r-values, l-values, семантика копированияб перемещения

Умные указатели

* Unique\_ptr
* Shared\_ptr
* Weak\_ptr

Что такое дерево?

Двоичное дерево

Двоичное дерево поиска

Вырожденное дерево

Сбалансированное дерево поиска

Левый(правый поворот)

АВЛ дерево

Способы обхода дерева

2-4 дерево и Красно-черное дерево

Итерирование по Красно-черному дереву .

r-values, l-values, семантика копирования

**l-values и r-values**

В языке C++ все переменные являются l-values. **l-value** (в переводе *«л-значение»*, произносится как *«ел-валью»*) — это значение, которое имеет свой собственный адрес в памяти. Поскольку все переменные имеют адреса, то они все являются l-values (например, переменные a, b, c — все они являются l-values). l от слова «left», так как только значения l-values могут находиться в левой стороне в операциях присваивания (в противном случае, мы получим ошибку). Например, стейтмент 9 = 10; вызовет ошибку компилятора, так как 9 не является l-value. Число 9 не имеет своего адреса в памяти и, таким образом, мы ничего не можем ему присвоить (9 = 9 и ничего здесь не изменить).

Противоположностью l-value является r-value (в переводе*«р-значение»*, произносится как *«ер-валью»*). **r-value**— это значение, которое не имеет постоянного адреса в памяти. Примерами могут быть единичные числа (например, 7, которое имеет значение 7) или выражения (например, 3 + х, которое имеет значение х плюс 3).

Вот несколько примеров операций присваивания с использованием r-values:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int a;      // объявляем целочисленную переменную a  a = 5;      // 5 имеет значение 5, которое затем присваивается переменной а  a = 4 + 6;  // 4 + 6 имеет значение 10, которое затем присваивается переменной а    int b;      // объявляем целочисленную переменную b  b = a;      // a имеет значение 10 (исходя из предыдущих операций), которое затем присваивается переменной b  b = b;      // b имеет значение 10, которое затем присваивается переменной b (ничего не происходит)  b = b + 2;  // b + 2 имеет значение 12, которое затем присваивается переменной b |

Давайте детально рассмотрим последнюю операцию присваивания:

|  |  |
| --- | --- |
| 1 | b = b + 2; |

Здесь переменная b используется в двух различных контекстах. Слева b используется как l-value (переменная с адресом в памяти), а справа b используется как r-value и имеет отдельное значение (в данном случае, 12). При выполнении этого стейтмента, компилятор видит следующее:

|  |  |
| --- | --- |
| 1 | b = 10 + 2; |

И здесь уже понятно, какое значение присваивается переменной b.

Сильно беспокоиться о l-values или r-values сейчас не нужно, так как мы еще вернемся к этой теме на следующих уроках. Всё, что вам нужно сейчас запомнить — это то, что в левой стороне операции присваивания всегда должно находиться l-value (которое имеет свой собственный адрес в памяти), а в правой стороне операции присваивания — r-value (которое имеет какое-то значение).

**Инициализация vs. Присваивание**

В языке C++ есть две похожие концепции, которые новички часто путают: присваивание и инициализация.

После объявления переменной, ей можно **присвоить** значение с помощью оператора присваивания (знак равенства =):

|  |  |
| --- | --- |
| 1  2 | int a; // это объявление переменной  a = 8; // а это присваивание переменной a значения 8 |

В языке C++ вы можете объявить переменную и присвоить ей значение одновременно. Это называется **инициализацией**(или ***«определением»***).

|  |  |
| --- | --- |
| 1 | int a = 8; // инициализируем переменную a значением 8 |

Переменная может быть инициализирована только после операции объявления.

Хотя эти два понятия близки по своей сути и часто могут использоваться для достижения одних и тех же целей, все же в некоторых случаях следует использовать инициализацию, вместо присваивания, а в некоторых — присваивание вместо инициализации.

**Правило: Если у вас изначально имеется значение для переменной, то используйте инициализацию, вместо присваивания.**

В отличие от других языков программирования, языки Cи и C++ не инициализируют переменные определенными значениями (например, нулем) по умолчанию. Поэтому, при создании переменной, ей присваивается ячейка в памяти, в которой уже может находиться какой-нибудь мусор! Переменная без значения (со стороны программиста или пользователя) называется **неинициализированной переменной**.

Использование неинициализированных переменных может привести к ошибкам, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <iostream>    int main()  {      // Объявляем целочисленную переменную a      int a;        // Выводим значение переменной a на экран (a - это неинициализированная переменная)      std::cout << a;        return 0;  } |

В этом случае компилятор присваивает переменной a ячейку в памяти, которая в данный момент свободна (не используется). Затем значение переменной a отправляется на вывод. Но что мы увидим на экране? Ничего, так как компилятор это не пропустит — выведется ошибка, что переменная a является неинициализированной. В более старых версиях Visual Studio компилятор вообще мог бы вывести какое-то некорректное значение (например, 7177728, т.е. мусор), которое было бы содержимым той ячейки памяти, которую он присвоил нашей переменной.

Использование неинициализированных переменных является одной из самых распространенных ошибок начинающих программистов, но, к счастью, большинство современных компиляторов выдадут ошибку во время компиляции, если обнаружат неинициализированную переменную.

Хорошей практикой считается всегда инициализировать свои переменные. Это будет гарантией того, что ваша переменная всегда имеет определенное значение и вы не получите ошибку от компилятора.

**Правило: Убедитесь, что все ваши переменные в программе имеют значения (либо через инициализацию, либо через операцию присваивания).**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Advanced

Несмотря на то, что в обоих терминах есть слово «value» (значение), l-values и r-values на самом деле являются не свойствами значений, а скорее свойствами выражений.

Каждое выражение в языке C++ имеет два свойства: **тип** и **категорию значения** (определяет, можно ли результат выражения присвоить другому объекту). В C++03 и в более ранних версиях С++ l-values ​​и r-values ​​были единственными категориями значений.

О **l-value** проще всего думать, как о функции, объекте или переменной (или выражении, результатом которого является функция, объект или переменная), которая имеет свой адрес памяти. Изначально l-values были определены как «значения, которые должны находиться в левой части операции присваивания». Однако позже в язык С++ было добавлено ключевое слово const, и l-values были разделены на **две подкатегории**:

   Модифицируемые l-values, которые можно изменить (например, переменной x можно присвоить другое значение).

   Немодифицируемые l-values, которые являются const (например, константа PI).

О **r-value** проще всего думать, как «обо всем остальном, что не является l-value». Это литералы (например, 5), временные значения (например, x + 1) и [**анонимные объекты**](https://ravesli.com/urok-127-anonimnye-obekty/) (например, Fraction(7, 3)). r-values имеют область видимости выражения (уничтожаются в конце выражения, в котором находятся) и им нельзя что-либо присвоить. Этот запрет на присваивание имеет смысл, так как присваивая значение мы вызываем в объекта [**побочные эффекты**](https://ravesli.com/urok-40-inkrement-dekrement-pobochnye-effekty/#toc-1).

А поскольку r-values имеют область видимости выражения, то, если бы мы присваивали какое-либо значение для r-value, r-value либо выходило бы из области видимости, прежде чем у нас была бы возможность использовать присвоенное значение в следующем выражении (что делает операцию присваивания бесполезной), либо нам пришлось бы использовать переменную с побочным эффектом, который возникал бы больше одного раза в выражении (что, как вы уже должны знать, привело бы к неопределенным результатам!).

Для поддержки семантики перемещения в C++11 ввели **3 новые категории значений**:

   pr-values;

   x-values;

   gl-values.

Их понимание не столь важно в изучении или эффективном использовании семантики перемещения, поэтому в значительной степени мы будем их игнорировать.

До версии C++11 существовал только один тип [**ссылок**](https://ravesli.com/urok-88-ssylki/), его называли просто — «ссылка». В C++11 этот тип ссылки еще называют «ссылкой l-value». **Ссылки l-value** могут быть инициализированы только изменяемыми l-values.

**Ссылки r-value**

В C++11 добавили новый тип ссылок — ссылки r-value. **Ссылки r-value** — это ссылки, которые инициализируются только значениями r-values. Хотя ссылка l-value создается с использованием одного амперсанда, ссылка r-value создается с использованием двойного амперсанда:

|  |  |
| --- | --- |
| 1  2  3 | int x = 7;  int &lref = x; // инициализация ссылки l-value переменной x (значение l-value)  int &&rref = 7; // инициализация ссылки r-value литералом 7 (значение r-value) |

Ссылки r-value имеют **два полезных свойства**:

   Они увеличивают продолжительность жизни объекта, которым инициализируются, до продолжительности жизни ссылки r-value (ссылки l-value на константные объекты также могут это делать).

   Неконстантные ссылки r-value позволяют нам изменять значения r-values, на которые указывают ссылки r-value!

Ссылки r-value чаще всего используются в качестве параметров функции. Это наиболее полезно при [**перегрузке функций**](https://ravesli.com/urok-102-peregruzka-funktsij/), когда мы хотим, чтобы выполнение функции отличалось в зависимости от аргументов (l-values или r-values). Например:

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

Конструктор перемещения

**Конструктор копирования** используется для инициализации [**класса**](https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/) путем создания копии необходимого объекта. **Оператор присваивания копированием** (или **«копирующее присваивание»**) используется для копирования одного класса в другой (существующий) класс. По умолчанию язык C++ автоматически предоставляет конструктор копирования и оператор присваивания копированием, если вы не предоставили их сами. Предоставляемые компилятором функции выполняют [**поверхностное копирование**](https://ravesli.com/urok-145-poverhnostnoe-i-glubokoe-kopirovanie/#toc-0), что может вызывать проблемы у классов, которые работают с [**динамически выделенной памятью**](https://ravesli.com/urok-85-dinamicheskoe-vydelenie-pamyati-operatory-new-i-delete/). Одним из вариантов решения таких проблем является переопределение конструктора копирования и оператора присваивания копированием для выполнения [**глубокого копирования**](https://ravesli.com/urok-145-poverhnostnoe-i-glubokoe-kopirovanie/#toc-1).

В C++11 добавили две новые функции для работы с семантикой перемещения: конструктор перемещения и оператор присваивания перемещением. В то время как цель семантики копирования состоит в том, чтобы выполнять копирование одного объекта в другой, цель семантики перемещения состоит в том, чтобы переместить владение ресурсами из одного объекта в другой (что менее затратно, чем выполнение операции копирования).

Определение **конструктора перемещения и оператора присваивания перемещением** выполняется аналогично определению конструктора копирования и оператора присваивания копированием. Однако, в то время как функции с копированием принимают в качестве параметра константную ссылку l-value, функции с перемещением принимают в качестве параметра неконстантную [**ссылку r-value**](https://ravesli.com/urok-190-ssylki-r-value/).

## Когда вызываются конструктор перемещения и оператор присваивания перемещением?

Конструктор перемещения и оператор присваивания перемещением вызываются, когда аргументом для создания или присваивания является r-value. Чаще всего этим r-value будет литерал или временное значение (временный объект).

В большинстве случаев конструктор перемещения и оператор присваивания перемещением не предоставляются по умолчанию. Однако в тех редких случаях, когда они могут быть предоставлены по умолчанию, эти функции будут выполнять то же самое, что и конструктор копирования вместе с оператором присваивания копированием — копирование, а не перемещение.

Если мы создаем объект или выполняем присваивание, где аргументом является l-value, то единственное разумное, что мы можем сделать — это скопировать l-value. Мы не можем сказать, что изменять l-value безопасно, так как он может использоваться в программе позже. Если у нас есть выражение a = b, то нам бы очень не хотелось, чтобы b каким-либо образом был изменен.

Однако, если мы создаем объект или выполняем присваивание, где аргументом является r-value, то мы знаем, что r-value — это просто некоторый временный объект. Вместо того, чтобы копировать его (что может быть затратно), мы можем просто переместить его ресурсы (что не так затратно) в другой объект, который мы создаем или которому присваиваем текущий. Это безопасно, поскольку временный объект будет уничтожен в конце выражения в любом случае, поэтому мы можем быть уверены, что он никогда не будет повторно использован!

В C++11 через ссылки r-value мы можем изменять поведение функций в зависимости от того, чем является аргумент: r-value или l-value. А это, в свою очередь, позволяет нам принимать более разумные и эффективные решения о том, как должен работать наш код.

Далее рассмотрим на примере умного указателя – unique\_ptr.

Умные указатели

Как C++ управляет памятью?

Прежде чем объяснить, зачем нужны и какие проблемы решают умные указатели, мы кратко опишем то, как C++ управляет памятью.

Когда мы говорим про управление памятью в C++, мы неизменно обращаемся к термину ***storage duration*** (длительность хранения). Storage duration – это свойство объекта, которое описывает, когда тот попадает в память и когда её освобождает.

В C++ существует четыре вида [[1]](https://en.cppreference.com/w/cpp/language/storage_duration) storage duration:

* **Автоматическая storage duration.** Когда управление входит в область видимости объекта (также известную как ***scope***[[2]](https://en.cppreference.com/w/cpp/language/scope)), он размещается в автоматической памяти, зачастую реализованной в виде стека; когда управление покидает эту область, вызывается деструктор и память освобождается.

Листинг 1

//Подопытный класс, который будет использоваться на протяжении всей статьи

class X {

int a;

double b;

public:

void func() {};

};

int main() {

X object; //В памяти создается объект класса X

{

X object2; //Ещё один объект размещён в памяти

//В данный момент в памяти размещены и доступны оба объекта

} //Здесь object2 выходит из области видимости и уничтожается

} //Здесь уничтожается object

* **Статическая** связана с использованием спецификаторов static и extern. Объекты со статической storage duration создаются при запуске программы и удаляются при её завершении.
* **Storage duration потока** устанавливается спецификатором thread\_local. Имеющие эту storage duration объекты создаются при старте потока и удаляются при его завершении.
* **Динамическая storage duration** неразрывно связана с использованием ключевых слов new и delete.

Листинг 2

X\* ptr = nullptr; //Указатель, не указывающий ни на что

{

X\* ptr2 = new X(); //Размещение объекта класса X в памяти (также известной как динамическая память, зачастую реализованной в виде кучи). ptr2 - указатель на этот объект

ptr = ptr2; //ptr указывает на тот же объект

} //Указатель ptr2 выходит из области видимости и уничтожается, но объект, на который он указывает, остаётся в памяти

delete ptr; //Происходит вызов деструктора, а после этого объект, на который указывает ptr, удаляется из памяти

Можно сказать, что в случае с автоматической storage duration память освобождается автоматически, а в случае с динамической – вручную. Почему же тогда не использовать всегда автоматическую память?

* Чтобы использовать стек, необходимо заранее на этапе компиляции знать, как много памяти понадобится, а это известно не всегда.
* Иногда надо, чтобы объект оставался в памяти и после выхода из области видимости в которой был создан, а в случае размещения объекта на стеке это невозможно.

*Чтобы обойти эти ограничения, необходимо использовать динамическую память про использование которой мы и будем сегодня говорить.*

Что такое умные указатели и зачем они нужны?

Используем динамическую память, отлично. Теперь объекты могут покидать область видимости, где были созданы, и иметь определяемый во время выполнения размер – жизнь стала налаживаться и жаловаться как будто не на что.

Предлагаем взглянуть на следующий фрагмент кода:

Примечание

Все умные указатели доступны через включение соответствующего хедера (#include <memory>). Для краткости во фрагментах кода внутри статьи это было опущено.

Листинг 3

X\* ptr = new X();

if (func()) {

func2();

return;

}

delete ptr;

На первый взгляд, здесь всё хорошо, но есть нюансы:

* Если func() выбросит исключение, то управление не дойдёт до delete и память не освободится.
* Если func() вернёт true, то после выполнения func2() управление покинет функцию, но память не освободится, т.к. автор кода забыл добавить delete внутрь условия.
* Если бы автор забыл delete также в 6-й строке, память тоже не освободилась бы.

Тут C++ программисты решили, что с них хватит, и придумали правило, заключающееся в том, чтобы никогда не использовать new/delete. Как – увидим ниже.

Помимо проблем непосредственно с new/delete, существует проблема и с простыми указателями. Она заключается в сложности разделения указателей, которые *владеют* объектом (*owning pointer*), а значит, и ответственны за вызов new/delete, и указателей, которые *используют* объект (*non owning pointer*).

При использование простых указателей (также известных как *raw pointers*) невозможно без дополнительных комментариев или дополнительного изучения кода определить, какой указатель объектом владеет, а какой – только использует. Взгляните на следующую декларацию:

Листинг 4

int\* func();

Главная проблема здесь, что тому, кто будет вызывать функцию, совершенно неясно, должен он вызвать delete для возвращаемого указателя или за это ответственен код где-то в другой части программы. Иначе говоря, здесь не видно, является указатель владеющим или использующим.

Все вышеназванные проблемы изящно решаются умными указателями. Умные указатели в C++ – это не что-то магическое, встроенное в синтаксис языка, а не более чем набор классов из стандартной библиотеки. Разберёмся с ними один за одним.

***Благодаря умным указателям можно избежать утечек памяти и обеспечить безопасное ее использование.***

***Нет таких ситуаций в которых не нужно было бы применять умных указателей. Вопрос скорее нужно поставить так - какой тип умного указателя нужно применить в той или иной ситуации?***

Unique\_ptr

Первым умным указателем, с которым мы познакомимся, будет std::unique\_ptr [[3]](https://en.cppreference.com/w/cpp/memory/unique_ptr). Он ссылается на объект в динамической памяти и при выходе из области видимости уничтожает хранимый объект. Взглянем на пример кода ниже:

Листинг 5

{

std::unique\_ptr<X> ptr(new X()); //Объект класса X создан в динамической памяти

} //Здесь указатель ptr покидает свою область видимости и уничтожается, но перед этим удаляет из памяти объект, на который указывает

Когда std::unique\_ptr выходит из области видимости, утечки памяти не происходит, потому что в своем деструкторе умный указатель вызывает delete для объекта на который ссылается, высвобождая тем самым память.

Важно понять, что внутри умные указатели всё равно используют new/delete, они лишь позволяют программисту не делать этого и, как следствие, защищают его от ошибок.

Поскольку std::unique\_ptr разработан с учетом семантики перемещения, то семантика копирования по умолчанию отключена. Если вы хотите передать содержимое, управляемое std::unique\_ptr, то вы должны использовать семантику перемещения. В программе, приведенной выше, мы передаем содержимое std::unique\_ptr с помощью [**функции std::move()**](https://ravesli.com/urok-192-std-move/) (которая конвертирует item1 в [**r-value**](https://ravesli.com/urok-190-ssylki-r-value/), являющееся триггером для выполнения семантики перемещения вместо семантики копирования).

## Доступ к объекту, который хранит умный указатель

Умный указатель std::unique\_ptr имеет **перегруженные операторы** \* и ->, которые используются для доступа к хранимым объектам. Оператор \* возвращает ссылку на управляемый ресурс, а оператор -> возвращает указатель.

Умный указатель std::unique\_ptr не всегда может управлять объектом: либо потому, что объект был создан пустым (с использованием конструктора по умолчанию, или в объект передан в качестве параметра **[nullptr](https://ravesli.com/urok-81-nulevye-ukazateli/" \t "_blank)**), либо потому, что ресурс, которым он управлял, был перемещен в другой std::unique\_ptr. Поэтому, прежде чем использовать какой-либо из этих операторов, вы должны проверить, действительно ли std::unique\_ptr управляет ресурсом. К счастью, это легко сделать: std::unique\_ptr имеет [**неявное преобразование**](https://ravesli.com/urok-55-neyavnoe-preobrazovanie-tipov-dannyh/) в тип bool, возвращая true, если std::unique\_ptr владеет ресурсом.

В отличие от std::auto\_ptr, std::unique\_ptr достаточно умен, чтобы знать, когда использовать единичный оператор delete, а когда форму оператора delete для [**массива**](https://ravesli.com/urok-74-massivy-chast-1/), поэтому std::unique\_ptr можно использовать как с единичными объектами, так и с [**динамическими массивами**](https://ravesli.com/urok-86-dinamicheskie-massivy/).

Однако использование **[std::vector](https://ravesli.com/urok-95-vvedenie-v-std-vector-vektory/" \t "_blank)** почти всегда является лучшим выбором, чем использование std::unique\_ptr с динамическим массивом.

**Правило: Используйте std::vector вместо использования умного указателя, который владеет динамическим массивом.**

Существует два способа неправильного использования std::unique\_ptr, оба из которых легко избежать. Во-первых, не позволяйте нескольким классам «владеть» одним и тем же ресурсом. Например:

|  |  |
| --- | --- |
| 1  2  3 | Item \*item = new Item;  std::unique\_ptr<Item> item1(item);  std::unique\_ptr<Item> item2(item); |

Хотя это синтаксически допустимо, конечным результатом будет то, что и item1, и item2 попытаются удалить Item, что приведет к неопределенному поведению/результатам.

Во-вторых, не удаляйте выделенный ресурс вручную из-под std::unique\_ptr:

|  |  |
| --- | --- |
| 1  2  3 | Item \*item = new Item;  std::unique\_ptr<Item> item1(item);  delete item; |

Если вы это сделаете, std::unique\_ptr попытается удалить уже удаленный ресурс, что опять приведет к неопределенному поведению/результатам.

Обратите внимание, функция std::make\_unique() предотвращает непреднамеренное возникновение обеих ситуаций, приведенных выше.

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

Вот тут реализацию добавить с объяснениями

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

Shared\_ptr && weak\_ptr

std::unique\_ptr и правда хорош, но он не поможет в ситуации, когда мы хотим, чтобы несколько объектов работали с одним общим ресурсом и чтобы в момент, когда все эти объекты были выгружены из памяти, за ненадобностью автоматически выгрузился бы и ресурс.

В такой ситуации необходимо использовать std::shared\_ptr [[8]](https://en.cppreference.com/w/cpp/memory/shared_ptr). Этот умный указатель разрешает объекту иметь несколько владельцев, а когда все владельцы уничтожаются, уничтожается и объект. Такое поведение достигается за счёт наличия специального счётчика ссылок внутри std::shared\_ptr. Каждый раз, когда такой указатель копируется, счётчик инкрементируется, а когда один из указателей уничтожается – декрементируется. В момент, когда счётчик достигает нуля, объект уничтожается. Посмотрим на код:

Листинг 11

{

std::shared\_ptr<X> ptr = std::make\_shared<X>(); //Создаётся объект

{

std::shared\_ptr<X> ptr2 = ptr; //Теперь у объекта два владельца, выраженных в виде ptr и ptr2

} //ptr2 выходит из области видимости, но объект не освобождается, потому что есть ptr, который по-прежнему ссылается на него

} //ptr выходит из области видимости, и объект уничтожается

Существуют ситуации, когда объект A должен ссылаться на B, а B – на A. Это называется циклической ссылкой (*cyclic reference/circular dependency*). В таком случае оба объекта никогда не будут выгружены из памяти.

Чтобы разорвать цикличность, необходимо использовать std::weak\_ptr[[9]](https://en.cppreference.com/w/cpp/memory/weak_ptr" \t "_blank). Это фактически умный указатель non owning, предназначенный для использования именно с std::shared\_ptr. Копирование std::weak\_ptr не увеличивает счётчик в std::shared\_ptr, а значит и не защищает объект от уничтожения. При этом всегда имеется возможность проверить, существует ли ещё объект, на который ссылается std::weak\_ptr, или нет. Внимание на код:

Листинг 12

class Owner {

public:

std::shared\_ptr<X> owningPtr;

Owner() {

owningPtr = std::make\_shared<X>();

}

};

class User {

std::weak\_ptr<X> usingPtr;

public:

User(std::weak\_ptr<X> object) {

usingPtr = object;

}

void use() {

if (std::shared\_ptr<X> object = usingPtr.lock()) { //Попытка получить оригинальный std::shared\_ptr из std::weak\_ptr, если возвращён пустой std::shared\_ptr, значит, объект уже был удалён

object->func();

} else {

//Объект уже удалён

}

}

};

int main() {

Owner owner;

User user(owner.owningPtr);

user.use();

}

Вообще говоря, std::weak\_ptr необходимо использовать всегда, когда надо ссылаться на управляемый std::shared\_ptr объект, но *не защищать* его от уничтожения.

///////////////////////////////////////////////////////////////////////////////////////////////

Пошаговая реализация.

///////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////

Что такое дерево?

Двоичное дерево

Двоичное дерево поиска

Вырожденное дерево

Сбалансированное дерево поиска

////////////////////////////////////////////

Что такое дерево?

**Дерево** — одна из наиболее широко распространённых [структур данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), эмулирующая [древовидную структуру](https://ru.wikipedia.org/wiki/%D0%94%D1%80%D0%B5%D0%B2%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0) в виде набора связанных узлов. Является связным [графом](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), не содержащим циклы.

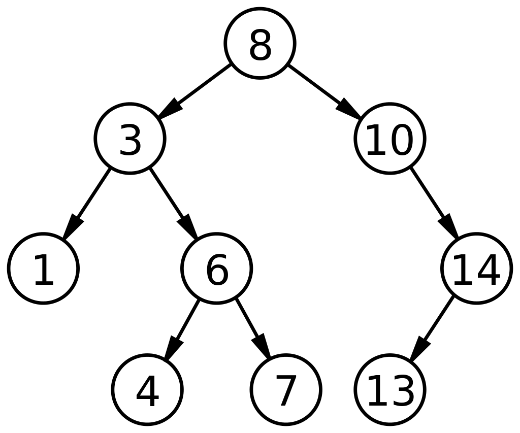
## Справочник терминов

* **Корень** — самый верхний узел дерева.
* **Ребро** — связь между двумя узлами.
* **Лист** — узел, не имеющий узлов-потомков на дереве.
* **Высота** — это длина самого дальнего пути к листу.
* **Глубина** — длина пути к корню.

## Бинарные деревья

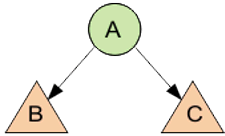
Теперь рассмотрим особый тип деревьев, называемых бинарными или двоичными деревьями.

“В информатике бинарным (двоичным) деревом называется иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.



**Способы обхода дерева**

Пусть имеем дерево, где A — корень, B и C — левое и правое поддеревья.



Существует три способа обхода дерева:

* Обход дерева сверху вниз (в прямом порядке): A, B, C — префиксная форма.
* Обход дерева в симметричном порядке (слева направо): B, A, C — инфиксная форма.
* Обход дерева в обратном порядке (снизу вверх): B, C, A — постфиксная форма.

***Бинарное (двоичное) дерево поиска*** – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

* оба поддерева – левое и правое, являются двоичными деревьями поиска;
* у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
* у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

struct tnode \* addnode(int x, tnode \*tree) {  
  if (tree == NULL) { // Если дерева нет, то формируем корень  
    tree =new tnode; // память под узел  
    tree->field = x;   // поле данных  
    tree->left =  NULL;  
    tree->right = NULL; // ветви инициализируем пустотой  
  }else  if (x < tree->field)   // условие добавление левого потомка  
    tree->left = addnode(x,tree->left);  
  else    // условие добавление правого потомка  
    tree->right = addnode(x,tree->right);  
  return(tree);  
}

void freemem(tnode \*tree) {  
  if(tree!=NULL) {  
    freemem(tree->left);  
    freemem(tree->right);  
    delete tree;  
  }  
}

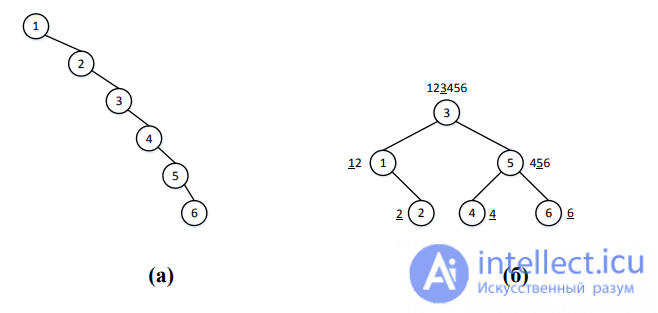
//////////////////////////////////////

Удаление с комментариями

///////////////////////////////////////

Сбалансированное дерево поиска

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.



Дерево , как показано на рисунке а) называется вырожденным, в котором высота равна количеству узлов, что делает дерево поиска тем же, что и односвязный список со сложность операций O(n).

Дерево из рисунка а) получилось после поочередной вставки упорядоченных ключей от 1 до 6. Те же ключи использовались и при построении дерева б) ,но порядок вставки был другим, в итоге б) получилось более сбалансированным, чем а).

Что бы о сбалансированности можно было судить не только по внешнему виду деревьев, а ещё и как-то выразить в числах, скажем, что коэффициент балансировки для вершины – это разность высот её поддеревьев.

Сбалансированные деревья поиска помогают избежать таких ситуаций, когда высота дерева далека от предполагаемой log2(n).

Наиболее простое, но наглядное сбалансированное дерево поиска – это АВЛ-дерево.

Основное свойство АВЛ-дерева – для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Как только разность высот поддеревьев становиться равной 2, происходит перебалансировка, которая меняет связи дочерних узлов, делая разницу высот поддеревьев не большей единицы.

Перебалансировка происходит путем вращений.

Используются 4 типа вращений:

1.**Малое левое вращение** [](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_LR.GIF) Данное вращение используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота С <= высота R.

2.**Большое левое вращение** [](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_BR.GIF) Данное вращение используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота c-поддерева > высота R.

3.**Малое правое вращение** [](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_LL.GIF) Данное вращение используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота С <= высота L.

4.**Большое правое вращение** [](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AVL_BL.GIF) Данное вращение используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота c-поддерева > высота L.

В каждом случае достаточно просто доказать то, что операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться. Также большое вращение это комбинация правого и левого малого вращения. Из-за условия балансированности высота дерева О(log(N)), где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций.

////////////////////////////////////////////////////////////////

/Код поворотов.

////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////

/Код вставки

////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////

/Код удаления

////////////////////////////////////////////////////////////////////

2-3 дерево и Красно-черное дерево

### **Два-три дерево**

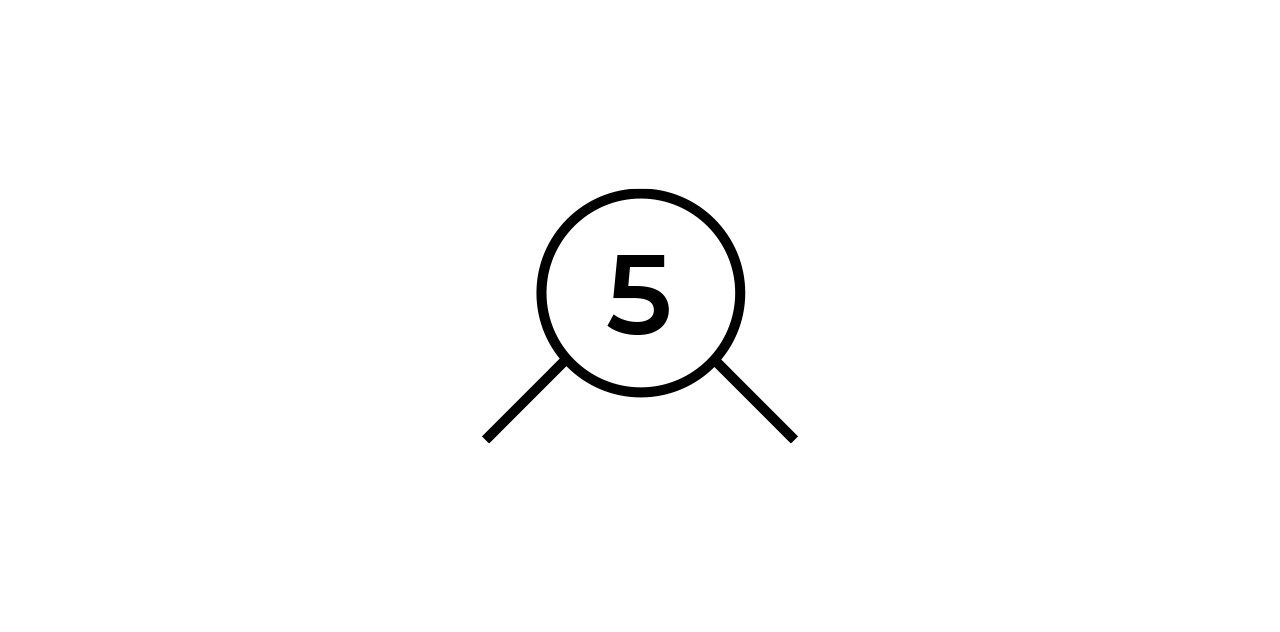
Чтобы понять красно-черное дерево, нужно понять два-три дерево

Забегая вперед, скажу, что два-три дерево - это, по сути, родитель нашего кчд, поэтому важно начать именно с него. Поймем два-три дерево - поймем и кчд.

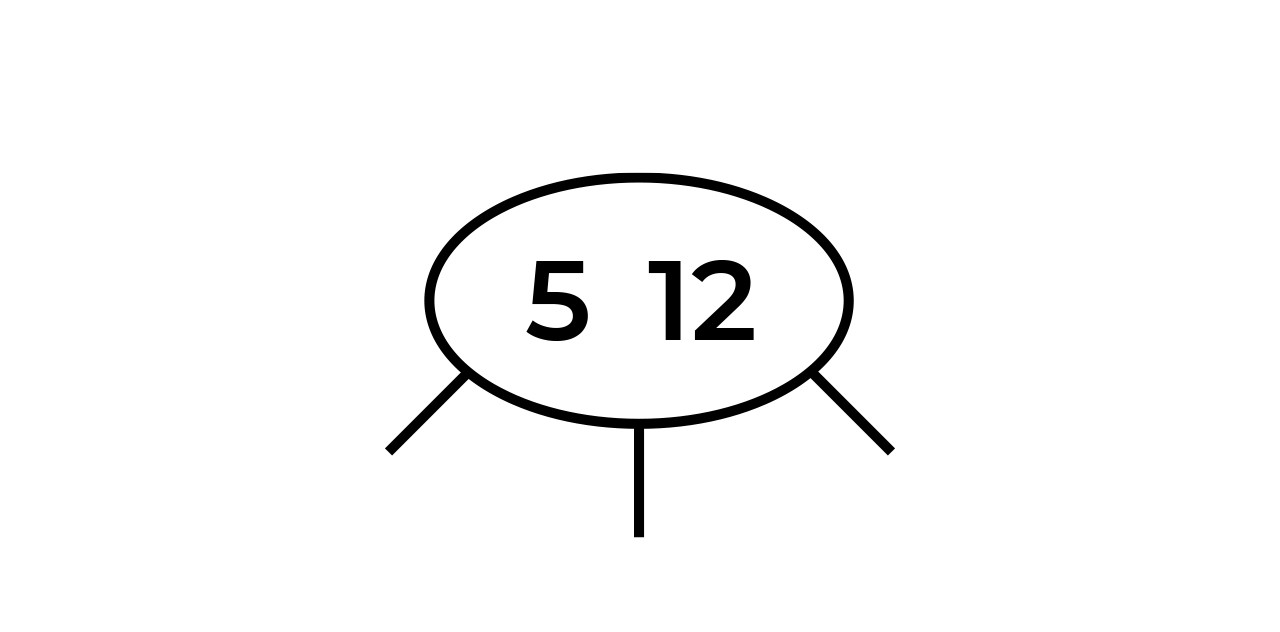
**Два-три дерево** - [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), напоминающий по структуре дерево. В нодах два-три дерева может быть одно или два значения и два или три потомка (от чего зависит количество значений и потомков ноды, узнаем ниже). Ноду с одним значением и двумя потомками будем называть 2-нода, ноду с двумя значениями и тремя потомками - 3-нода. Объяснение я начну с создания такого дерева: это наглядно и просто. Но некоторые уточнения нужны все же вначале:

1. Добавляя элемент, мы всегда спускаемся вниз по дереву.
2. Дерево отсортировано классически - меньшие значения находятся слева, бОльшие - справа.
3. Два-три дерево - отсортированное, сбалансированное дерево.

Итак, начнем с первой ноды, это число 5. Тут все просто - 5 становится корнем.

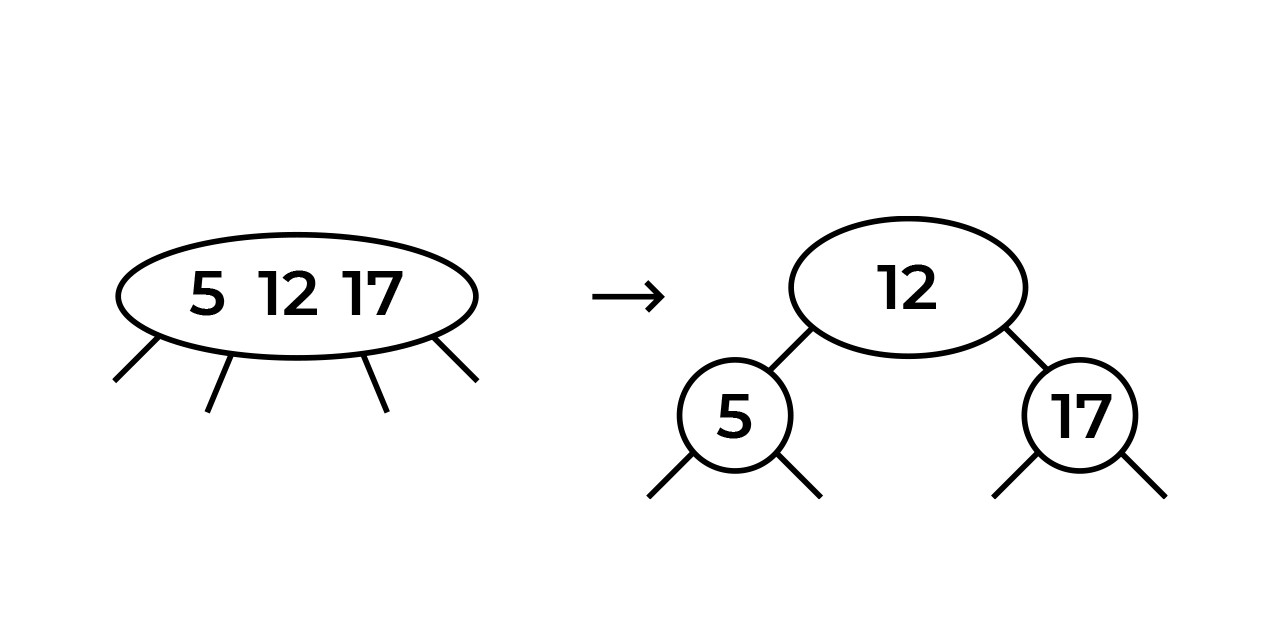


Добавим число 12. Число 12 мы так же добавляем в корень (помним, что нода может иметь два значения), но теперь нам нужно "отсортировать" нашу ноду (сортировать два элемента, ха), т.е. уложить их в порядке возрастания. В итоге получается нода 5-12.



Добавим следующее число. Пусть это будет 17. Давайте пока добавим наш элемент в единственную ноду и снова отсортируем ее. Получилась нода 5-12-17.

Внимательный читатель заметит тут подвох - выше я говорил о том, что наши ноды могут содержать не более двух элементов. Вот тут и происходит магия! Мы берем средний элемент нашей ноды и "просачиваем" его наверх. Итог виден на картинке. Корнем стало число 12, левым сыном число 5, правым - 17.

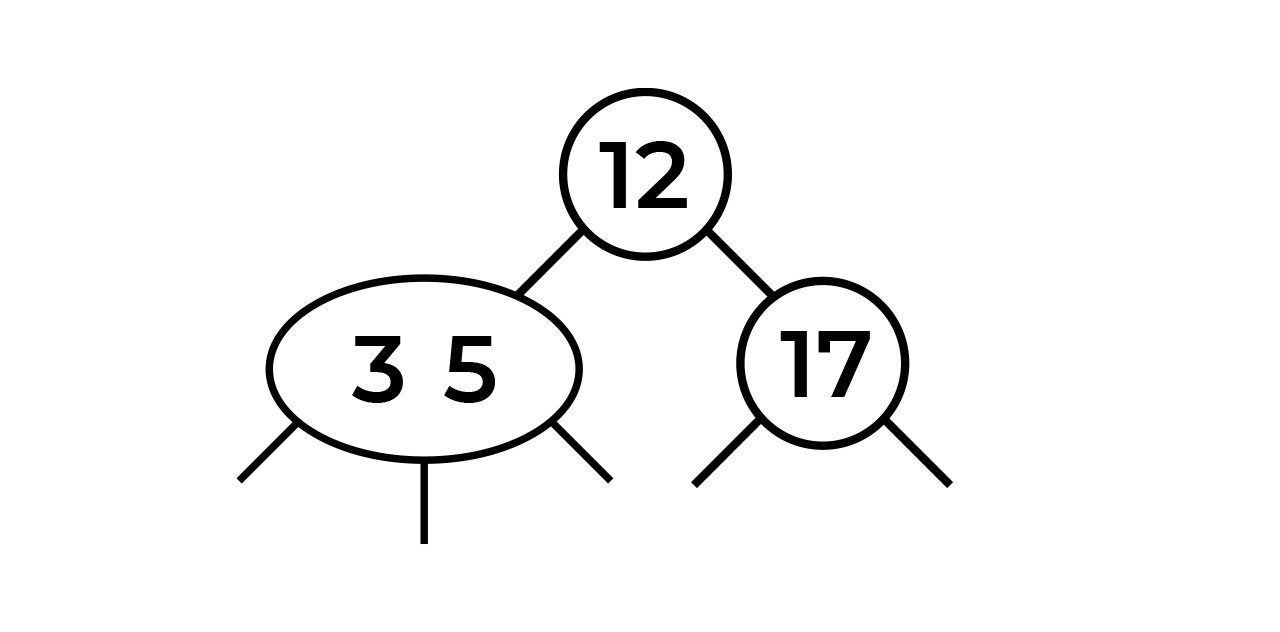


То, что мы сделали выше, можно назвать балансировкой два-три дерева. Правило в этой балансировке простое: если в ноде оказывается три значения, среднее значение мы "просачиваем" вверх. Алгоритм действий зависит от 3 условий:

1. Нода является корнем. Тогда ничего не остается, как создать новую ноду с одним значением и сделать ее новым корнем (как в нашем случае).
2. Родительская нода имеет одно значение. Тогда мы просто добавляем значение к родителю и завершаем балансировку (при этом у родителя появляется третий потомок).
3. Родительская нода имеет два значения. Тогда мы снова просачиваем значение вверх, пока не придем к пункту один или два.

Второй и третий случай балансировки будут рассмотрены ниже.

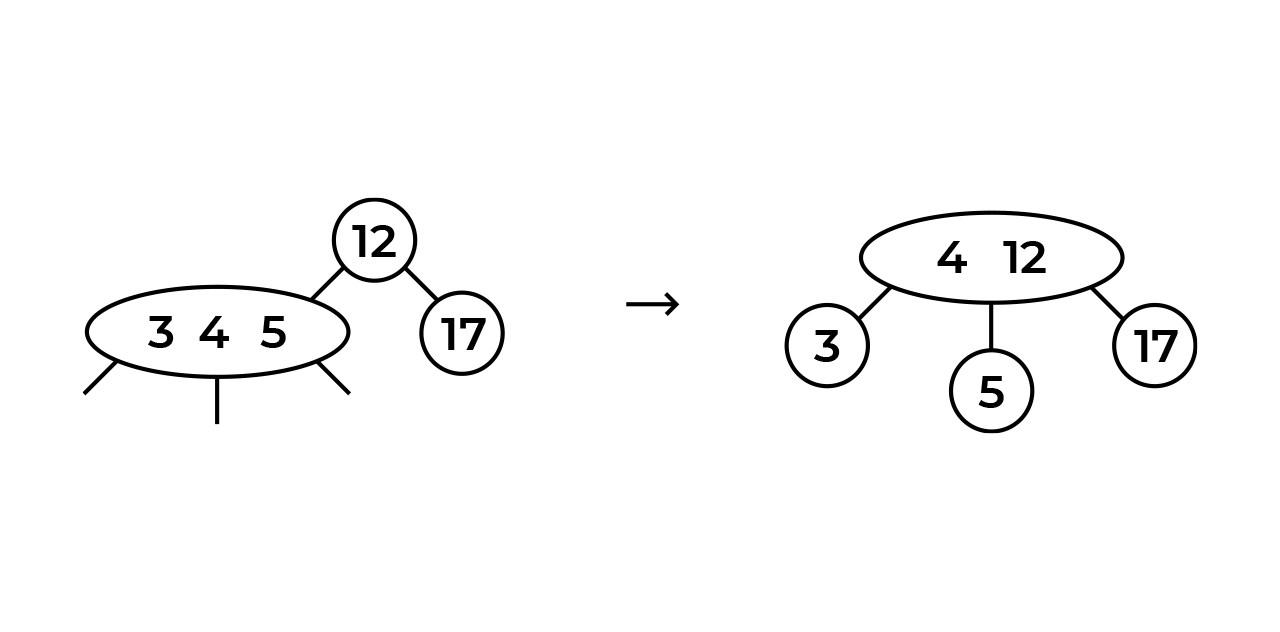
Окей, идем дальше. Давайте добавим число 3. Так как теперь мы не ограничиваемся одной нодой, спускаемся вниз. Понятно, что спуститься надо влево и добавить 3 к ноде со значением 5. Не забываем расставить 3 и 5 в нужном порядке. В итоге получилась нода 3-5.



Потерпите, мы близимся к самому интересному:)

Давайте добавим число 4 которое также пойдет влево и присоединится к ноде 3-5. Получится нода 3-4-5, которую, как мы уже знаем, нужно привести к нужному виду. Что делаем? Балансируем наше дерево, т.е. "просачиваем" значение 4 наверх.

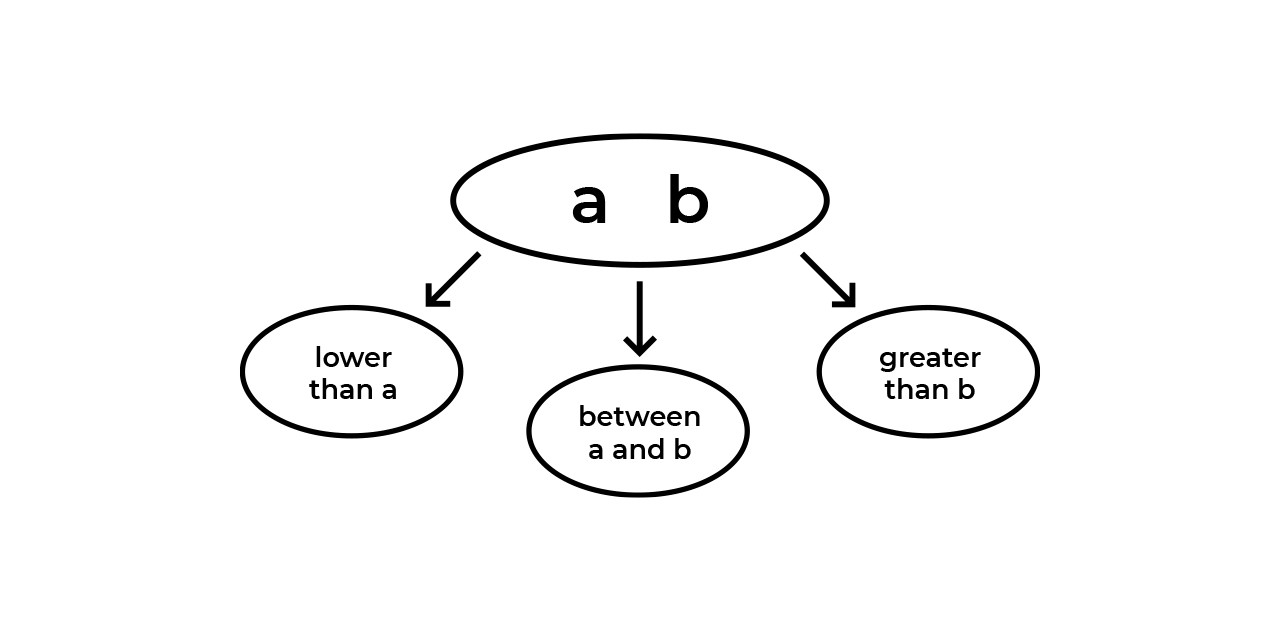
Теперь самое интересное. Помимо того, что мы добавим 4 к корню, мы так же добавим корню третьего потомка - это будет нода, которая была больше 4. В нашем случае это 5. Картина будет выглядеть так:



Почему 5 не могла остаться на месте в своей ноде? Тут мы вспоминаем правило отсортированного дерева: значения меньше - слева, значения больше - справа. И так как 5 больше 4, мы не можем оставить 5 слева, т.к. это нарушит наш инвариант. Поэтому ноде со значением 5 ничего не остается, как "переехать", и стать потомком ноды 4-12 (кстати, если бы у 5 были потомки, они так же "переехали" бы вместе с родителем).

Тут нужно сделать небольшую паузу и объяснить, как ориентироваться в нодах с тремя потомками. Все просто:

1. Значение, что меньше левого значения в ноде, будет левым потомком.
2. Значение, что больше левого, но меньше правого значения в ноде, будет средним потомком.
3. Значение, что больше правого значения в ноде, будет правым потомком.



А теперь посмотрите на то, что получилось. Смело можно заявить, что это отсортированное, сбалансированное два-три дерево. Значения меньше лежат слева, значения больше - справа (тут, как и в случае со значениями 4-5-12, мы считаем, что 5 лежит справа от 4 и слева от 12). Корень имеет два значения и три потомка, что абсолютно соответствует описанию дерева выше. Сейчас вы можете сами попробовать добавить любое значение, чтобы удостовериться, что вы поняли логику (что произойдет с деревом, если добавить значение 7? А затем 9?).

Окей, с самим деревом, я надеюсь, разобрались. Ноды добавляются, дерево балансируется, искать можно, все супер. Но есть нюансы.

Главный минус такой структуры в том, что она, в отличие от бинарного дерева, неудобна в реализации. Нужно следить за количеством потомков и значением, плюс за их порядком, балансировкой (и это я еще не говорил про удаление).

В итоге пришлось бы реализовывать 3 класса для разных типов узлов со сложными методами.

Красно-черное дерево решает эту проблему.

Красно-чёрные деревья являются одними из наиболее активно используемых на практике самобалансирующихся деревьев поиска. В частности, контейнеры set и map в большинстве реализаций библиотеки [STL](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BE%D0%B2) языка [C++](https://ru.wikipedia.org/wiki/C%2B%2B)[[3]](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#cite_note-3), класс TreeMap языка [Java](https://ru.wikipedia.org/wiki/Java" \o "Java)[[4]](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#cite_note-4), так же, как и многие другие реализации [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2) в различных библиотеках, основаны на красно-чёрных деревьях.

Красно-чёрные деревья более популярны, чем идеально сбалансированные деревья, т.к. в последних может тратиться слишком много ресурсов на операции удаления из дерева и поддержание необходимой сбалансированности. После вставки или удаления требуется операция перекраски, требующая (O(log *n*) или O(1)) смен цветов (что на практике довольно быстро) и не более чем трёх поворотов дерева (для вставки — не более двух). Хотя вставка и удаление сложны, их трудоёмкость остается O(log *n*).

Красно-чёрное дерево — [двоичное дерево поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0), в котором каждый узел имеет атрибут *цвета*. При этом:

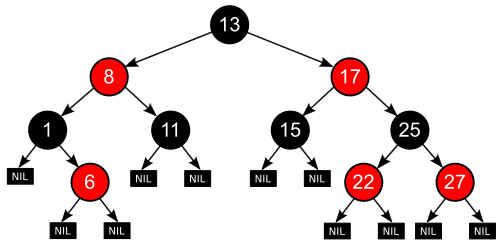
1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

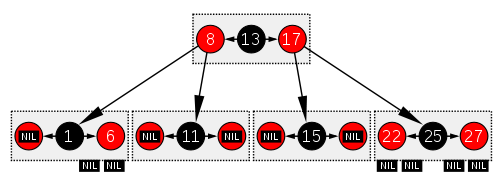
Благодаря этим ограничениям путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Чтобы понять, как это работает, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B. Тогда кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из 2B-1 узлов, попеременно красных и чёрных.

Красно-чёрное дерево схоже по структуре с [B-деревом](https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) с параметром 4, в котором каждый узел может содержать от 1 до 3 значений и, соответственно, от 2 до 4 указателей на потомков. В таком В-дереве каждый узел будет содержать только одно значение, соответствующее значению чёрного узла красно-чёрного дерева с необязательным значениями до и/или после него в том же узле, оба из которых соответствуют эквивалентным красным узлам красно-чёрного дерева.

Один из способов увидеть эту эквивалентность — «поднять» красные узлы в графическом представлении красно-чёрного дерева так, чтобы они оказались на одном уровне по горизонтали со своими предками чёрными узлами, образуя страницу. В В-дереве, или в модифицированном графическом представлении красно-чёрного дерева, у всех листовых узлов глубина одинаковая.





Новый узел в красно-чёрное дерево добавляется на место одного из листьев, окрашивается в красный цвет и к нему прикрепляется два листа (так как листья являются абстракцией, не содержащей данных, их добавление не требует дополнительной операции). Что происходит дальше, зависит от цвета близлежащих узлов. Заметим, что:

* Свойство 3 (Все листья чёрные) выполняется всегда.
* Свойство 4 (Оба потомка любого красного узла — чёрные) может нарушиться только при добавлении красного узла, при перекрашивании чёрного узла в красный или при повороте.
* Свойство 5 (Все пути от любого узла до листовых узлов содержат одинаковое число чёрных узлов) может нарушиться только при добавлении чёрного узла, перекрашивании красного узла в чёрный (или наоборот), или при повороте.

*Примечание*: Буквой **N** будем обозначать текущий узел (окрашенный красным). Сначала это новый узел, который вставляется, но эта процедура может применяться рекурсивно к другим узлам (смотрите случай 3). **P** будем обозначать предка **N**, через **G** обозначим дедушку **N**, а **U** будем обозначать дядю (узел, имеющий общего родителя с узлом **P**). Отметим, что в некоторых случаях роли узлов могут меняться, но, в любом случае, каждое обозначение будет представлять тот же узел, что и в начале. Любой цвет, изображенный на рисунке, либо предполагается в данном случае, либо получается из других соображений.

Каждый случай рассматривается с примерами кода на [языке C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)). Определение структуры узла может выглядеть следующим образом:

**enum** node\_colors { RED, BLACK };

**struct** **node** {

**struct** **node** \*parent, \*left, \*right;

**enum** node\_colors color;

int key;

};

Дядя и дедушка текущего узла могут быть найдены с помощью функций:

**struct** **node** \*

grandparent(**struct** **node** \*n)

{

**if** ((n != NULL) && (n->parent != NULL))

**return** n->parent->parent;

**else**

**return** NULL;

}

**struct** **node** \*

uncle(**struct** **node** \*n)

{

**struct** **node** \*g = grandparent(n);

**if** (g == NULL)

**return** NULL; *// No grandparent means no uncle*

**if** (n->parent == g->left)

**return** g->right;

**else**

**return** g->left;

}

Левый и правый поворот дерева может быть выполнен так:

void

rotate\_left(**struct** **node** \*n)

{

**struct** **node** \*pivot = n->right;

pivot->parent = n->parent; */\* при этом, возможно, pivot становится корнем дерева \*/*

**if** (n->parent != NULL) {

**if** (n->parent->left==n)

n->parent->left = pivot;

**else**

n->parent->right = pivot;

}

n->right = pivot->left;

**if** (pivot->left != NULL)

pivot->left->parent = n;

n->parent = pivot;

pivot->left = n;

}

void

rotate\_right(**struct** **node** \*n)

{

**struct** **node** \*pivot = n->left;

pivot->parent = n->parent; */\* при этом, возможно, pivot становится корнем дерева \*/*

**if** (n->parent != NULL) {

**if** (n->parent->left==n)

n->parent->left = pivot;

**else**

n->parent->right = pivot;

}

n->left = pivot->right;

**if** (pivot->right != NULL)

pivot->right->parent = n;

n->parent = pivot;

pivot->right = n;

}

**Случай 1:** Текущий узел **N** в корне дерева. В этом случае, он перекрашивается в чёрный цвет, чтобы оставить верным Свойство 2 (Корень — чёрный). Так как это действие добавляет один чёрный узел в каждый путь, Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается.

void

insert\_case1(**struct** **node** \*n)

{

**if** (n->parent == NULL)

n->color = BLACK;

**else**

insert\_case2(n);

}

**Случай 2:** Предок **P** текущего узла чёрный, то есть Свойство 4 (Оба потомка каждого красного узла — чёрные) не нарушается. В этом случае дерево остаётся корректным. Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается, потому что текущий узел **N** имеет двух чёрных листовых потомков, но так как **N** является красным, путь до каждого из этих потомков содержит такое же число чёрных узлов, что и путь до чёрного листа, который был заменен текущим узлом, так что свойство остается верным.

void

insert\_case2(**struct** **node** \*n)

{

**if** (n->parent->color == BLACK)

**return**; */\* Tree is still valid \*/*

**else**

insert\_case3(n);

}

*Примечание:* В следующих случаях предполагается, что у **N** есть дедушка **G**, так как его родитель **P** является красным, а если бы он был корнем, то был бы окрашен в чёрный цвет. Таким образом, **N** также имеет дядю **U**, хотя он может быть листовым узлом в случаях 4 и 5.

|  |
| --- |
| [Схема случая 3](https://commons.wikimedia.org/wiki/File:Red-black_tree_insert_case_3.png?uselang=ru)  **Случай 3:** Если и родитель **P,** и дядя **U** — красные, то они оба могут быть перекрашены в чёрный, и дедушка **G** станет красным (для сохранения свойства 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов)). Теперь у текущего красного узла **N** чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка **G** теперь может нарушить свойства 2 (Корень — чёрный) или 4 (Оба потомка каждого красного узла — чёрные) (свойство 4 может быть нарушено, так как родитель **G** может быть красным). Чтобы это исправить, вся процедура рекурсивно выполняется на **G** из случая 1. |

void

insert\_case3(**struct** **node** \*n)

{

**struct** **node** \*u = uncle(n), \*g;

**if** ((u != NULL) && (u->color == RED)) {

*// && (n->parent->color == RED) Второе условие проверяется в insert\_case2, то есть родитель уже является красным.*

n->parent->color = BLACK;

u->color = BLACK;

g = grandparent(n);

g->color = RED;

insert\_case1(g);

} **else** {

insert\_case4(n);

}

}

*Примечание:* В оставшихся случаях предполагается, что родитель **P** является левым потомком своего предка. Если это не так, необходимо поменять *лево* и *право*. Примеры кода позаботятся об этом.

|  |
| --- |
| [Схема случая 4](https://commons.wikimedia.org/wiki/File:Red-black_tree_insert_case_4.png?uselang=ru)  **Случай 4:** Родитель **P** является красным, но дядя **U** — чёрный. Также, текущий узел **N** — правый потомок **P**, а **P** в свою очередь — левый потомок своего предка **G**. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла **N** и его предка **P**. Тогда, для бывшего родительского узла **P** в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено. Вращение приводит к тому, что некоторые пути (в поддереве, обозначенном «1» на схеме) проходят через узел **N**, чего не было до этого. Это также приводит к тому, что некоторые пути (в поддереве, обозначенном «3») не проходят через узел **P**. Однако, оба эти узла являются красными, так что Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается при вращении. Однако Свойство 4 всё ещё нарушается, но теперь задача сводится к Случаю 5. |

void

insert\_case4(**struct** **node** \*n)

{

**struct** **node** \*g = grandparent(n);

**if** ((n == n->parent->right) && (n->parent == g->left)) {

rotate\_left(n->parent);

n = n->left;

*/\**

*\* rotate\_left может быть выполнен следующим образом, учитывая что уже есть \*g = grandparent(n)*

*\**

*\* struct node \*saved\_p=g->left, \*saved\_left\_n=n->left;*

*\* g->left=n;*

*\* n->left=saved\_p;*

*\* saved\_p->right=saved\_left\_n;*

*\**

*\*/*

} **else** **if** ((n == n->parent->left) && (n->parent == g->right)) {

rotate\_right(n->parent);

n = n->right;

*/\**

*\* rotate\_right может быть выполнен следующим образом, учитывая что уже есть \*g = grandparent(n)*

*\**

*\* struct node \*saved\_p=g->right, \*saved\_right\_n=n->right;*

*\* g->right=n;*

*\* n->right=saved\_p;*

*\* saved\_p->left=saved\_right\_n;*

*\**

*\*/*

}

insert\_case5(n);

}

|  |
| --- |
| [Схема случая 5](https://commons.wikimedia.org/wiki/File:Red-black_tree_insert_case_5.png?uselang=ru)  **Случай 5:** Родитель **P** является красным, но дядя **U** — чёрный, текущий узел **N** — левый потомок **P** и **P** — левый потомок **G**. В этом случае выполняется поворот дерева на **G**. В результате получается дерево, в котором бывший родитель **P** теперь является родителем и текущего узла **N** и бывшего дедушки **G**. Известно, что **G** — чёрный, так как его бывший потомок **P** не мог бы в противном случае быть красным (без нарушения Свойства 4). Тогда цвета **P** и **G** меняются и в результате дерево удовлетворяет Свойству 4 (Оба потомка любого красного узла — чёрные). Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) также остается верным, так как все пути, которые проходят через любой из этих трех узлов, ранее проходили через **G**, поэтому теперь они все проходят через **P**. В каждом случае, из этих трёх узлов только один окрашен в чёрный. |

void

insert\_case5(**struct** **node** \*n)

{

**struct** **node** \*g = grandparent(n);

n->parent->color = BLACK;

g->color = RED;

**if** ((n == n->parent->left) && (n->parent == g->left)) {

rotate\_right(g);

} **else** {

rotate\_left(g);

}

}

### Удаление**[**[**править**](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE&veaction=edit&section=5)**|**[**править код**](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE&action=edit&section=5)**]**

При удалении узла с двумя нелистовыми потомками в обычном двоичном дереве поиска мы ищем либо наибольший элемент в его левом поддереве, либо наименьший элемент в его правом поддереве и перемещаем его значение в удаляемый узел. Затем мы удаляем узел, из которого копировали значение. Копирование значения из одного узла в другой не нарушает свойств красно-чёрного дерева, так как структура дерева и цвета узлов не изменяются. Стоит заметить, что новый удаляемый узел не может иметь сразу два дочерних нелистовых узла, так как в противном случае он не будет являться наибольшим/наименьшим элементом. Таким образом, получается, что случай удаления узла, имеющего два нелистовых потомка, сводится к случаю удаления узла, содержащего как максимум один дочерний нелистовой узел. Поэтому дальнейшее описание будет исходить из предположения существования у удаляемого узла не более одного нелистового потомка.

Будем использовать обозначение **M** для удаляемого узла; через **C** обозначим потомка **M**, который также будем называть просто «его потомок». Если **M** имеет нелистового потомка, возьмем его за **C**. В противном случае за **C** возьмем любой из листовых потомков.

Если **M** является красным узлом, заменим его своим потомком **C**, который по определению должен быть чёрным. (Это может произойти только тогда, когда **M** имеет двух листовых потомков, потому что если красный узел **M** имеет чёрного нелистового потомка с одной стороны, а с другой стороны — листового, то число чёрных узлов на обеих сторонах будет различным, таким образом дерево станет недействительным красно-чёрным деревом из-за нарушения Свойства 5.) Все пути через удаляемый узел просто будут содержать на один красный узел меньше, предок и потомок удаляемого узла должны быть чёрными, так что Свойство 3 («Все листья — чёрные») и Свойство 4 («Оба потомка красного узла — чёрные») все ещё сохраняется.

Другим простым является случай, когда **M** — чёрный и **C** — красный. Простое удаление чёрного узла нарушит Свойство 4 («Оба потомка красного узла — чёрные») и Свойство 5 («Всякий простой путь от данного узла до любого листового узла, содержит одинаковое число чёрных узлов»), но если мы перекрасим **С** в чёрный, оба эти свойства сохранятся.

Сложным является случай, когда и **M** и **C** — чёрные. (Это может произойти только тогда, когда удаляется чёрный узел, который имеет два листовых потомка, потому что если чёрный узел **M** имеет чёрного нелистового потомка с одной стороны, а с другой — листового, то число чёрных узлов на обеих сторонах будет различным и дерево станет недействительным красно-чёрным деревом из-за нарушения Свойства 5.) Мы начнём с замены узла **M** своим потомком **C**. Будем называть этого потомка (в своем новом положении) **N**, а его «брата» (другого потомка его нового предка) — **S**. (До этого **S** был «братом» **M**.) На рисунках ниже мы также будем использовать обозначение **P** для нового предка **N** (старого предка **M**), **SL** для левого потомка **S** и **SR** для правого потомка **S** (**S** не может быть листовым узлом, так как если **N** по нашему предположению является чёрным, то поддерево **P**, которое содержит **N**, чёрной высоты два и поэтому другое поддерево **P**, которое содержит **S** должно быть также чёрной высоты два, что не может быть в случае, когда **S** — лист).

*Примечание*: В некоторых случаях мы меняем роли и обозначения узлов, но в каждом случае любое обозначение продолжает означать тот же узел, что и в начале случая. Любые цвета, изображенные на рисунке либо предполагаются случаем, либо получается из других предположений. Белый означает неизвестный цвет (либо красный, либо чёрный).

Будем искать «брата», используя эту функцию:

**struct** **node** \*

sibling(**struct** **node** \*n)

{

**if** (n == n->parent->left)

**return** n->parent->right;

**else**

**return** n->parent->left;

}

*Примечание*: Для того, чтобы дерево оставалось верно определенным, нам нужно, чтобы каждый лист оставался листом после всех преобразований (чтобы у него не было потомков). Если удаляемый нами узел имеет нелистового потомка **N**, легко видеть, что свойство выполняется. С другой стороны, если **N** — лист, то, как можно увидеть из рисунков или кода, свойство также выполняется.

Мы можем выполнить действия, описанные выше, используя следующий код, где функция replace\_node ставит child на место узла n в дереве. Для удобства, код в этом разделе предполагает, что нулевые листья представлены реальными объектами узла, а не NULL (код вставки должен работать с таким же представлением).

void

replace\_node(node\* n, node\* child) {

child->parent = n->parent;

**if** (n == n->parent->left) {

n->parent->left = child;

} **else** {

n->parent->right = child;

}

}

void

delete\_one\_child(**struct** **node** \*n)

{

*/\**

*\* Условие: n имеет не более одного ненулевого потомка.*

*\*/*

**struct** **node** \*child = is\_leaf(n->right) ? n->left : n->right;

replace\_node(n, child);

**if** (n->color == BLACK) {

**if** (child->color == RED)

child->color = BLACK;

**else**

delete\_case1(child);

}

free(n);

}

*Примечание*: Если **N** является нулевым листом и мы не хотим представлять нулевые листы как реальные объекты, мы можем изменить алгоритм сначала вызывая delete\_case1() на его отца (узел, который мы удалили, n в коде выше) и удаляя его после этого. Мы можем сделать это потому, что отец чёрный, и поэтому ведет себя так же как нулевой лист (и иногда называется 'phantom' лист). Мы можем безопасно удалить его так как n останется листом после всех операций, как показано выше.

Если **N** и его текущий отец чёрные, тогда удаление отца приведет к тому, что пути, которые проходят через **N** будут иметь на один чёрный узел меньше, чем пути, которые не проходят через него. Так как это нарушает свойство 5 (все пути из любого узла к его листовым узлам содержат одинаковое количество чёрных узлов), дерево должно быть перебалансировано. Есть несколько случаев для рассмотрения:

**Случай 1:** **N** — новый корень. В этом случае, все сделано. Мы удалили один чёрный узел из каждого пути и новый корень является чёрным узлом, так что свойства сохранены.

void

delete\_case1(**struct** **node** \*n)

{

**if** (n->parent != NULL)

delete\_case2(n);

}

*Примечание*: В случаях 2, 5, и 6 мы предполагаем, что **N** является левым потомком своего предка **P**. Если он — правый потомок, *left* и *right* нужно поменять местами во всех трех случаях. Опять-таки, примеры кода принимают это во внимание.

|  |
| --- |
| [Диаграмма случая 2](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_2_as_svg.svg?uselang=ru)  **Случай 2:** **S** — красный. В этом случае мы меняем цвета **P** и **S**, и затем делаем [вращение](https://ru.wikipedia.org/w/index.php?title=Tree_rotation&action=edit&redlink=1) влево вокруг **P**, ставя **S** дедушкой **N**. Нужно заметить, что **P** должен быть чёрным, если он имеет красного потомка. Результирующее поддерево всё равно имеет черных узлов на единицу меньше, поэтому на этом мы ещё не закончили. Теперь **N** имеет чёрного брата и красного отца, поэтому мы можем перейти к шагу 4, 5 или 6. (Его новый брат является чёрным потому, что он был потомком красного **S**.)  Далее через **S** будет обозначен новый брат **N**. |

void delete\_case2(**struct** **node** \*n)

{

**struct** **node** \*s = sibling(n);

**if** (s->color == RED) {

n->parent->color = RED;

s->color = BLACK;

**if** (n == n->parent->left)

rotate\_left(n->parent);

**else**

rotate\_right(n->parent);

}

delete\_case3(n);

}

|  |
| --- |
| [Диаграмма случая 3](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_3.png?uselang=ru)  **Случай 3:** **P**, **S**, и дети **S'** — чёрные. В этом случае мы просто перекрашиваем **S** в красный. В результате все пути, проходящие через **S**, но не проходящие через **N**, имеют на один чёрный узел меньше. Так как удаление отца **N** приводит к тому, что все пути, проходящие через **N**, содержат на один чёрный узел меньше, то такие действия выравнивают баланс. Тем не менее, все проходящие через **P** пути теперь содержат на один чёрный узел меньше, чем пути, которые через **P** не проходят, поэтому свойство 5 (все пути из любой вершины к её листовым узлам содержат одинаковое количество чёрных узлов) все ещё нарушено. Чтобы это исправить, мы применяем процедуру перебалансировки к **P**, начиная со случая 1. |

void delete\_case3(**struct** **node** \*n)

{

**struct** **node** \*s = sibling(n);

**if** (

(n->parent->color == BLACK) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)

)

{

s->color = RED;

delete\_case1(n->parent);

} **else**

delete\_case4(n);

}

|  |
| --- |
| [Диаграмма случая 4](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_4.png?uselang=ru)  **Случай 4:** **S** и его дети — чёрные, но **P** — красный. В этом случае мы просто меняем цвета **S** и **P**. Это не влияет на количество чёрных узлов на путях, проходящих через **S**, но добавит один к числу чёрных узлов на путях, проходящих через **N**, восстанавливая тем самым влияние удаленного чёрного узла. |

void delete\_case4(**struct** **node** \*n)

{

**struct** **node** \*s = sibling(n);

**if** (

(n->parent->color == RED) &&

(s->color == BLACK) &&

(s->left->color == BLACK) &&

(s->right->color == BLACK)

)

{

s->color = RED;

n->parent->color = BLACK;

} **else**

delete\_case5(n);

}

|  |
| --- |
| [Диаграмма случая 5](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_5.png?uselang=ru)  **Случай 5:** **S** — чёрный, левый потомок **S** — красный, правый потомок **S** — чёрный, и **N** является левым потомков своего отца. В этом случае мы вращаем дерево вправо вокруг **S**. Таким образом левый потомок **S** становится его отцом и новым братом **N**. После этого мы меняем цвета у **S** и его нового отца. Все пути по прежнему содержат одинаковое количество чёрных узлов, но теперь у **N** есть чёрный брат с красным правым потомком, и мы переходим к случаю 6. Ни **N**, ни его отец не влияют на эту трансформацию. (Для случая 6 мы обозначим через **S** нового брата **N**.) |

void delete\_case5(**struct** **node** \*n)

{

**struct** **node** \*s = sibling(n);

**if** (s->color == BLACK) { */\* this if statement is trivial,*

*due to case 2 (even though case 2 changed the sibling to a sibling's child,*

*the sibling's child can't be red, since no red parent can have a red child). \*/*

*/\* the following statements just force the red to be on the left of the left of the parent,*

*or right of the right, so case six will rotate correctly. \*/*

**if** (

(n == n->parent->left) &&

(s->right->color == BLACK) &&

(s->left->color == RED)

)

{

*/\* this last test is trivial too due to cases 2-4. \*/*

s->color = RED;

s->left->color = BLACK;

rotate\_right(s);

} **else** **if** (

(n == n->parent->right) &&

(s->left->color == BLACK) &&

(s->right->color == RED)

)

{

*/\* this last test is trivial too due to cases 2-4. \*/*

s->color = RED;

s->right->color = BLACK;

rotate\_left(s);

}

}

delete\_case6(n);

}

|  |
| --- |
| [Диаграмма случая 6](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_6.png?uselang=ru)  **Случай 6:** **S** — чёрный, правый потомок **S** — красный, и **N** является левым потомком своего отца **P**. В этом случае мы вращаем дерево влево вокруг **P**, после чего **S** становится отцом **P** и своего правого потомка. Далее мы меняем местами цвета у **P** и **S** (**P** принимает цвет **S**, **S** принимает цвет **P**), и делаем правого потомка **S** чёрным. Поддерево по прежнему имеет тот же цвет корня, поэтому свойства 4 (Оба потомка каждого красного узла — чёрные) и 5 (все пути из любой вершины к её листовым узлам содержат одинаковое количество чёрных узлов) не нарушаются. Тем не менее, у **N** теперь появился дополнительный чёрный предок: либо **P** стал чёрным, или он был чёрным и **S** был добавлен в качестве чёрного дедушки. Таким образом, проходящие через **N** пути проходят через один дополнительный чёрный узел.  Между тем, если путь не проходит через **N**, то есть 2 возможных варианта:   * Он проходит через нового брата **N**. Тогда, он должен проходить через **S** и **P**, которые просто поменяли цвета и места. Поэтому путь содержит то же количество чёрных узлов. * Он проходит через нового дядю **N**, правого потомка **S**. Когда-то он проходил через **S**, отца **S** и правого потомка **S** (который был красным), но теперь он проходит только через **S**, который принял на себя цвет своего прежнего родителя, и правого потомка **S**, который был перекрашен из красного в чёрный (Предполагаем, что цвет **S**: чёрный). Эффект заключается в том, что этот путь проходит через такое же количество чёрных узлов.   В любом случае, число чёрных узлов на этих путях не изменится. Поэтому, мы восстановили свойства 4 (Оба потомка каждого красного узла — чёрные) и 5 (все пути из любой вершины к её листовым узлам содержат одинаковое количество чёрных узлов). Белый узел на диаграмме может быть как красным так и чёрным, но должен указывать на тот же цвет как в начале, так и в конце трансформации. |

void delete\_case6(**struct** **node** \*n)

{

**struct** **node** \*s = sibling(n);

s->color = n->parent->color;

n->parent->color = BLACK;

**if** (n == n->parent->left) {

s->right->color = BLACK;

rotate\_left(n->parent);

} **else** {

s->left->color = BLACK;

rotate\_right(n->parent);

}

}

Все рекурсивные вызовы функции [хвостовые](https://ru.wikipedia.org/wiki/%D0%A5%D0%B2%D0%BE%D1%81%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D1%80%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F) и преобразуются в циклы, так что алгоритм [требует памяти O(1)](https://en.wikipedia.org/wiki/in-place_algorithm). В алгоритме выше, все случаи связаны по очереди, кроме случая 3, где может произойти возврат к случаю 1, который применяется к предку узла: это единственный случай когда последовательная реализация будет эффективным циклом (после одного вращения в случае 3).

Так же, хвостовая рекурсия никогда не происходит на дочерних узлах, поэтому цикл хвостовой рекурсии может двигаться только от дочерних узлов к их последовательным родителям. Произойдет не более, чем O(log *n*) циклических возвратов к случаю 1 (где *n* — общее количество узлов в дереве до удаления). Если в случае 2 произойдет вращение (единственно возможное в цикле случаев 1-3), тогда отец узла **N** становится красным после вращения и мы выходим из цикла. Таким образом будет произведено не более одного вращения в течение этого цикла. После выхода из цикла произойдет не более двух дополнительных поворотов. А в целом произойдет не более трех поворотов дерева.

Объяснение моего кода.

Итерирование по Красно-черному дереву .

Модифицирование кода для обычного итерирования

Модифицирование кода для итерирования по списку