ЛЕКЦИЯ 9

ДЕРЕВЬЯ. ДВОИЧНОЕ ДЕРЕВО ПОИСКА. СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

1. Деревья как структуры данных

Сегодняшняя лекция посвящена структурам данных.

Из курса теории графов известно определение дерева как связного ациклического графа. Граф является **связным**, если между любыми двумя его вершинами существует как минимум один путь. Граф является **ациклическим**, если не он не содержит замкнутых последовательностей рёбер, где все рёбра различны. Деревья принято изображать так, как показано на рисунке 9.1. В дереве выделяются слои, и на верхнем из них находится только один узел. Верхняя вершина дерева называется его **корнем**. Глубина дерева может отсчитывать либо с нуля, либо с единицы, в разных задачах делается по-разному. Рёбра соединяют только узлы, чьи номера слоёв отличаются на единицу. Если узлы соединены линией, то тот, что выше из них, называется **родителем**, или **родительским узлом**, а тот, что ниже, называется **потомком**. Узлы без потомков называются **листьями**.

Дерево называется **бинарным**, если у одного родителя могут быть не более двух

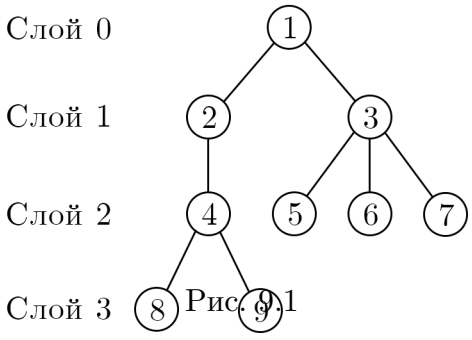

Рис. 9.1

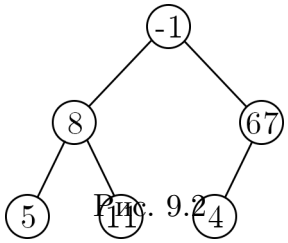

Рис. 9.2

детей. В узлах структуры данных «дерево» могут храниться какие-то данные, например, целые числа. Бинарное дерево может быть **упорядоченным** или **неупорядоченным**. Если дерево неупорядочено, то чтобы найти узел с нужным числом, нужно осуществить поиск по всему дереву. Если в дереве n узлов, то сложность такого поиска — $O(n)$, что не лучше, чем поиск по неупорядоченному массиву или списку.

2. Двоичное дерево поиска

Пусть в некоторой задаче нужно хранить однотипные данные, причём в структуру данных достаточно редко нужно добавлять новые элементы, зато нужно часто проверять, если ли в этой структуре определённый элемент. Если множество элементов неупорядочено, то для ответа на вопрос, есть ли элемент в этом множестве, потребуется $O(n)$ операций.

Пусть структура данных — это упорядоченный массив. Возможен «тупой» вариант: перебирать элементы массива один за другим; как только искомый элемент найден, или пошли элементы большего значения, чем искомый элемент, то выполнение можно прервать. В худшем и среднем случае такой способ поиска всё равно даёт сложность $O(n)$. Можно воспользоваться методом деления пополам, или двоичным поиском, который даёт $O(\log n)$ в худшем и среднем случае. Есть стандартная функция `bisect`, которая реализует этот алгоритм. Кроме того, чтобы вставить элемент внутрь динамического массива, тоже понадобится $O(n)$ операций: придётся сдвигать элементы справа от места вставки на одну ячейку.

Если структура данных — список, то вставка элемента после определённого элемента производится быстро и имеет сложность $O(1)$. Зато обращение к элементу по его порядковому номеру имеет сложность $O(n)$.

Таким образом, в список легко вставлять, но тяжело брать `i`-тый элемент, а в массиве легко взять `i`-тый элемент, но тяжело в него вставлять. Желательно, чтобы обе эти операции выполнялись быстрее, чем за $O(n)$ операций, поскольку и то, и другое в данной задаче нужно делать часто. Нужна такая упорядоченная структура данных, для которой можно быстро определить, есть ли в ней определённый элемент, и быстро вставлять элементы.

Такую структуру можно реализовать на основе двоичного дерева, и она называется **бинарное дерево поиска**. Название «дерево поиска» намекает, что поиск в нём будет

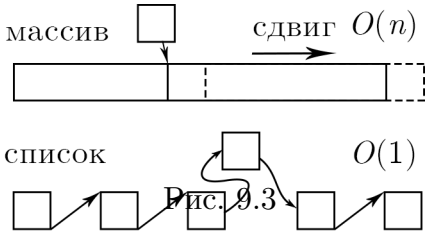

Рис. 9.3

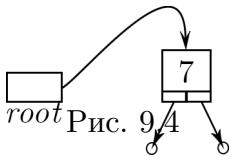

Рис. 9.4

производиться быстро. У каждого родителя не более двух детей; у каждого узла, кроме корня, ровно один предок.

Займёмся реализацией двоичного дерева. Две часто используемые операции над деревом — это добавление элемента в дерево и распечатка дерева. Потом можно будет реализовать поиск по дереву. В двусвязном списке были указатели на ячейки справа и слева, а здесь в структуре должны быть указатели на потомков. Каждый узел хранит данные типа `Data` и два указателя на другие узлы:

Listing 9.1: Структура, определяющая узел двоичного дерева.

```
struct Node {
        Data val;
        struct Node * left;
        struct Node * right;
};
```

Если какого-то потомка нет, то соответствующий указатель ссылается на `null`. При добавлении элемента в дерево нужно создать структуру типа `Node` с пустыми указателями, соответствующую этому элементу, и изменить какую-то `null`-ссылку в дереве на адрес новой структуры.

Организуем процедуру вставки так, чтобы дерево всегда было упорядоченным. Будем считать, что все вставляемые элементы уникальны. Упорядочивать будем так, чтобы меньшие элементы будут находиться слева от больших. Пусть, для примера, числа, которые будут последовательно вставляться в дерево, таковы:

$$7, 3, 2, 1, 9, 5, 3, 4, 8. \tag{9.1}$$

Создадим переменную `root` типа `Node`, в которой будет храниться указатель на корневой узел. Добавляем число 7.

1. Создаём структуру типа `Node` с переменной `val=7` и обеими ссылками на потомков, равными `null`.
2. Присваиваем переменной `root` адрес этой структуры (рис. 9.4).

Добавляем следующий элемент — число 3.

1. Создаём структуру типа `Node` с переменной `val=3` и обеими ссылками на потомков, равными `null`.
2. Так как $3 < 7$, то будем добавлять этот элемент слева от корня. Присваиваем левому указателю узла с числом 7 адрес нового элемента.

Добавляем следующий элемент — число 2.

1. Создаём структуру типа `Node` с переменной `val = 2` и обеими ссылками на потомков, равными `null`.

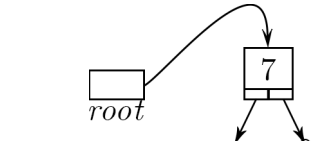

Рис. 9.5

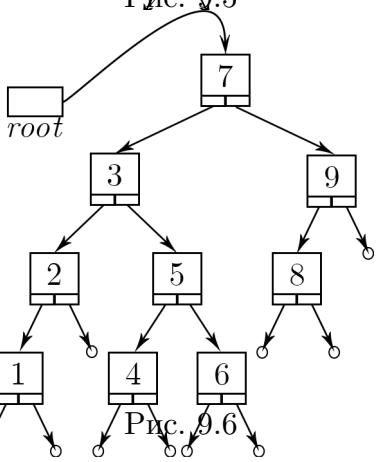

Рис. 9.6

2. Так как $2 < 7$, то будем добавлять этот элемент слева от корня. Ссылка слева от корня занята, поэтому спустимся на один уровень ниже.

3. Так как $2 < 3$, то будем добавлять этот элемент слева от узла 3. Так как эта ссылка пуста, то присваиваем левому указателю узла с числом 3 адрес нового элемента (рис. 9.5).

Аналогично добавим остальные элементы.

1. Узел 1 добавляем слева от узла 2, спускаясь на уровень ниже три раза.

2. Узел 9 вставляем справа от корневого узла 7, так как $9 > 7$.

3. Узел 5 добавляем справа от узла 3, так как $5 < 7$, но $5 > 3$.

4. Узел 4 добавляем слева от узла 5, так как $4 < 7$, $4 > 3$, $4 < 5$.

5. Узел 6 добавляем справа от узла 5, так как $6 < 7$, $6 > 3$, $6 > 5$.

6. Узел 8 добавляем слева от узла 9, так как $8 > 7$, но $8 < 9$.

В результате получается дерево, изображённое на рисунке 9.6. Листьями этого дерева являются узлы 1, 4, 6 и 8.

Напишем процедуру, которая распечатывает получившееся дерево. Числа должны выводиться на экран в порядке возрастания. Рассмотрим корневой узел 7. Сначала должны быть распечатаны все числа на поддереве слева от 7, потом само число 7, и, наконец, все числа на поддереве справа от 7. Для узла 5, например, сначала должно быть выведено число 4 на левой ветви, затем само число 5, затем его правый потомок 6. Таким образом, для каждого узла нужно выполнять одно и то же:

1. если указатель `left` не пуст, то распечатать левое поддерево;
2. распечатать число в узле;
3. если указатель `right` не пуст, то распечатать правое поддерево.

Для реализации такого алгоритма нужно использовать рекурсию. Ниже приведена реализация функции `tree_print`, выводящей дерево на экран:

```
void tree_print(Node * nd){
        if(nd->left != null) tree_print(nd->left);
        printf("%d\n", nd->val);
        if(nd->right != null) tree_print(nd->right);
}
```

Вот что будет, если вызвать функцию `tree_print` для узла 7.

- Функция `tree_print` рекурсивно вызовется для узлов 3, 2 и 1.
- Так как левый потомок у узла 1 отсутствует, то распечатается число 1.
- Так как правый потомок у узла 1 отсутствует, то управление будет передано функции `tree_print` в узле 2.
- Распечатывается число 2.
- Правый потомок у узла 2 отсутствует, поэтому управление будет передано функции `tree_print` в узле 3.
- Функция `tree_print` рекурсивно вызовется для узлов 5 и 4.
- Распечатывается число 4; так как это лист, то вызова функции `tree_print` из этого узла производиться не будет, а поэтому управление будет передано функции `tree_print` в узле 5.
- ...
- Будет выведено на экран число 9, управление будет передано функции `tree_print` в узле 7, и процедура завершит работу.

Чтобы реализовать добавление нового узла, нужно тоже использовать рекурсию:[1]

```
Node * tree_add(Node * p, Data x) {
        if(p == null){
                Node * s = malloc(sizeof(Node));
                s->val = x;
                s->left = null;
                s->right = null;
                p = s;
        }
```

[1] Не уверен, что полностью правильно воспроизвёл листинг, так как его части не видно на экране.

```
        else if(x < p->val)
                p->left = tree_add (p->left , x);
        else if(x > p->val)
                p->right = tree_add (p->right , x);
        return p;
}
```

Количество простых задач на деревья можно придумать очень много. Например, вывести на экран чётные/нечётные числа дерева, найти количество чётных/нечётных чисел, просуммировать определённые числа, и т. д. Распечатка дерева — это, по сути, полный его обход, или **обход в глубину**. Но при обходе не обязательно выводить число на экран, можно ещё что-нибудь с ним делать, так что все подобные задачи решаются небольшой модификацией предыдущего листинга. Если нужно перебирать узлы не глубже определённой величины, то можно добавить ещё один параметр функции — текущую глубину, и передавать его вниз, изменяя на единицу. Таким образом, все такие задачи решаются просто.

Что будет, если изменить порядок строк в процедуре `tree_print`? Например, так:

```
printf("%d\n", nd->val);
if(nd->left != null) print(nd->left);
if(nd->right != null) print(nd->right);
```

Тогда номер узла будет выводиться на экран раньше, чем распечатка левого поддерева. Дерево на рис. 9.6 будет распечатываться в таком порядке:

$$7, 3, 2, 1, 5, 4, 6, 9, 8. \tag{9.2}$$

Видно, что от перестановки двух строк порядок печати элементов дерева кардинально изменяется. Это следует иметь в виду на семинарских занятиях при реализации деревьев.

Если нужно уничтожить всё дерево, то точно так же нужно устроить обход в глубину и рекурсивно удалить все узлы. Узел удаляется после того, как удалены его правый и левый потомки. При этом нужно не забывать освобождать занятую узлами память с помощью функции `free`.

В лучшем случае двоичное дерево поиска будет плотным, полностью заполненным, с высотой H. Сложность поиска по такому дереву составляет $O(\log n)$, что, конечно, гораздо лучше, чем для списка. Вставка в такое дерево тоже займёт $O(\log n)$ операций.

В худшем случае дерево будет расти в глубину и выродится в цепочку элементов, например, если вставлять в пустое дерево элементы

$$1, 2, 3, 4, 5, 6, 7, 8, 9. \tag{9.3}$$

Тогда дерево становится односвязным списком, и оценки сложности его операций становятся такими же, как у списка.

Удаление одного узла из дерева требует определённой аккуратности. Если у этого узла нет детей, то его можно просто удалить, высвободив занимаемую им память, но если у него есть дети, то нельзя просто так взять и удалить его, потому что ссылки на его детей окажутся потеряны. Пусть, например, из дерева на рис. 9.6 нужно удалить узел 3. Если бы при постройке дерева узла 3 не было, то его место занял бы узел 2 или узел

4. Значит, при удалении узла 3 на его место должен стать один из этих узлов. Общее правило таково: при удалении узла нужно ставить на его место узел с ближайшим к нему значением. Например, при удалении узла 7 его место должен занять узел 6 или узел 8. Выбор между правой и левой заменой можно делать по-разному. Можно делать его произвольно, но лучше всего делать его так, чтобы минимизировать глубину дерева — тогда выполнение операций вставки и поиска будет происходить быстрее.

3. Применение деревьев при анализе литературы

Один из возможных анализов литературного произведения на плагиат — это сравнение частотных словарей. У каждого автора есть свои часто встречающиеся слова, а некоторые, наоборот, отсутствуют в его текстах. По таким особенностям часто можно распознать автора текста.

Пусть нужно провести такой анализ для поэмы Пушкина «Руслан и Людмила». Заметим, что частотные словари для прозы и стихов различаются. Для хранения слов можно использовать дерево, не обязательно бинарное. В каждом узле будет храниться одно слово, а сравниваться слова будут **лексикографически**, то есть по алфавиту.

Построим, например, бинарное дерево из слов предложения «I am a student» (рис. 9.7). Регистр букв будем игнорировать. Между словами этого предложения отношения сравнения таковы: a $<$ am $<$ I $<$ student.

Для того, чтобы хранить дерево, хранящее уникальные слова в поэме «Руслан и Людмила», нужна память под все эти слова, а также дополнительная память в каждом узле для указателей на потомков. Неплохо было бы как-то оптимизировать выделение памяти.

Пример дерева с произвольным числом потомков в каждом узле — файловая система без учёта ссылок. Узел представляется директорией, а файл — листом. Корневая директория будет корнем этого дерева.

Для больших текстов с короткими словами, как в английском языке (но не как в немецком), можно использовать следующую структуру. Пусть количество букв в алфавите — 26. Для каждого узла хранится таблица 3×26. Каждый столбец соответствует букве алфавита. В первой строке хранятся буквы алфавита.[2] Во второй строке хранится число раз, когда в тексте встретилось слово, полностью совпавшее с последовательностью букв данного узла + буквой в данном столбце. В третьей строке хранится ссылка на другой узел с последовательностью букв, равной комбинации букв данного узла + буква, соответствующая данному столбцу.

[2] По-моему, эту строку можно не добавлять в структуру, потому что номер столбца уже задаёт букву. Разве что, отображать её в таблице для наглядности.

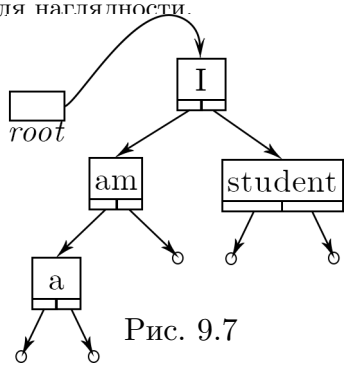

Рис. 9.7

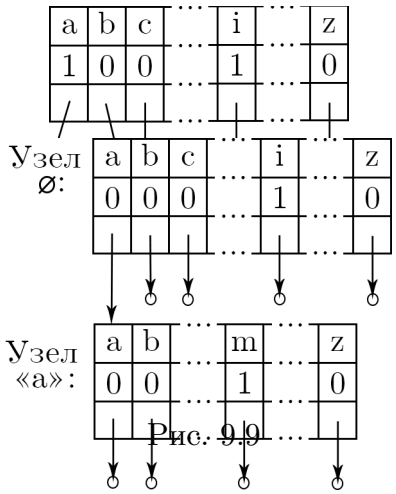

Рис. 9.9

Корневой узел тоже содержит такую структуру, которая имеет 26 ссылок на узлы, соответствующие буквам алфавита (рис. 9.8). Проиллюстрируем сказанное, заполняя пустое дерево словами. Изначально есть только корневой узел, в котором вторая строка содержит только нули, а ссылки в третьей строке указывают на `null`. Когда слов в структуре содержится мало, она довольно неэффективна, но при обработке больших текстов получается довольно компактная структура. По мере добавления новых слов структура будет расти в глубину; максимальная глубина дерева равна длине максимального слова. Будем условно связывать с корневым узлом пустое множество букв.

1. Встретилось слово «I». Так как оно в точности равно сумме пустого множества букв и буквы «i», то ставим 1 во второй строке и столбце «i» корневого узла. Ссылка в третьей строке и столбце «i» корневого узла по-прежнему нулевая.

2. Следующее слово — «am». Создаём узел, который назовём «a». Отныне разбор всех слов, начинающихся на «a», будет проходить через этот узел. Присвоим указателю в третьей строке и столбце «a» корневого узла ссылку на узел «a». В узле «a» ставим 1 в во второй строке и столбце «m», так как слово в точности равно сумме названия узла и буквы «m» (рис. 9.9).

3. Пусть следующее слово — «an». Спускаемся по ссылке из корневого узла в узел «a». В узле «a» ставим 1 в во второй строке и столбце «n».

4. Встретилось слово «cat». Создаём узел, который назовём «c». Присвоим указателю в третьей строке и столбце «c» корневого узла ссылку на узел «c». Спустимся по этой ссылке в узел «c». Создаём узел с условным названием «ca». Присвоим указателю в третьей строке и столбце «a» узла «c» ссылку на узел «ca» и спустимся по этой ссылке. Так как слово «cat» в точности равно сумме названия узла «ca» и буквы «t», то ставим 1 во второй строке и столбце «t».

5. Следующее слово — «cop». Спустимся по ссылке из корневого узла в узел «c». Создаём узел с условным названием «co». Присвоим указателю в третьей строке и столбце «o» узла «c» ссылку на узел «co» и спустимся по этой ссылке. Так как слово «cop» в точности равно сумме названия узла «co» и буквы «p», то ставим 1 во второй строке и столбце «p».

6. Пусть следующее слово — «catty». Спустимся по ссылкам в узел «ca». Создаём узел с условным названием «cat». Присвоим указателю в третьей строке и столбце «t» узла «ca» ссылку на узел «cat» и спустимся по этой ссылке. Аналогичным образом создадим узел «catt». В этом узле ставим 1 в во второй строке и столбце «y», так как слово в точности равно сумме названия узла «catt» и буквы «y».

7. Снова встретилось слово «cat». Спускаемся по ссылкам к узлу «ca» и прибавляем единицу к числу во второй строке и столбце «t». Теперь там стоит число 2, что говорит о том, то слово «cat» встретилось в тексте 2 раза.

В результате произведённый операций структура имеет вид, изображённый на рисунке 9.10. При разборе очередного слова из текста либо повторяется уже проторенный путь, либо создаётся новая ветвь. Повторяем, это хорошо для языков с короткими словами, в этом случае эта древовидная структура будет достаточно плотной. Если используется русский язык, то количество слов, начинающихся на буквы «ъ» и «ь», всегда будет равно нулю. Указатели на потомков у этих букв в корневом узле тоже будет равны нулю.

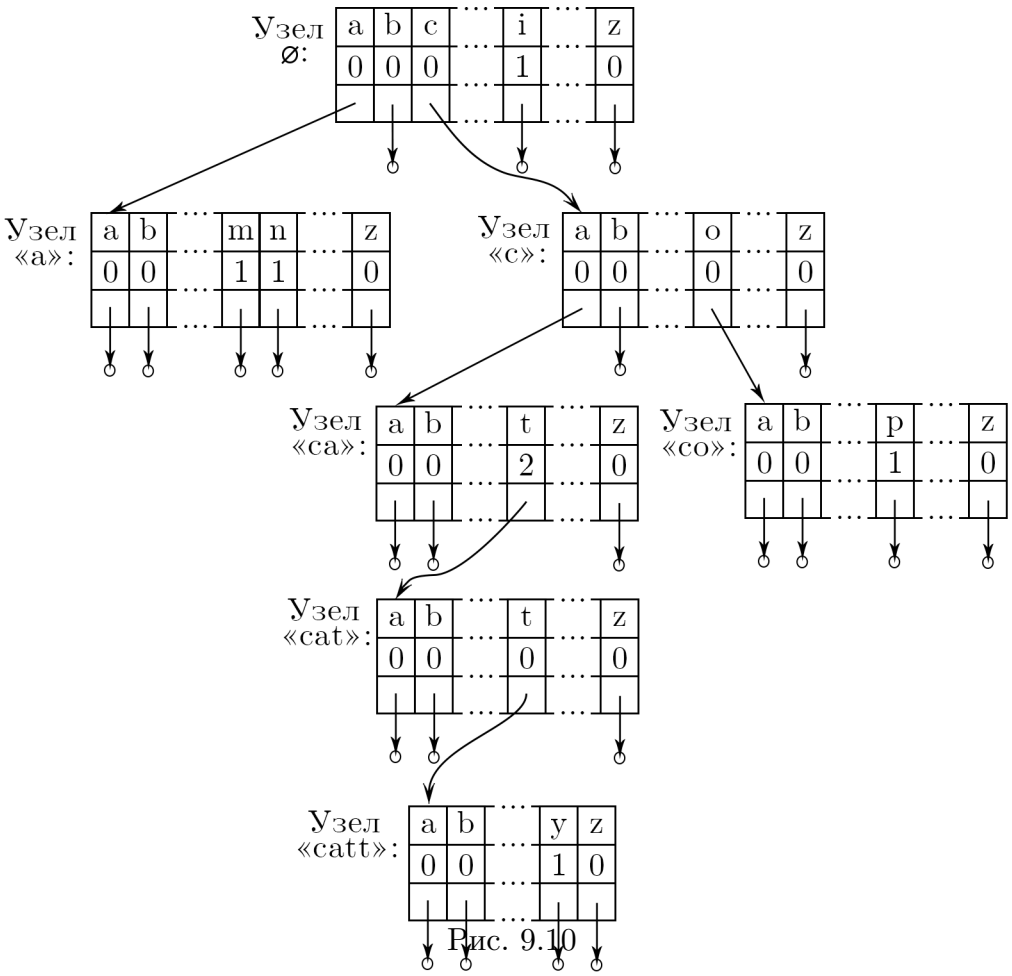

Рис. 9.10

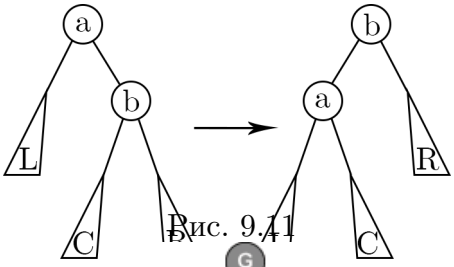

Рис. 9.11

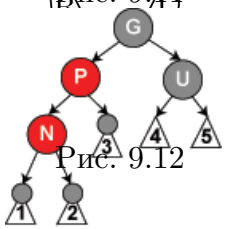

Рис. 9.12

4. Сбалансированные деревья

Попробуем избежать худших случаев в упорядоченных деревьях, когда дерево вырождается в список. Нужно как-то балансировать деревья, чтобы длинные цепи выравнивались, и степень заполнения была больше. Количество типов самобалансирующихся деревьев огромно. В данном курсе рассмотрим только один вид — **красно-чёрные деревья**.

Пусть поддерево R представляет собой длинную цепь элементов, которую требуется сделать более короткой и широкой. Корнем дерева является узел a, его правый потомок — узел b, а справа от b находится поддерево R (рис. 9.11). Все числа между a и b лежат в поддереве C. Нужно преобразовать дерево так, чтобы оно стало более сбалансированным и при этом сохранило свою упорядоченность. Должны выполняться соотношения $l < a < c < b < r$ для любых $l \in L$, $c \in C$, $r \in R$.

Преобразование на рис. 9.11 удовлетворяет данной задаче: оно сокращает максимальную глубину элемента в поддереве R на единицу и сохраняет упорядоченность. Теперь корнем дерева является узел b. L осталась левым ребёнком узла a, R — правым ребёнком узла b, а родителем для поддерева C вместо узла b стал узел a. Такое преобразование дерева называется **поворотом**.

4.1. Красно-чёрные деревья

В красно-чёрных деревьях выполняется несколько правил.

1. Каждому узлу присвоен цвет — красный или чёрный.
2. Корень дерева всегда чёрный.
3. Все воображаемые листья на месте нулевых ссылок тоже считаются чёрными.
4. Два красных узла не могут формировать связи «родитель-потомок».
5. Всякий простой путь от корня до любого листа содержит одинаковое число чёрных узлов.

Исходя из такой структуры дерева, можно гарантировать, что наименьший и наибольший путь от корня до листа различаются не более, чем в два раза: длина минимальна, если все узлы на этом пути чёрные, и максимальна, если красные и чёрные

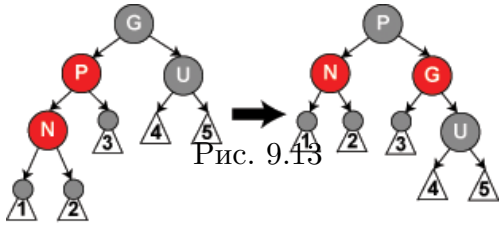

Рис. 9.13

узлы чередуются. Обозначим текущий узел как `N`, его родителя — как `P` (parent), его дедушку — как `G` (grandparent), а потомка узла `G`, не являющегося узлом `P` — как `U` (uncle) (см. рис. 9.12). Если узел `U` не занят, то считаем его равным `null`.

Если родитель — чёрный, то при вставке присваиваем его потомку красный цвет. Если при этом узлы `P` и `U` — красные, то просто перекрашиваем их в чёрный цвет, а узел `G` перекрашиваем в красный. Если узел `P` — чёрный, а `U` — красный, то осуществляется поворот структуры из 4 узлов, как показано на рис. 9.13. `P` становится корнем этого поддерева, `G` становится его правым потомком. При этом узлы `G` и `P` меняют свой цвет. Здесь процедура вставки была показана «на пальцах», подробнее разбор разных случаев можно посмотреть в интернете.

Удаление узлов тоже происходит довольно сложным образом. Обо всём этом можно почитать на Википедии, откуда лектор взяла картинки для презентации.

Далее посмотрим на движущиеся картинки. [3]

Итак, самобалансирующихся деревьев существует много. За библиотечной структурой данных типа «дерево» может стоять та или иная теория и вид самобалансирующегося дерева. Нужно иметь в виду, что при вставка элемента в такое дерево может иногда быть процессом долгим, так как при этом могут затрагиваться соседние узлы. Зато после этого поиск происходит очень быстро, так как дерево оказывается сбалансированным.

[3] Лектор показывает на проекторе видео с Youtube, которое на записи лекции не видно. Попутно она рассказывает, как происходит вставка элементов в примере с видео.