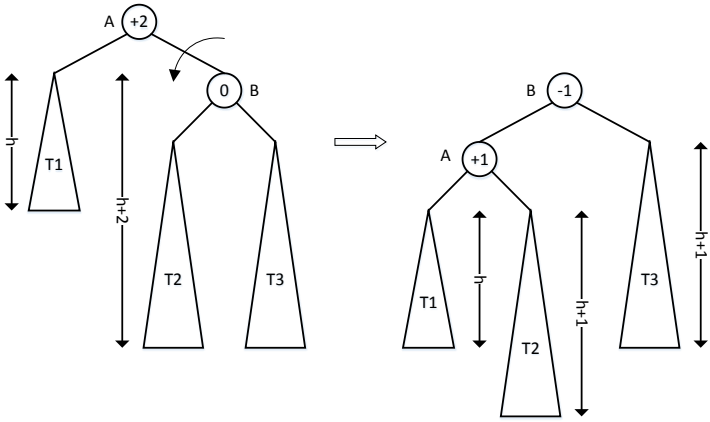
b. **У *правого* сына (C) высота *правого* поддерева *меньше* высоты *левого* поддерева (height(T4) < height(B)).**

Необходимо произвести *двойной* поворот — *направо, потом налево* (RL): сначала правый сын опорного узла (C) поворачивается *направо* относительно своего левого сына (B), а затем опорный узел (A) поворачивается *налево* относительно своего нового правого сына (B).

Поддерево с корнем в B должно иметь высоту h+1. При этом высоты поддеревьев T2 и T3 могут быть равны h, а могут различаться на 1: либо h и h-1, либо h-1 и h.



(а)



(б)

Рис. 11. Левое поддерево короче правого и
(а) height(T3) > height(T2); (б) height(T3) = height(T2).
Балансировка: левый поворот

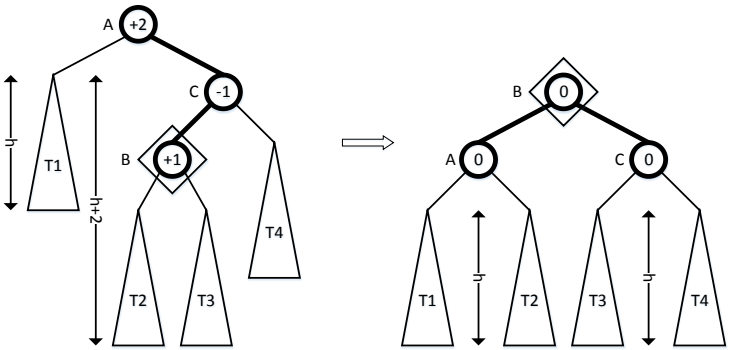


Рис. 12. Левое поддерево короче правого и height(T4) < height(B). Балансировка: двойной поворот

2. *Правое* поддерево стало *ниже левого* на 2 уровня (случай, симметричный случаю 1).

a. **У *левого* сына высота *левого* поддерева *больше, либо равна* высоте *правого* поддерева.**

Случай, симметричный случаю 1a.

b. **У *левого* сына высота *левого* поддерева *меньше* высоты *правого* поддерева.**

Случай, симметричный случаю 1b.

Здесь мнемоническое правило такое же, как и в случае добавления узла – если самые глубокие узлы находятся справа или слева, то производится одинарный поворот опорного узла в противоположную сторону, а если они находятся посередине, то производится двойной поворот.

Далее приведена реализация операции удаления узла из АВЛ-дерева с использованием описанной выше процедуры восстановления баланса.

AVL-DELETE(T, x)

/ На вход подается дерево T и узел x, который надо удалить из дерева. */*

```
1 deleted = TREE-DELETE(T, x)
/* В строках 2–5 проходим по всем узлам, лежащим на пути от
родителя удаленного узла к корню, и восстанавливаем в каждом
из них баланс, если он был нарушен. */
2 current ← parent[deleted]
3 while current ≠ NULL do
4       AVL-RESTORE-BALANCE(T, current)
5       current ← parent[current]
```

3.3 Оценка сложности поиска в АВЛ-дереве

Два крайних случая АВЛ-деревьев это:

(а) совершенное дерево – все узлы имеют показатель баланса 0;

(б) дерево Фибоначчи – все узлы, кроме листовых, имеют показатель баланса +1, либо все узлы, кроме листовых, имеют показатель баланса –1.

Примеры приведены на Рис. 14.

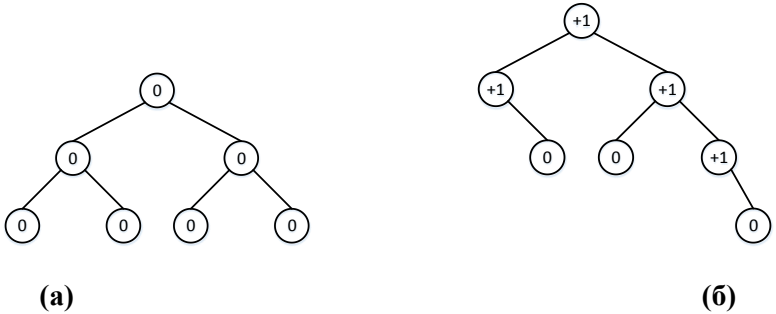


Рис. 14. Крайние случаи АВЛ-деревьев: (а) совершенное дерево; (б) дерево Фибоначчи

Не для каждого набора ключей можно построить совершенное дерево, равно как и не для каждого набора ключей можно построить дерево Фибоначчи. Но эти деревья позволяют оценить диапазон возможных высот АВЛ-деревьев. Совершенное дерево является частным случаем идеально сбалансированного дерева, поэтому

оно имеет минимально возможную высоту для данного количества узлов. Дерево Фибоначчи, напротив, имеет максимально возможную высоту для данного количества узлов, при условии, что сохраняются свойства АВЛ-дерева.

В совершенном дереве у каждого узла, кроме листовых, ровно два сына. Тогда количество узлов m равно $1 + 2 + 2^2 + \ldots$ Количество таких слагаемых равно количеству уровней в дереве, т.е. на единицу больше высоты этого дерева. Если количество слагаемых равно $h + 1$, то степень двойки в последнем слагаемом будет равна h:

$$m = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1.$$

Поэтому высота совершенного дерева вычисляется как

$$h = \log_2(m + 1) - 1 \qquad (1)$$

Покажем, что дерево Фибоначчи имеет минимально возможную высоту для заданного количества узлов среди всех АВЛ-деревьев. Если переформулировать свойсто дерева Фибоначчи, то получится, что при заданной высоте оно имеет минимальное количество узлов из всех возможных АВЛ-деревьев. Пусть W_h – множество АВЛ-деревьев заданной высоты h с минимально возможным количеством узлов, а w_h – количество узлов в каждом из этих деревьев. Тогда $w_0 = 1$ и $w_1 = 2$. Пусть T – некоторое дерево из множества W_h высоты $h \geq 2$, а T_L и T_R – его левое и правое поддеревья. Поскольку T имеет высоту h, то либо T_L, либо T_R имеет высоту $h - 1$. Не ограничивая общности рассуждений, предположим, что T_R имеет высоту $h - 1$. Поскольку T является АВЛ-деревом, и оба его поддерева T_L и T_R также являются АВЛ-деревьями, T_R является АВЛ-деревом высоты $h - 1$. Более того, оно должно АВЛ-деревом высоты $h - 1$ с минимально возможным количеством узлов, потому как иначе оно могло бы быть заменено деревом той же высоты, но с меньшим количеством узлов, чтобы все дерево T имело минимально возможное количество узлов. Поэтому $T_R \in W_{h-1}$. Аналогично, поскольку T является АВЛ-деревом, разность высот его левого и правого поддеревьев не должна превышать 1 по модулю, а высота дерева равна h, высота дерева T_L может быть равна либо $h - 1$, либо $h - 2$. Но T

должно иметь минимальное количество узлов, поэтому $T_L \in W_{h-2}$. Следовательно, $w_h = 1 + w_{h-2} + w_{h-1}$. Поскольку $w_0 = 1$ и $w_1 = 2$, первые несколько значений w_h будут равны 1, 2, 4, 7, 12, 20, ... Можно доказать по индукции, что $w_h = f_{h+3} - 1$.

Для оценки высоты дерева Фибоначчи потребуется привести некоторые выкладки.

Из определения дерева Фибоначчи следует, что для любого узла, кроме листовых, разность высот левого и правого поддеревьев равна 1 либо –1. Не ограничивая общности рассуждений, будем считать, что эта разность равна 1. Таким образом, если высота дерева равна *h*, то высоты левого и правого поддеревьев равны $h - 2$ и $h - 1$ соответственно. Это свойство выполняется для любого поддерева дерева Фибоначчи.

Определение 3: *Числа Фибоначчи – это элементы числовой последовательности, где первые два числа равны 1, а каждое последующее число равно сумме двух предыдущих*:

$$f_1 = 1, f_2 = 1, f_n = f_{n-1} + f_{n-2}, n >= 3, n \in Z. \quad (2)$$

Теорема 1. Число узлов в дереве Фибоначчи F_h высоты *h* равно $m(h) = f_{h+2} - 1$.

Доказательство (по индукции)

Базис: $m(0) = f_2 - 1 = 0$, $m(1) = f_3 - 1 = 1$.

Шаг: по определению $m(h) = m(h - 1) + m(h - 2) + 1$.

Имеем $m(h) = (f_{h+1} - 1) + (f_h - 1) + 1 = f_{h+2} - 1$, т.к. $f_{h+1} + f_h = f_{h+2}$ (следует из Определения 1). Теорема доказана.

Теорема 2. Пусть C_1 и C_2 таковы, что уравнение

$$r^2 - C_1 r - C_2 = 0 \quad (3)$$

имеет два различных корня r_1 и r_2, $r_1 \neq r_2$. Пусть последовательность a_n такова, что

$$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n. \quad (4)$$

Тогда выполняется соотношение

$$a_n = C_1 a_{n-1} + C_2 a_{n-2}. \quad (5)$$

Доказательство. r_1 и r_2 – корни уравнения (3), тогда $r_1^2 = C_1 r_1 + C_2$ и $r_2^2 = C_1 r_2 + C_2$.

Имеем:

$$C_1 a_{n-1} + C_2 a_{n-2} = C_1(\alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1}) + C_2(\alpha_1 r_1^{n-2} + \alpha_2 r_2^{n-2}) =$$
$$\alpha_1 r_1^{n-2}(C_1 r_1 + C_2) + \alpha_2 r_2^{n-2}(C_1 r_2 + C_2) =$$
$$\alpha_1 r_1^{n-2} r_1^2 + \alpha_2 r_2^{n-2} r_2^2 = \alpha_1 r_1^n + \alpha_2 r_2^n = a_n. \quad (6)$$

Теорема доказана.

Теорема 3. Пусть C_1 и C_2 таковы, что уравнение (3) имеет два различных действительных корня r_1 и r_2, $r_1 \neq r_2$.

Пусть последовательность a_n задана значениями своих первых членов a_0 и a_1 и рекуррентным соотношением (5). Тогда для некоторых α_1 и α_2 и любого n = 1,2,… выполняется соотношение (4).

Доказательство. Подберем такие α_1 и α_2, чтобы выполнялось соотношение (4):

$$a_0 = \alpha_1 + \alpha_2, \text{ и} \quad (7)$$

$$a_1 = \alpha_1 r_1 + \alpha_2 r_2. \quad (8)$$

Для этого рассмотрим совокупность (7) и (8) как систему линейных уравнений, которую надо решить относительно α_1 и α_2. Получим:

$$\alpha_1 = \frac{a_1 - a_0 r_2}{r_1 - r_2},$$

$$\alpha_2 = \frac{-a_1 + a_0 r_1}{r_1 - r_2}.$$

Соотношение (4) выполняется для a_0 и a_1. Действительно, $a_0 = \alpha_1 r_1^0 + \alpha_2 r_2^0$ и $a_1 = \alpha_1 r_1^1 + \alpha_2 r_2^1$. Докажем по индукции, что если соотношение выполняется для a_{n-2} и a_{n-1}, то оно выполняется для a_n. Для этого повторим в обратном порядке вывод (5) из (4),

который проводился при доказательстве Теоремы 2, т.е. выведем (4) из (5). Для этого необходимо выписать цепочку (6) в обратном порядке:

$$\alpha_1 r_1^{\,n} + \alpha_2 r_2^{\,n} = \alpha_1 r_1^{\,n-2} r_1^{\,2} + \alpha_2 r_2^{\,n-2} r_2^{\,2} =$$
$$\alpha_1 r_1^{\,n-2}(C_1 r_1 + C_2) + \alpha_2 r_2^{\,n-2}(C_1 r_2 + C_2) =$$
$$C_1(\alpha_1 r_1^{\,n-1} + \alpha_2 r_2^{\,n-1}) + C_2(\alpha_1 r_1^{\,n-2} + \alpha_2 r_2^{\,n-2}) = C_1 a_{n-1} + C_2 a_{n-2} = a_n.$$

Теорема доказана.

Применим доказанные теоремы к числам Фибоначчи. Соотношение (2) эквивалентно соотношению (5) при $C_1 = C_2 = 1$. Уравнение (3) при этом записывается как

$$r^2 - r - 1 = 0$$

и имеет корни $r_1 = \frac{1+\sqrt{5}}{2}$ и $r_2 = \frac{1-\sqrt{5}}{2}$. Следовательно, согласно Теореме 3:

$$f_0 = \alpha_1 + \alpha_2 = 0,$$

$$f_1 = \alpha_1(\frac{1+\sqrt{5}}{2}) + \alpha_2(\frac{1-\sqrt{5}}{2}) = 1,$$

$$\alpha_1 = \frac{1}{\sqrt{5}}, \alpha_2 = -\frac{1}{\sqrt{5}},$$

$$f_n = \alpha_1 r_1^n + \alpha_2 r_2^n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n.$$

Согласно Теореме 1

$$m(h) = f_{h+2} - 1 = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+2} - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^{h+2} - 1$$

$$\left|\frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^{h+2}\right| < 1$$

$$m(h)+1>\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+2}.$$

Введем обозначение $\gamma=\frac{1+\sqrt{5}}{2}$. Тогда

$$m(h)+1>\frac{1}{\sqrt{5}}\gamma^{h+2} \quad (9)$$

Логарифмируя обе части (9), получаем

$$\log_2(m(h)+1)>\log_2(\frac{1}{\sqrt{5}})+(h+2)\log_2\gamma,$$

$$\log_2(m(h)+1)>-\log_2(\sqrt{5})+(h+2)\log_2\gamma,$$

$$h+2<\frac{\log_2(m+1)}{\log_2\gamma}+\frac{\log_2\sqrt{5}}{\log_2\gamma},$$

откуда

$$h<1.44\log_2(m+1)-0.32. \quad (10)$$

Таким образом, учитывая оценку высоты совершенного дерева (1) и оценку высоты дерева Фибоначчи (10) получаем, что высота *h* АВЛ-дерева из *m* узлов находится в диапазоне

$$\log_2(m+1)\le h<1.44\log_2(m+1)-0.32. \quad (11)$$

Это соотношение (11) задает оценку количества сравнений при поиске узла в АВЛ-дереве на пути от корня к этому узлу. Если, например, в АВЛ-дереве 10^6 вершин, то его высота, а, следовательно, и сложность поиска узла в нем, не превысит 29.

3.4 Задачи

1. Обосновать, почему после поворота дерево поиска остается деревом поиска.

2. Обосновать, почему при вставке узла в АВЛ-дерево выполнения одинарного или двойного поворота поддерева с корнем в опорном узле (первый узел, в котором нарушен баланс, на пути от добавленного узла к корню) достаточно для восстановления баланса во всем дереве.

3. Доказать по индукции, что минимальное количество узлов в АВЛ-дереве заданной высоты h равно $f_{h+3} - 1$, где f_{h+3} – (h+3)-е число Фибоначчи.

4. Существуют ли два АВЛ-дерева, у которых высота h ($0 \le h \le 10$) одинакова, а число вершин различается на 800? Привести ответ («да» или «нет» и его обоснование). Замечание: пустое дерево имеет высоту 0, а дерево из одного узла – высоту 1.

5. Существуют ли два АВЛ-дерева, у которых высота h ($0 \le h \le 11$) одинакова, а число вершин различается на 1712? Привести ответ («да» или «нет» и его обоснование).

6. Пусть в АВЛ-дереве используется стандартный алгоритм поиска по ключу и пусть для нахождения некоторого ключа пришлось просмотреть 11 вершин дерева. Каково минимально возможное число вершин в таком дереве? Ответ обосновать.

7. Ключи 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 добавляются в изначально пустое АВЛ-дерево в некоторой последовательности. Существует ли такая последовательность добавления этих ключей, при которой не будет необходимости выполнять повороты при вставке? Ответ обосновать.

8. В первоначально пустое АВЛ-дерево были занесены (согласно стандартному алгоритму вставки) в указанном порядке следующие ключи: 20, 15, 9, 18, 40, 35, 51, 27, 37, 36. Нарисовать АВЛ-деревья, которые получаются после добавления каждого из этих ключей.

4. Красно-черные деревья

АВЛ-деревья исторически были первым примером использования сбалансированных деревьев поиска. В настоящее время более популярны красно-черные деревья (КЧ-деревья). Изобретателем красно-черного дерева считается немецкий ученый Рудольф Байер. Название эта структура данных получила в статье Леонидаса Гимпаса и Роберта Седжвика 1978 года.

КЧ-деревья – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color, обозначающее цвет: красный или черный, и для которых выполнены приведенные ниже свойства.

C	Pascal

struct RBNode
{
key_type key;
struct RBNode *left;
struct RBNode *right;
struct RBNode *parent;
char color; // цвет
};
	type
	RBTree = ^RBNode;
	RBNode = record
	key: key_type;
	left: RBTree;
	right: RBTree;
	parent: RBTree;
	color: boolean;
	end;

Будем считать, что если left или right равны NULL, то это «указатели» на фиктивные листья. Таким образом, все узлы – внутренние (нелистовые).

Свойства КЧ-деревьев:

1. каждый узел либо красный, либо черный;
2. каждый лист (фиктивный) – черный;
3. если узел красный, то оба его сына – черные;
4. все пути, идущие от корня к любому фиктивному листу, содержат одинаковое количество черных узлов;
5. корень – черный.

Определение 4: *Черной высотой узла называется количество черных узлов на пути от этого узла к узлу, у которого оба сына – фиктивные листья.*

Сам узел не включается в это число. Например, у дерева, приведенного на Рис. 15, черная высота корня равна 2.

Определение 5: *Черная высота дерева – черная высота его корня.*

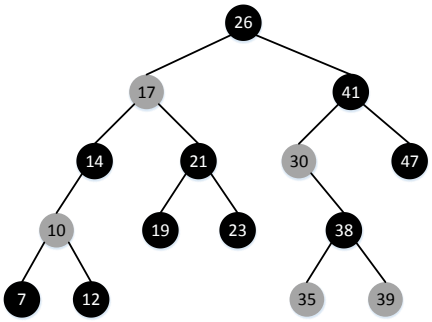


Рис. 15. Пример красно-черного дерева

4.1 Вставка узла в КЧ-дерево

Сначала узел добавляется в дерево с помощью стандартного алгоритма вставки узла в двоичное дерево поиска. Вновь добавленный узел красится в красный цвет. Если это первый узел в дереве, то он становится корнем и перекрашивается в черный цвет. Далее производится проверка, не нарушились ли свойства КЧ-дерева. Если добавленный узел не первый, то он красный, поэтому свойство 4 об одинаковом количестве черных узлов на любом пути от корня к листу, не нарушается. Если родитель нового узла черный, то свойство 3 о том, что если узел красный, то оба его сына черные, также не нарушается. Но если родитель нового узла красный, то это свойство будет нарушено – возникнет так называемое красно-красное нарушение. Тогда потребуется перекраска и, возможно, перестройка дерева. Обозначим добавленный узел за X, его отца за F, его деда за G, а второго сына деда – дядю – за U. Поддеревья Ti обозначены просто буквами, в отличие от иллюстраций в разделе

про АВЛ-деревья, потому что их высота в случае КЧ-деревьев не играет роли, а существенна только черная высота. Будем считать, что узел X находится в левом поддереве своего деда (G), иначе все приведенные ниже изображения будут симметричны относительно вертикальной оси, проходящей через деда. Рассматривается случай, когда узел X и его отец – красные, поэтому дед G узла X – черный, иначе красно-красное нарушение было бы еще до добавления узла X. Только дядя U узла X может иметь в данном случае как черный, так и красный цвет. При этом цепочка узлов X-F-G может образовывать как прямую линию, так и угол. Поэтому можно выделить 3 случая, которые проиллюстрированы на Рис. 16–18.

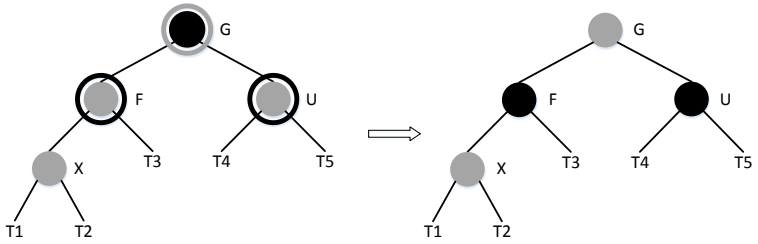


Рис. 16. Дядя U добавляемого узла X тоже красный: перекраска F и U в черный цвет, а G в красный

1. Дядя U добавляемого узла X красный.

В этом случае достаточно выполнить только перекраску: отца и дядю (F и U) – в черный цвет, деда (G) – в красный цвет. Тогда свойство, что у красного узла оба сына черные, будет выполнено. Свойство, что все пути, идущие от корня к листу, содержат одинаковое количество черных узлов, также не будет нарушено, потому что в каждом из путей два узла просто поменялись цветами (G и F слева, G и U справа). Количество черных узлов в путях при этом не изменилось. Если G – корень, то он просто перекрашивается в черный цвет. При этом все 5 свойств КЧ-деревьев оказываются выполненными. Если отец узла G тоже красный, то появится новое красно-красное нарушение, и понадобится дальнейшее восстановление свойств КЧ-дерева, только в роли узла X теперь будет выступать узел G. Может иметь место один из случаев 1–3.

2. Дядя U добавляемого узла X черный и при этом цепочка узлов X-F-G образует прямую линию.

Только перекраски отца F узла X в черный цвет, а деда G в красный будет недостаточно для восстановления свойств КЧ-дерева, потому что количество черных узлов в путях, проходящих через G направо, сократится на 1. Поэтому потребуется одинарный поворот деда (G) относительно отца, в данном случае направо. Тогда количество черных узлов в путях, идущих от F, который теперь стал корнем поддерева, налево и направо, вновь станет одинаковым и будет равно первоначальному количеству. Действительно, количество черных узлов в поддеревьях Ti не изменилось. При этом слева от корня в комбинации X-F-G-U не было черных узлов, а справа был только один черный узел U. В результирующем дереве, изображенном на Рис. 17, в комбинации X-F-G-U слева от корня также отсутствуют черные узлы, а справа – также только один черный узел.

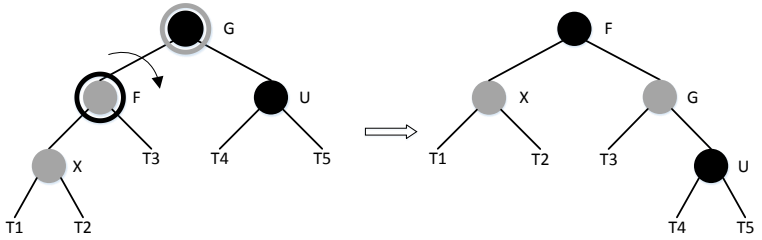


Рис. 17. Дядя U добавляемого узла X черный и X-F-G образуют прямую линию: перекраска F и G и одинарный поворот

3. Дядя U добавляемого узла X черный и при этом цепочка узлов X-F-G образует угол.

Перекрасив узел X в черный цвет, а его деда G – в красный, получим новое красно-красное нарушение F-G между отцом и дедом X. Ситуацию разрешает двойной поворот комбинации X-F-G, знакомый нам из раздела про АВЛ-деревья, когда нижний узел (X) оказывается наверху комбинации. Из Рис. 18 видно, что количество черных узлов, идущих от корня поддерева налево и направо, не

изменилось относительно первоначальной конфигурации, но при этом красно-красное нарушение ликвидировалось. Изменилось только количество красных узлов – слева уменьшилось на 1, а справа увеличилось на 1. А количество красных узлов в путях ни на что не влияет.

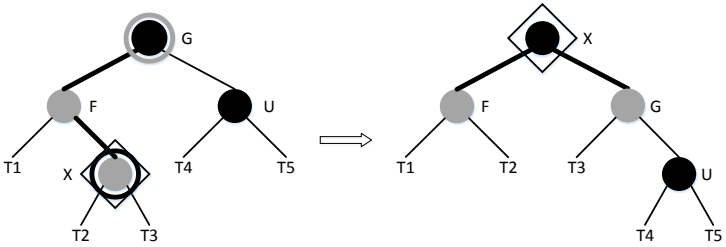


Рис. 18. Дядя U добавляемого узла X черный и X-F-G образуют угол: перекраска X и G и двойной поворот

Таким образом, получается, что после вставки в КЧ-дерево для восстановления свойств дерева требуется не более 2 поворотов. Действительно, повороты требуются только в случаях 2 и 3, а в случае 1 достаточно только перекраски. При этом случаи 2 и 3 являются терминальными, а рекурсивно продолжиться наверх может только процедура в случае 1, а она не требует поворотов.

Можно заметить, что повороты при балансировке как АВЛ-деревьев, так и КЧ-деревьев, делают одно и то же – поднимают поддеревья, содержащие более глубокие узлы, и опускают поддеревья, содержащие менее глубокие узлы. Различие заключается лишь в определении высоты. Для АВЛ-деревьев это высота в обычном понимании, а для КЧ-деревьев это черная высота.

RB-INSERT(T, x)

/ На вход подается дерево T и узел x, который надо добавить в дерево. */*

```
1  TREE-INSERT(T, x)
2  color[x] ← RED
3  while x ≠ root[T] и color[parent[x]] = RED do
5        if parent[x] = left[parent[parent[x]]] then
```

```
6                y ← right[parent[parent[x]]] /* y – дядя x */
7                if color[y] = RED then /* случай 1 */
8                        color[parent[x]] ← BLACK
9                        color[y] ← BLACK
10                       color[parent[parent[x]]] ← RED
11                       x ← parent[parent[x]] /* при следующем за-
ходе в цикл начнем проверку с деда x */
12               else
13                       if x = right[parent[x]] then /* случай 3 */
                         /* сводим к случаю 2 */
14                               x ← parent[x]
15                               TREE-ROTATE-L(T, x)
16                       color[parent[x]] ← BLACK /* случай 2 */
17                       color[parent[parent[x]]] ← RED
18                       TREE-ROTATE-R(T, parent[parent[x]])
19       else (аналогичный текст с заменой left ↔ right)
20   color[root[T]] ← BLACK
```

4.2 Удаление узла из КЧ-дерева

Удаление узла из КЧ-дерева, так же, как и удаление узла из АВЛ-дерева производится в два этапа: собственно удаление с помощью алгоритма из Раздела 2.3, и восстановление свойств дерева, если они были нарушены. Напомним, алгоритм удаления узла из двоичного дерева поиска зависит от того, сколько сыновей у удаляемого узла – 0, 1 или 2. Если у узла два сына, то удаляемым узлом становится его последователь – самый левый элемент правого поддерева. В этом случае у удаляемого узла уже будет максимум один сын (правый). Поэтому, в дальнейшем предполагается, что у удаляемого узла не более одного сына.

Если удаляемый узел является красным, то после его удаления свойства КЧ-дерева не будут нарушены. Свойство 4 сохранится: все пути будут по-прежнему содержать одинаковое количество черных узлов, так как при удалении красного узла оно не изменится. Свойство 3 – если узел красный, то оба его сына черные – также сохранится. Удаляемый узел красный, значит, как его отец, так

и его сын могут быть только черными. После удаления узла его сын присоединится к его отцу, а остальные связи останутся без изменения.

Таким образом, восстановление свойств дерева может понадобиться только в том случае, если удаляемый узел – черный. Если при этом сын удаляемого узла красный, то после удаления достаточно будет перекрасить сына удаленного узла в черный цвет, чтобы восстановить количество черных узлов на этом пути. Остальные свойства КЧ-деревьев не будут нарушены. Подобная ситуация может возникнуть и в том случае, если удаляемый узел является корнем. В этом случае дерево может состоять только из двух узлов – корня, который всегда черный, и его красного сына. Любые другие конфигурации в этом случае не отвечают свойствам КЧ-дерева. После удаления корня его сын станет единственным узлом в дереве.

Рассмотрим 5 случаев конфигурации дерева после удаления черного узла, у которого сын – черный. Обозначим сына удаленного узла за N, а отца удаленного узла за F. После удаления F стал новым отцом N. Обозначим за B нового брата N, а за CL и CR – левого и правого сыновей B, соответственно. На то, какой именно будет процедура восстановления, влияет только комбинация из этих 5 узлов. Будем считать, что N – левый сын F. В противном случае, все изображения будут симметричными относительно вертикальной оси, проходящей через F. Если узел обозначен белым цветом, это означает, что его цвет в данной комбинации может быть как черным, так и красным. По условию, во всех рассмотренных ниже случаях количество черных узлов в путях, идущих от F налево (через N) после удаления стало на один меньше, чем количество черных узлов в путях, идущих от F направо (через B), поэтому свойство 4 оказалось нарушено. Если хотя бы один из пяти узлов в рассматриваемой комбинации красный (случаи 1–4), то можно восстановить это свойство за O(1) операций перекраски или поворота. В первых трех случаях предполагается, что брат B узла N черный. При этом отец F и сыновья брата CL и CR могут иметь любые цвета. Наиболее простой случай, когда F красный, а CL и

CR – черные (случай 1). Тогда не требуется поворота, а понадобится только перекраска. В случае 2 предполагается, что у B красный правый сын CR. В этом случае потребуется и перекраска, и поворот. Случай 3, когда правый сын черный, а левый – красный, с помощью перекраски и поворота сводится к случаю 2. Когда брат B узла N красный, его сыновья CL и CR и отец F могут быть только черными, следовательно, этому условию соответствует только один случай (4), и он сводится к предыдущим трем случаям с помощью перекраски и поворота. В последнем случае, когда все узлы комбинации черные, можно будет только уровнять количество черных узлов в путях, идущих от F налево и направо, перекрасив брата B узла N в красный цвет, и продолжить процедуру восстановления свойства 4 КЧ-деревьев наверх.

1. Отец F узла N красный, остальные узлы черные. Эта конфигурация является наиболее простой для восстановления свойств КЧ-дерева. Достаточно просто перекрасить узлы F и B в противоположные цвета, после чего количество черных узлов в путях, идущих через F налево, увеличится на один. Количество черных узлов в путях, идущих через F направо при этом не изменится, как видно из Рис. 19. Поэтому, все пути будут содержать одинаковое количество черных узлов.

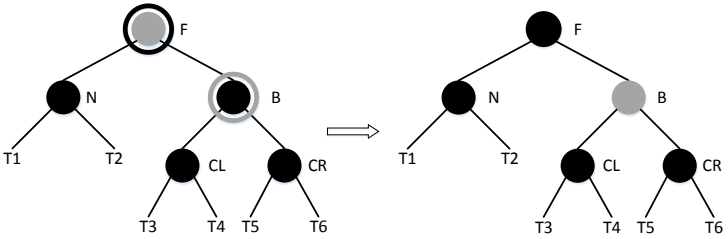


Рис. 19. Отец F узла N красный: перекраска B и F

2. Брат B узла N черный, а его правый сын CR красный. В этом случае, независимо от того, является ли F черным или красным, свойства КЧ-дерева восстанавливаются после поворота F вокруг B, перекраски CR в черный цвет и обмена цветами F и B. Действительно, если F был черным, то и B станет черным, поэтому

после поворота количество черных узлов слева увеличится на один, а справа не изменится, потому что CR перекрашивается в черный цвет. Если же F был красным, то количество черных узлов слева все равно увеличится на один, потому что после поворота F перекрашивается в черный цвет. А справа количество черных узлов не изменится, потому что B станет красным.

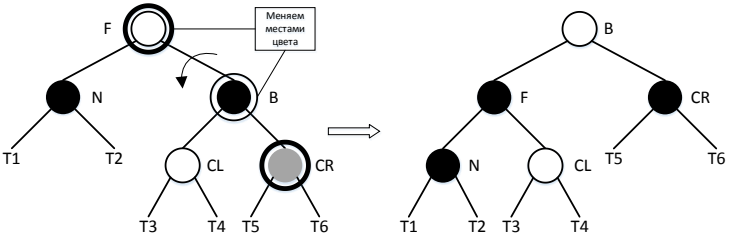


Рис. 20. Брат B узла N черный и его правый сын красный: поворот F вокруг B, перекраска CR и обмен цветами F и B

3. Брат B узла N черный, его правый сын CR черный, а левый сын CL красный. Этот случай сводится к предыдущему, если повернуть левого сына CL относительно своего отца F так, чтобы поддерево B со своими сыновьями вытянулось в одну линию, и перекрасить CL и B. Тогда у N будет черный брат с красным правым сыном.

4. Брат B узла N красный. В этом случае F, CL и CR могут быть только черными. За один шаг восстановить свойства КЧ-дерева при данной конфигурации невозможно. Но при помощи поворота F налево вокруг B и перекраски F и B в противоположные цвета можно свести этот случай к тем, когда брат узла N черный, то есть к первым трем случаям. Из Рис. 22 видно, что после поворота и перекраски количество черных узлов во всех путях не изменится. Действительно, слева добавился красный узел, а справа удалился.

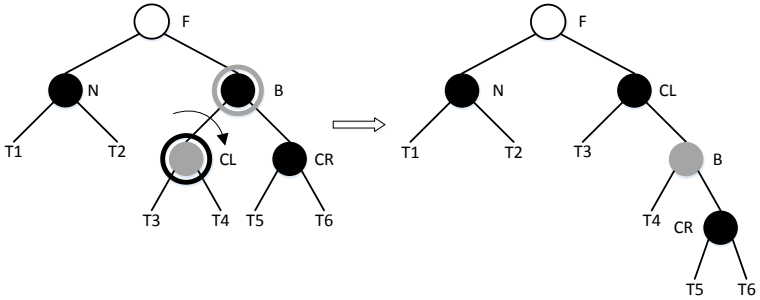


Рис. 21. Брат B узла N черный, его левый сын красный, а правый – черный: поворот CL вокруг B, перекраска CL и B

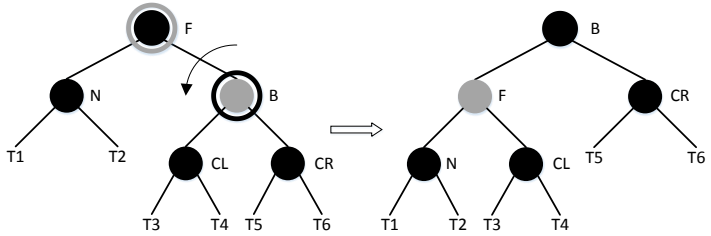


Рис. 22. Брат B узла N красный: поворот F вокруг B и перекраска F и B

5. Все узлы комбинации (брат B, его дети CL и CR, а также отец F узла N) черные. В этом случае для того, чтобы уравнять количество черных узлов в путях, идущих от P налево и направо, достаточно перекрасить B в красный цвет. Тогда справа станет на один черный узел меньше. Но при этом необходимо учесть, что у F могут быть предки, а количество черных узлов во всех путях, проходящих через F, сократилось, поэтому оно будет меньше, чем в путях, проходящих, например, через брата F, если таковой имеется. Следовательно, если F не является корнем всего дерева, нужно продолжить наверх процедуру восстановления свойств КЧ-дерева. Тогда роль узла N будет играть F. Происходит поиск соответствующего случая, начиная со случая 1.

Оценим количество поворотов, необходимых для восстановления свойств КЧ-дерева после удаления узла. Перекраска является менее трудоемкой операцией. В случае 1 поворота не требуется, в

случае 2 требуется один поворот. Учитывая, каким образом случаи сводятся друг к другу (4->1, 4->2, 4->3->2) и что каждый переход -> предполагает один поворот, получаем, что количество поворотов не превышает 3. В единственном случае (5), который приводит к возникновению цикла, производится только перекраска одного узла. На каком-либо из шагов при подъеме наверх процедура остановится и сведется к случаям 1–4. Количество шагов ограничено высотой дерева.

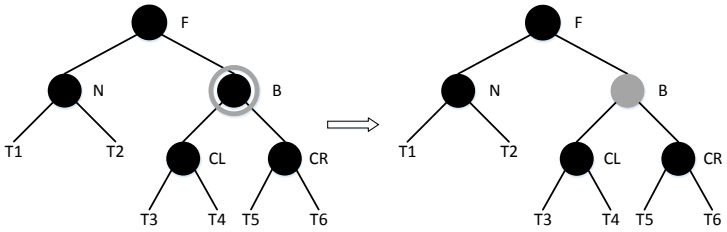


Рис. 23. Все узлы комбинации черные: перекраска B в красный цвет и продолжение процедуры наверх

Ниже приведена реализация операции удаления узла из КЧ-дерева с использованием вспомогательной процедуры восстановления свойств КЧ-дерева.

RB-DELETE-FIXUP(T, x)

```
/* На вход подается дерево T и n – сын удаленного узла. */
1 while n ≠ root[T] и color[n] = BLACK do
2       if n = left[parent[n]] then
3               b ← right[parent[n]] /* b – брат n */
4               if color[b] = RED then /* случай 4 */
5                       color[b] ← BLACK
6                       color[parent[n]] ← RED
7                       TREE-ROTATE-L(T, parent[n])
8                       b ← right[parent[n]] /* теперь у n черный брат */
9               if color[left[b]] = BLACK и color[right[b]] = BLACK then /* случай 1 или 5 */
```

```
10                  color[b] ← RED
11                  n ← parent[n] /* при следующем заходе в цикл просмотрим отца n: если он красный, то имел место случай 1 (в цикл не заходим), а если он черный, то имел место случай 5 (продолжаем цикл) */
12              else
13                  if color[right[b]] = BLACK then /* случай 3 */
                    /* сводим к случаю 2 */
14                      color[left[b]] ← BLACK
15                      color[b] ← RED
16                      RIGHT-ROTATE(T, b)
17                      b ← right[parent[n]]
18                  color[b] ← color[parent[n]] /* случай 2 */
19                  color[parent[n]] ← BLACK
20                  color[right[b]] ← BLACK
21                  LEFT-ROTATE(T, parent[n])
22                  n ← root[T] /* при попытке зайти в цикл следующий раз процесс прекратится */
23          else
24              (симметричный фрагмент с заменой left ↔ right)
25 color[n] ← BLACK
```

RB-DELETE(T, z)

```
/* На вход подается дерево T и узел z, который необходимо удалить из дерева, возвращается удаленный узел. */
1   if left[z] = null[T] или right[z] = null[T] then /* один из сыновей узла z – фиктивный лист */
2       y ← z
3   else
4       y ← TREE-SUCCESSOR(z)
5   if left[y] ≠ null[T] then
/* присваиваем x единственного сына y */
6       x ← left[y]
7   else
8       x ← right[y]
```

```
9   parent[x] ← parent[y]
10 if parent[y] = null[T] then
11        root[T] ← x
12 else
13        if y = left[parent[y]] then
14                  left[parent[y]] ← x
15        else
16                  right[parent[y]] ← x
17 if y ≠ z then
18        key[z] ← key[y]
19 if color[y] = BLACK then
/* если удаленный узел y – черный, то вызываем процедуру восстановления, которой передаем его сына */
20        RB-DELETE-FIXUP(T, x)
21 return y
```

4.3 Оценка сложности поиска в КЧ-дереве

Лемма 1. Красно-черное дерево с n внутренними узлами (без фиктивных листьев) имеет высоту не более $2log_2(n+1)$.

Доказательство.

1. Обозначим через $bh(x)$ черную высоту поддерева с корнем в узле x. Покажем вначале, что оно содержит не менее $2^{bh(x)} - 1$ внутренних узлов.

a. Индукция. Для листьев (не фиктивных) $bh(x) = 0$, то есть, $2^{bh(x)} - 1 = 2^0 - 1 = 0$.

b. Пусть теперь x – не лист и имеет черную высоту $bh(x) = k$. Тогда каждый сын x имеет черную высоту не менее $k - 1$. Действительно, если узел красный, его сыновья могут быть только черными, и в этом случае черная высота сына x будет на 1 меньше, чем черная высота x, т.е. будет равна $k - 1$, потому как при расчете черной высоты узла сам узел в это число не включается. Если узел черный, то его сыновья

могут быть как черными, так и красными. Если сын черный, то аналогично предыдущему случаю его черная высота на 1 меньше, чем у x и равна $k - 1$. Если сын красный, то он имеет черную высоту такую же, как и x, т.е. k, т.к. количество черных узлов от x до листа, не включая x, такое же, как и количество черных узлов от его красного сына до листа.

c. По предположению индукции каждый сын имеет черную высоту $bh \geq k - 1$, а, следовательно, не менее $2^{k-1} - 1$ узлов. Поэтому поддерево с корнем x имеет не менее $2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1$ узлов.

2. Теперь пусть высота дерева равна h.

a. По свойству красно-черных деревьев, что если узел красный, то оба его сына черные, черные узлы составляют не меньше половины всех узлов на пути от корня к листу. Поэтому, черная высота дерева bh не меньше $h/2$.

b. Тогда $n \geq 2^{h/2} - 1$, откуда

$$h \leq 2log_2(n + 1). \qquad (12)$$

Лемма доказана.

Из Леммы 1 следует, что поиск по КЧ-дереву имеет сложность $O(log_2 n)$.

4.4 Задачи

1. В каком случае при добавлении нового узла происходит увеличение черной высоты КЧ-дерева?

2. Как и где используется свойство, что корень КЧ-дерева черный?

3. Могут ли все узлы КЧ-дерева из пяти внутренних узлов быть черными? А из 7 внутренних узлов?

4. Может ли КЧ-дерево из 3 узлов, где все узлы черные, быть построено путем добавления в пустое дерево узлов по одному? А если еще и удалять можно?

5. Переформулируйте определение КЧ-дерева так, чтобы для обеспечения сбалансированности не требовалось вводить фиктивные листовые узлы.

6. Чему равно максимальное и минимальное число красных узлов в КЧ-дереве высоты h?

7. Чему равно минимальное количество узлов в КЧ-дереве высоты h?

8. Назовем узел полулистовым, если у него не более одного потомка. (Рассматриваются только реальные потомки, а не фиктивные листья.) Относительной разбалансировкой назовем отношение наибольшей длины пути от корня до полулистового узла в дереве к наименьшей. Чему равна максимальная возможная относительная разбалансировка КЧ-дерева? Чему равна максимальная возможная относительная разбалансировка АВЛ-дерева?

5. Самоперестраивающиеся деревья (splay trees)

Самоперестраивающееся дерево – это двоичное дерево поиска, которое, в отличие от предыдущих двух видов деревьев не содержит дополнительных служебных полей в структуре данных (баланс, цвет и т.п.). Оно позволяет находить быстрее те данные, которые использовались недавно. Самоперестраивающееся дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году.

Идея самоперестраивающихся деревьев основана на принципе перемещения найденного узла в корень дерева. Эта операция называется splay(T, k), где k – это ключ, а T – двоичное дерево поиска. После выполнения операции splay(T, k) двоичное дерево T перестраивается, оставаясь при этом деревом поиска, так, что:

- если узел с ключом *k* есть в дереве, то он становится корнем;
- если узла с ключом *k* нет в дереве, то корнем становится его предшественник или последователь.

Таким образом, поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay. Эвристика move-to-front (перемещение найденного узла в корень) основана на предположении, что если тот же самый элемент потребуется в ближайшее время, он будет найден быстрее.

Определение 6: *Словарными операциями над деревом называются базовые операции: поиск, вставка и удаление.*

Рассмотрим реализацию других словарных операций через splay.

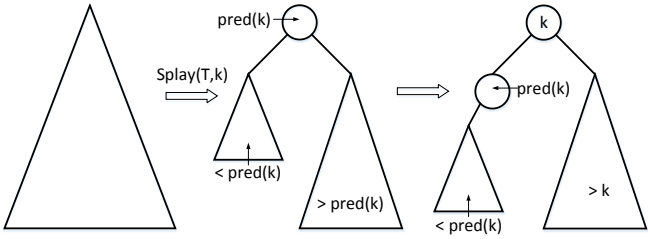
5.1 Вставка узла в самоперестраивающееся дерево

Алгоритм вставки узла в самоперестраивающееся дерево начинает свою работу с поиска узла в этом дереве с помощью описанной выше операции splay(*T, k*). Далее, проверяется значение ключа в корне. Если оно равно *k*, значит, узел найден. Если же оно не равно *k*, значит, в корне находится его предшественник или последователь. Тогда *k* становится новым корнем и, в зависимости от того, было ли до этого в корне значение, большее *k* или меньшее, старый корень становится правым или, соответственно, левым сыном корня. Пример показан на Рис. 24.

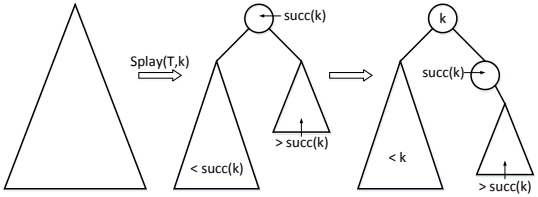
```
SPLAY-INSERT(T, x)

/* На вход подается дерево T и узел x, который надо добавить в дерево. */
1 if root[T] = NULL then
2        root[T] ← x
3        return
4 SPLAY(T, key[x])
5 if key[root[T]] < key[x] then /* в корне находится предшественник x */
```

```
6       left[x] ← root[T]
7       right[x] ← right[root[T]]
8       if right[x] ≠ NULL then
9               parent[right[x]] ← x
10 else /* в корне находится последователь x */
11      right[x] ← root[T]
12      left[x] ← left[root[T]]
13      if left[x] ≠ NULL then
14              parent[left[x]] ← x
15 parent[root[T]] ← x
16 root[T] ← x
```



(а)



(б)

Рис. 24. Вставка узла с ключом k через операцию splay:
(а) в корне находится предшественник узла с ключом k (pred(k));
(б) в корне находится последователь узла с ключом k (succ(k))

5.2 Удаление узла из самоперестраивающегося дерева

Перед тем, как удалить узел с ключом *k* из самоперестраивающегося дерева *T*, необходимо выполнить операцию splay(*T, k*) и проверить значение в корне. Если оно не равно *k*, то узла с таким ключом в дереве нет, поэтому удалять нечего. Если оно равно *k*, то корень удаляется, а его левое и правое поддеревья склеиваются с помощью операции слияния concat(*T->left, T->right*). Пример приведен на Рис. 25.

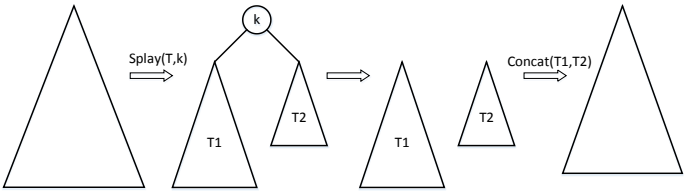


Рис. 25. Удаление узла с ключом k через операции splay и concat

Операция concat (T_1, T_2) – слияние двух деревьев поиска T_1 и T_2, таких, что **все** ключи в дереве T_1 **меньше**, чем любой ключ в дереве T_2, в одно дерево поиска, производится следующим образом. Сначала выполняется операция splay(max(T_1), T_1), в результате которой максимальный ключ в дереве T_1 становится его корнем. После этого у корня T_1 не будет правого сына. Затем дерево T_2 присоединяется в качестве правого сына к корню T_1. Нетрудно проверить, что получившееся дерево сохранит свойства дерева поиска. Действительно, у корня T_1 слева все ключи меньше него, т.к. в корень был помещен узел с максимальным ключом. А справа все ключи больше него, т.к. по условию все ключи в дереве T_2 больше всех ключей в дереве T_1. Схема выполнения операции слияния приведена на Рис. 26.

```
CONCAT(T1, T2)

/* На вход подаются два дерева – T1 и T2, возвращается объединенное дерево. */
1  if root[T1] = NULL then
2        return T2
3  if root[T2] = NULL then
```

```
4       return T1
5 y ← TREE-MAXIMUM(T1)
6 SPLAY(T1, key[y])
7 right[root[T1]] ← root[T2]
8 parent[root[T2]] ← root[T1]
9 return T1
```

SPLAY-DELETE(T, k)

/ На вход подается дерево T и значение ключа k, узел с которым надо удалить из дерева. Возвращается удаленный узел или NULL, если узла с таким ключом в дереве не оказалось. */*

```
1 SPLAY(T, k)
2 if root[T] = NULL или key[root[T]] ≠ k then /* дерево пусто или удаляемого узла в дереве нет */
3       return NULL
4 x ← root[T]
5 root[T] ← CONCAT(left[root[T]], right[root[T]])
6 return x
```

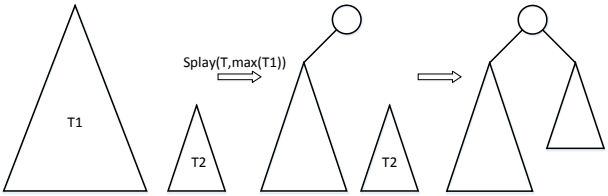


Рис. 26. Слияние деревьев T_1 и T_2 через операцию splay

5.3 Выполнение операции splay(T, k)

Теперь рассмотрим, каким образом выполняется сама операция splay(*T*,*k*). Сначала производится поиск узла с ключом *k* в дереве обычным способом, спускаясь вниз, начиная с корня. При этом запоминается пройденный путь. В итоге, получаем указатель на узел дерева либо с ключом *k*, либо с его предшественником или последователем, на котором закончился поиск. Далее, происходит

возвращение назад по запомненному пути, с перемещением этого узла к корню. Для того, чтобы при этом сохранялись свойства двоичного дерева поиска, необходимы повороты.

Если узел с ключом *k* является сыном корня, как, например, в случае на Рис. 27, то потребуется одинарный поворот этого узла относительно корня таким образом, чтобы он сам стал корнем. В указанном примере понадобится поворот направо. Но можно в качестве примера привести симметричный случай, который потребует аналогичного поворота налево.

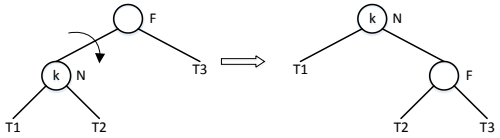


Рис. 27. Поднимаем узел с ключом k наверх – правый поворот

Если у узла с ключом *k* есть дед, и при этом сам узел и его родитель являются одинаковыми потомками своих родителей (оба левыми или оба правыми), т.е. цепочка «узел – родитель – дед» образует прямую линию, как на Рис. 28, то понадобится последовательность из двух одинарных поворотов, которая приведет к тому, что узел с ключом *k* окажется наверху – сначала поворот деда вокруг отца, потом отца вокруг самого узла. В примере на Рис. 28 повороты будут правые.

Также можно привести симметричный случай, где понадобятся два последовательных поворота налево. Если последовательность из двух поворотов еще не приведет к тому, что узел с ключом *k* станет корнем, потребуются новые повороты. В зависимости от расположения новых родителя и деда узла с ключом *k* могут потребоваться повороты в другую сторону или двойной поворот, описанный ниже. И так, пока случай не сведется к тому, который показан в примере на Рис. 27, где требуется одинарный поворот для того, чтобы k оказался в корне.

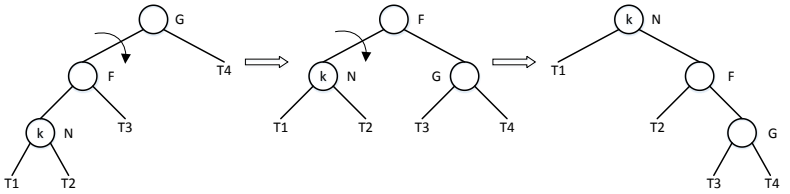


Рис. 28. N-F-G образуют прямую линию: два правых поворота для перемещения узла с ключом *k* в корень

Двойной поворот представляет собой уже известную из предыдущих разделов перестройку треугольника «узел – родитель – дед» таким образом, чтобы нижний узел стал верхним, а родитель и дед стали его новыми сыновьями. Применяется в тех случаях, когда узел, его родитель и дед образуют не прямую линию, а угол, как в примере на Рис. 29.

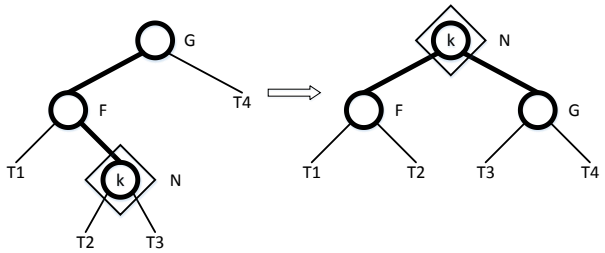


Рис. 29. N-F-G образуют угол: двойной поворот для перемещения узла с ключом *k* в корень

```
SPLAY(T, k)

/* На вход подается дерево T и значение ключа k */
1   n ← TREE-SEARCH-INEXACT(T, k) /* в n будет либо узел с ключом k, либо узел, на котором остановился поиск (его последователь или предшественник) */
2   while n ≠ NULL и parent[n] ≠ NULL do
3       f ← parent[n]
4       g ← parent[f]
5       if g = NULL then
6           if n = left[f] then
```

```
7                       TREE-ROTATE-R(T, f)
8               else
9                       TREE-ROTATE-L(T, f)
10      else
11              if f = left[g] then
12                      if n = left[f] then
13                              TREE-ROTATE-R(T, g)
14                              TREE-ROTATE-R(T, f)
15                      else
16                              TREE-ROTATE-LR(T, g)
17              else
18                      if n = right[f] then
19                              TREE-ROTATE-L(T, g)
20                              TREE-ROTATE-R(T, f)
21                      else
22                              TREE-ROTATE-RL(T, g)
```

5.4 Оценка сложности операций над самоперестраивающимся деревом

Во-первых, заметим, что все вышеперечисленные базовые операции (поиск, вставка, удаление) требуют выполнения O(1) операций splay и O(1) дополнительного времени. Действительно, при поиске и вставке требуется одна операция splay, а при удалении – две. Дополнительные действия – просмотр корня, удаление вершины и т.п. занимают фиксированное время, которое не зависит от высоты дерева. Сама операция splay занимает O(n) времени, где n – количество узлов в дереве. Это происходит тогда, когда дерево совсем не сбалансировано. Но, в целом, эти деревья за счет того, что они самоперестраивающиеся, имеют тенденцию к сбалансированности.

Введем следующие понятия.

Определение 7: *Весом узла x называется количество узлов в поддереве T с корнем в x, включая сам узел: |T(x)|.*

Определение 8: *Рангом узла называется двоичный логарифм его веса:* $r(x) = \log_2 |T(x)|$.

Проведем усредненную оценку сложности операций с помощью так называемого *метода бухгалтерского учета*. Представим, что каждая операция с деревом стоит фиксированную сумму денег за единицу времени. В самом начале каждый узел содержит кредит – $r(x)$ рублей, т.е. сумму, равную его рангу, которая может частично или полностью использоваться для оплаты операций. Также можно инвестировать дополнительную сумму для оплаты операций, помимо кредита. Если после серии операций над деревом и перед началом новых операций суммарный кредит (сумма рангов всех вершин) будет не меньше, чем до них, то будем говорить о *сохранении денежного инварианта*. Тогда можно проводить эту серию операций любое количество раз. Докажем следующую лемму.

Лемма 2: Операция splay(T, x) требует инвестирования не более чем $3\log_2 n + 1$ рублей с сохранением денежного инварианта.

Доказательство:

Сложность будет оцениваться по количеству поворотов. Обозначим через $r(x)$ и $r'(x)$ значения ранга узла x до и после поворота (1-го типа – одинарного, 2-го типа – серии из двух одинарных или 3-го типа – двойного).

Рассчитаем, какую сумму потребуется инвестировать, чтобы ее хватило и на сохранение денежного инварианта, и непосредственно на операции.

Ниже будет показано, что для выполнения поворота любого типа и сохранения денежного инварианта требуется не более $3(r'(x) - r(x)) + 1$ рублей, причем 1 добавляется только при одинарном повороте. При выполнении операции splay(T, x) производится последовательность из k поворотов 2-го и 3-го типов и поворота 1-го типа на заключительном этапе. После каждого поворота ранг узла x будет меняться. Пусть изначально ранг узла x состав-

ляет $r_0(x)$, а после i-го поворота $r_i(x)$. Тогда для выполнения последовательности из k поворотов потребуется сумма

$$1+\sum_{i=1}^{k}3(r_i(x)-r_{i-1}(x))=1+3(r_k(x)-r_0(x)) \qquad (13)$$

Учитывая, что $r_k(x) = r(T)$, поскольку в конце последовательности поворотов узел оказывается в корне всего дерева, а $r_0(x) = r(x)$, получаем общее количество рублей, необходимое для выполнения операции splay(x, T) и сохранения денежного инварианта, равное $3(r(T)-r(x))+1$ рублей. Учитывая, что минимальное значение $r(x)$ равно 0, а ранг дерева является логарифмом от количества его узлов, получаем верхнюю оценку: $3\log_2 n+1$ рублей.

Теперь покажем, что для выполнения поворота и сохранения денежного инварианта требуется не более $3(r'(x)-r(x))+1$ рублей. Рассмотрим каждый тип поворота по отдельности.

1. Одинарный поворот

Обозначим корень за y, а его сына – искомый узел – за x. Для сохранения денежного инварианта потребуется сумма

$$\delta = r'(x)+r'(y)-r(x)-r(y).$$

После поворота x и y меняются ролями – x становится отцом, а y – сыном. Поэтому $r'(x)=r(y)$ и $r'(y)\le r'(x)$. В остальных узлах значения r не меняются. Следовательно,

$$\delta = r'(y)-r(x)\le r'(x)-r(x).$$

Для выполнения, собственно, вращения потребуется 1 операция, и, следовательно, 1 рубль. Поэтому общее количество рублей N, необходимое для выполнения поворота и сохранения денежного инварианта, удовлетворяет неравенству:

$$N\le r'(x)-r(x)+1\le 3(r'(x)-r(x))+1. \qquad (14)$$

2. Два одинарных поворота

Обозначим искомый узел за *x*, его отца за *y*, а деда за *z*. Для сохранения денежного инварианта потребуется сумма

$$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

После выполнения двух одинарных поворотов узел *x* становится «старшим» в этой комбинации, его отец *y* становится его сыном, а его дед *z* – его внуком. Поэтому $r'(x) = r(z)$, $r'(y) \le r'(x)$ и $r(y) \ge r(x)$. В остальных узлах значения *r* не меняются. Следовательно, получим:

$$\delta = r'(y) + r'(z) - r(x) - r(y) \le r'(x) + r'(z) - 2r(x).$$

Непосредственно два поворота составляют 2 операции, следовательно, требуют 2 рубля. Поэтому, для выполнения вращения и сохранения денежного инварианта потребуется сумма

$$N \le r'(x) + r'(z) - 2r(x) + 2.$$

Докажем, что

$$r'(x) + r'(z) - 2r(x) + 2 \le 3(r'(x) - r(x)). \qquad (15)$$

Перепишем неравенство (15) в виде

$$r'(z) + r(x) - 2r'(x) \le -2.$$

По определению ранга

$$r'(z) + r(x) - 2r'(x) = \log_2 |T'(z)| + \log_2 |T(x)| - 2\log_2 |T'(x)|, \qquad (16)$$

где $T(x)$ ($T(z)$) – дерево с корнем в узле *x* (*z*) до поворота, а $T'(x)$ ($T'(z)$) – дерево с корнем в узле *x* (*z*) после поворота.

Преобразуем правую часть (16):

$$\log_2 |T'(z)| + \log_2 |T(x)| - 2\log_2 |T'(x)| =$$
$$= (\log_2 |T'(z)| - \log_2 |T'(x)|) + (\log_2 |T(x)| - \log_2 |T'(x)|) =$$
$$= \log_2 \frac{|T'(z)|}{|T'(x)|} + \log_2 \frac{|T(x)|}{|T'(x)|}.$$

Зная, как будет перестраиваться дерево при повороте, можно заметить, что $|T'(x)| \geq |T'(z)| + |T(x)|$.

Следовательно,

$$\frac{|T'(z)|}{|T'(x)|} + \frac{|T(x)|}{|T'(x)|} = \frac{|T'(z)| + |T(x)|}{|T'(x)|} \leq 1.$$

Выпуклость функции логарифма предполагает, что

$$\log_2 x + \log_2 y = \log_2 xy \text{ для } x, y > 0, x + y \leq 1$$

принимает максимальное значение –2 при

$$x = y = \frac{1}{2}.$$

Поэтому

$$\log_2 \frac{|T'(z)|}{|T'(x)|} + \log_2 \frac{|T(x)|}{|T'(x)|} \leq -2,$$

следовательно,

$$\log_2 |T'(z)| + \log_2 |T(x)| - 2\log_2 |T'(x)| \leq -2.$$

Таким образом, мы получили, что для выполнения двух одинарных поворотов и сохранения денежного инварианта требуется сумма

$$N \leq 3(r'(x) - r(x)). \quad (17)$$

Двойной поворот

После выполнения двойного поворота узел x становится «старшим», а его отец y и дед z становятся его сыновьями. Так же, как и в предыдущем случае, имеют место соотношения $r'(x) = r(z)$

и $r(y) \geq r(x)$. Следовательно, для сохранения денежного инварианта потребутся сумма

$$\delta = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \leq r'(y) + r'(z) - 2r(x).$$

Для выполнения двойного поворота и сохранения денежного инварианта потребуется сумма $N = \delta + 2$, поэтому

$$N \leq r'(y) + r'(z) - 2r(x) + 2.$$

Докажем, что

$$r'(y) + r'(z) - 2r(x) + 2 \leq 2(r'(x) - r(x)). \quad (18)$$

Перепишем неравенство (18) в виде

$$r'(y) + r'(z) - 2r'(x) \leq -2.$$

Это неравенство доказывается аналогично предыдущему случаю, только, учитывая, что y и z стали сыновьями x, используем неравенство

$$|T'(x)| \geq |T'(y)| + |T'(z)|.$$

Получаем, что для выполнения двойного поворота и сохранения денежного инварианта требуется сумма

$$N \leq 2(r'(x) - r(x)) \leq 3(r'(x) - r(x)). \quad (19)$$

Учитывая неравенства (14), (17) и (19), получаем, что для выполнения любого из трех типов поворотов и сохранения денежного инварианта требуется не более

$$3(r'(x) - r(x)) + 1 \text{ рублей.}$$

Лемма доказана.

Теорема 4: Любая последовательность из m словарных операций на самоперестраивающемся дереве, которое было изначально пусто и на каждом шаге содержало не более n узлов, занимает не более $O(m \log n)$ времени.

Доказательство:

Из Леммы 2 следует, что операция splay(*T, x*) требует инвестирования не более $3\log_2 n+1$ рублей. Вспоминая, что каждая из словарных операций требует O(1) операций splay и O(1) дополнительного времени, получаем, что для выполнения последовательности из *m* операций дополнительно требуется не более O($m(3\log_2 n+1)$) инвестиций (при удалении операция splay производится два раза) и при этом можно использовать деньги узла. Сначала дерево содержит 0 рублей, в конце ≥ 0 рублей. Следовательно, O(m log n) рублей хватает на все операции. Теорема доказана.

5.5 Задачи

1. Дано самоперестраивающееся дерево, такое, что путь от корня к узлу с ключом 90 проходит через следующие узлы в порядке: 10, 20, 30, 40, 50, 60, 70, 80, 90. Нарисовать результат операции splay над узлом 90.

2. Дано самоперестраивающееся дерево, такое, что путь от корня к узлу с ключом 90 проходит через следующие узлы в порядке: 50, 130, 60, 120, 70, 110, 80, 100, 90. Нарисовать результат операции splay над узлом 90.

3. Пусть до выполнения операции splay все узлы дерева из задачи 1 на пути к узлу 90 имеют ранг k. Показать, что после выполнения операции splay над узлом 90 ранги этих узлов не увеличатся, а ранги как минимум трех из них уменьшатся.

4. Пусть до выполнения операции splay все узлы дерева из задачи 2 на пути к узлу 90 имеют ранг k. Показать, что после выполнения операции splay над узлом 90 ранги этих узлов не увеличатся, а ранги как минимум четырех из них уменьшатся.

6. Сравнение АВЛ-деревьев, КЧ-деревьев и самоперестраивающихся деревьев

В общем случае сравнение трех рассмотренных типов деревьев провести затруднительно, поскольку для разных задач и разных наборов данных лучший тип дерева может быть разным. Сравнивать деревья можно по разным критериям: по сложности реализации, в теории, на практике.

По сложности реализации самые простые – АВЛ-дерево, самые сложные – КЧ-деревья, поскольку приходится рассматривать много нетривиальных случаев при вставке и удалении узла.

В теории оценки сверху для всех трех типов деревьев примерно одинаковые. Восстановление свойств как АВЛ-дерева, так и КЧ-дерева после вставки требует не более двух поворотов. Но после удаления узла из КЧ-дерева потребуется не более трех поворотов, а в АВЛ-дереве после удаления узла может потребоваться количество поворотов до высоты дерева (от листа до корня). Поэтому операция удаления в КЧ-дереве эффективнее, в связи с чем они больше распространены.

Самоперестраивающиеся деревья существенно отличаются от АВЛ- и КЧ-деревьев потому как не накладываются никакие ограничения на структуру дерева. Операция поиска в дереве модифицирует само дерево, поэтому в случае обращения к **разным** узлам самоперестраивающееся дерево может работать медленнее. К тому же, в процессе работы дерево может оказаться полностью разбалансированным. Но доказано, что если вероятности обращения к узлам фиксированы, то самоперестраивающееся дерево будет работать асимптотически не медленнее двух других рассмотренных видов деревьев [6]. Отсутствие дополнительных полей дает преимущество по памяти.

Различные виды сбалансированных деревьев поиска используются, в частности, в системном программном обеспечении, например, в ядрах операционных систем. В статье [9] приведены результаты

тестов, имитирующих некоторую реальную нагрузку на деревья поиска. Учитывая, что таблицы виртуальных адресов в Linux часто делаются на двоичных деревьях поиска, авторы инструментировали несколько приложений, чтобы получить последовательность их обращений к подсистемам виртуальной памяти, а затем использовали эти последовательности для эмуляции нагрузки на двоичные деревья в ядре операционной системы. Так, например, показано, что если при использовании браузером Mozilla виртуальной памяти менеджер виртуальной памяти будет использовать самоперестраивающиеся деревья, то преимущество этого вида деревьев по времени работы над АВЛ- и КЧ-деревьями будет минимум в 2, а максимум в 3.4 раза.

В статье также показано, в каком случае какой вид деревьев лучше всего использовать. Если входные данные полностью рандомизированы, то наилучшим вариантом оказываются деревья поиска общего вида – несбалансированные. Если входные данные в основном рандомизированные, но периодически встречаются упорядоченные наборы, то стоит выбрать КЧ-деревья. Если при вставке преобладают упорядоченные данные, то АВЛ-деревья оказываются лучше в случае, когда дальнейший доступ к элементам рандомизирован, а самоперестраивающиеся деревья – когда дальнейшие доступ последователен или кластеризован.

7. Литература

1. А.А. Белеванцев. Лекции по курсу «Алгоритмы и алгоритмические языки». 2013. http://algcourse.cs.msu.su/?page_id=30

2. Вирт Н. Алгоритмы и структуры данных М.: Мир, 1989. Глава 4.5 (С. 272–286).

3. Г. М. Адельсон-Вельский, Е. М. Ландис. Один алгоритм организации информации // Доклады АН СССР. 1962. Т. 146, № 2. С. 263–266.

4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ = Introduction to algorithms. — 2-е изд. — М.: Издательский дом «Вильямс», 2011. — С. 336–364.

5. Д. Кнут. Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. — С. 720.

6. R. E. Tarjan. Data Structures and Networks Algorithms. SIAM. 1983.

7. D. D. Sleator, R.E. Tarjan. Self-Adjusting Binary Search Trees // Journal of the ACM (JACM) JACM, vol. 32(3), pp. 652–686. 1985.

8. H.R. Lewis, L. Denenberg. Data Structures and Their Algorithms. Addison-Wesley, 1991.

9. P. Bfaff. Performance Analysis of BSTs in System Software // SIGMETRICS Perform. Eval. vol. 32(1), pp. 410–411. 2004.

Учебно-методическое издание

СЕНЮКОВА Ольга Викторовна

СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ ПОИСКА

Учебно-методическое пособие

Издательский отдел
Факультета вычислительной математики и кибернетики МГУ
имени М.В. Ломоносова
Лицензия ИД N 05899 от 24.09.01 г.

119992, ГСП-2, Москва, Ленинские горы,
МГУ имени М.В. Ломоносова,
2-й учебный корпус

Напечатано с готового оригинал-макета
в издательстве ООО “МАКС Пресс”.
Лицензия ИД N 00510 от 01.12.99 г.

Подписано в печать 25.11.2014 г.
Формат 60x90 1/16. Усл.печ.л. 4,25. Тираж 100 экз. Заказ 272.

119992, ГСП-2, Москва, Ленинские горы, МГУ им. М.В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891