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

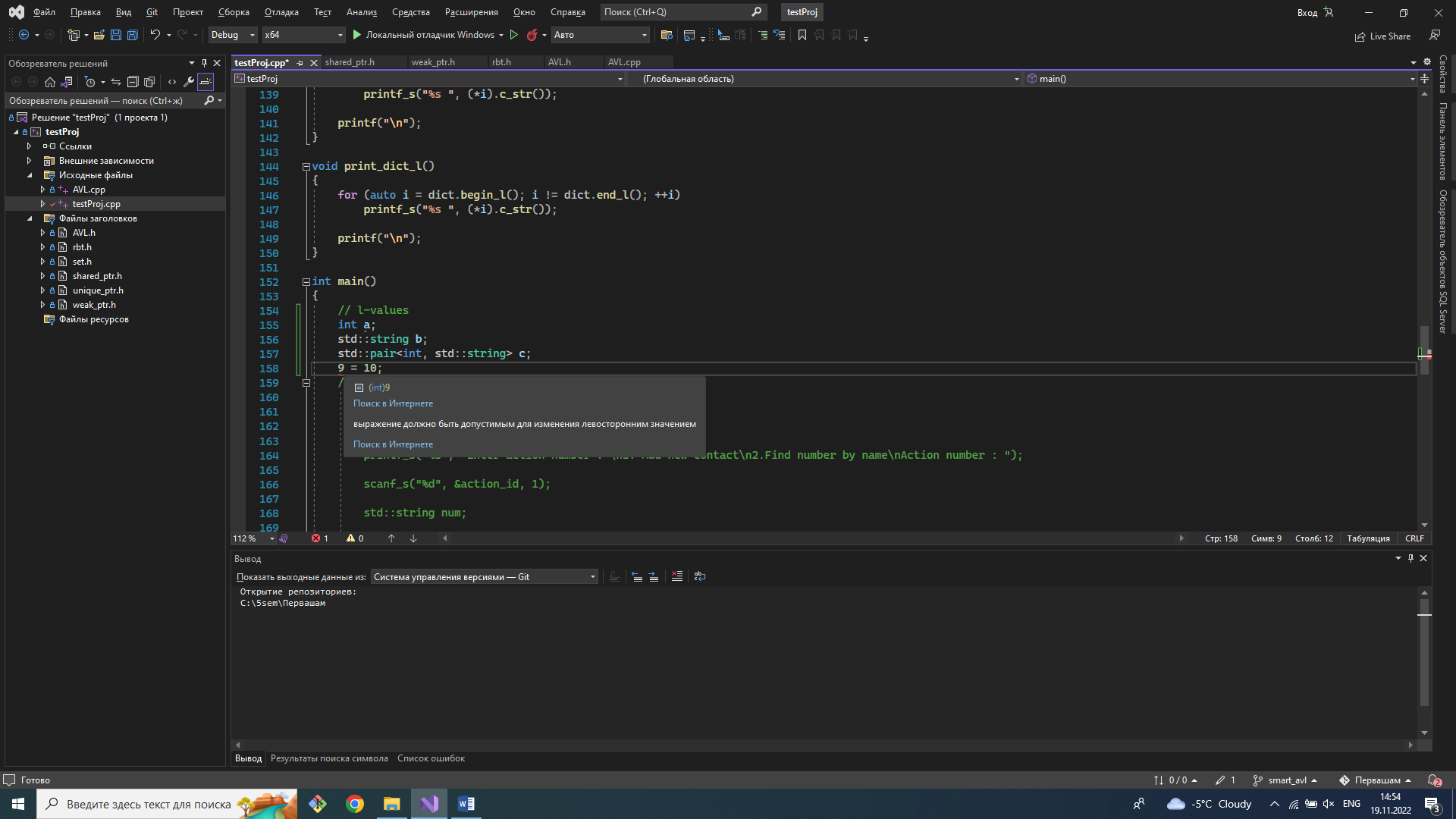
Способы обхода дерева

2-4 дерево и Красно-черное дерево

Итерирование по Красно-черному дереву .

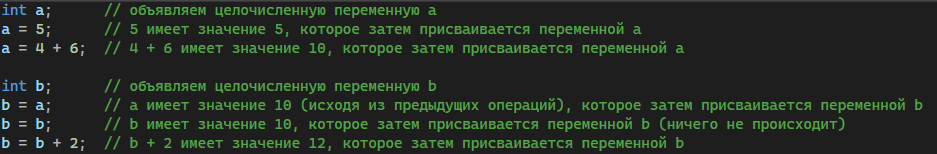
**l-values и r-values**

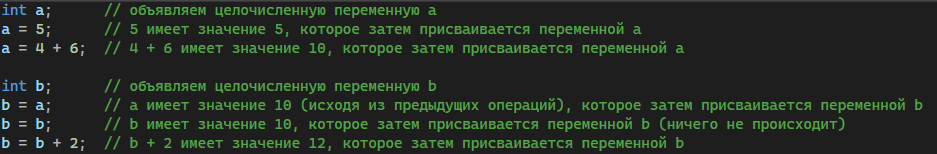
В языке C++ все переменные являются *l-values. l-value* — это значение, которое имеет свой собственный адрес в памяти. Поскольку все переменные имеют адреса, то они все являются *l-values* (например, переменные *a, b, c* — все они являются *l-values*). *l* от слова *«left»*, так как только значения *l-values* 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.****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*:

Давайте детально рассмотрим последнюю операцию присваивания:



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:



**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 =):



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



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.

**Если у вас изначально имеется значение для переменной, то лучше использовать 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!

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.

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

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*). *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 у объекта *побочные эффекты*.

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.

До версии C++11 существовал только один тип ссылок, 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:

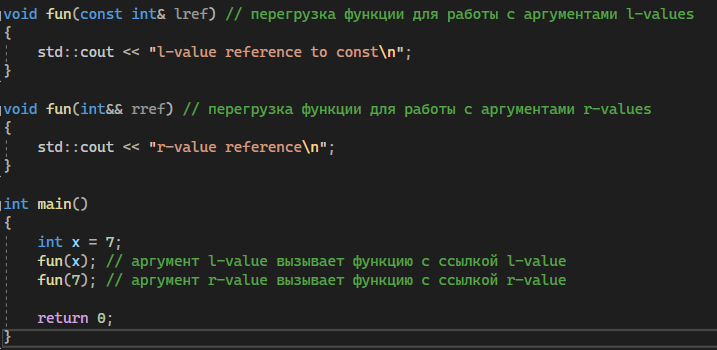


*Ссылки r-value* имеют **два полезных свойства:**

   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, 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 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, что может вызывать проблемы у классов, которые работают с динамически выделенной памятью. Одним из вариантов решения таких проблем является переопределение конструктора копирования и оператора присваивания копированием для выполнения ***глубокого копирования***.

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*.

## 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

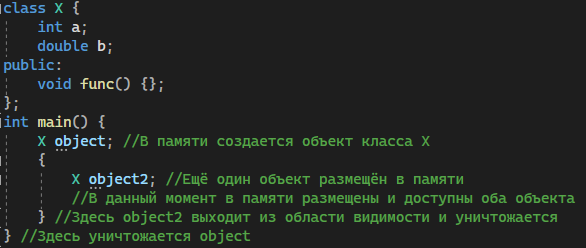
Как C++ управляет памятью?

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

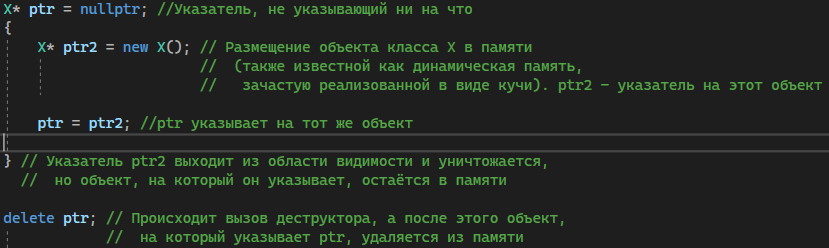
Когда мы говорим про управление памятью в C++, мы неизменно обращаемся к термину ***storage duration*** (длительность хранения). Storage duration – это свойство объекта, которое описывает, когда тот попадает в память и когда её освобождает.

В C++ существует четыре вида storage duration:

* **Автоматическая storage duration.** Когда управление входит в область видимости объекта (также известную как ***scope***), он размещается в автоматической памяти, зачастую реализованной в виде стека; когда управление покидает эту область, вызывается деструктор и память освобождается.



* **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*.



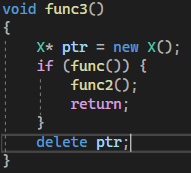
Можно сказать, что в случае с автоматической storage duration память освобождается автоматически, а в случае с динамической – вручную. Почему же тогда не использовать всегда автоматическую память?

* 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.



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:

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.

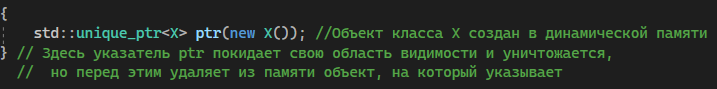
Все вышеназванные проблемы изящно решаются умными указателями. Умные указатели в C++ – это не что-то магическое, встроенное в синтаксис языка, а не более чем набор классов из стандартной библиотеки. Разберёмся с ними один за одним.

***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. Он ссылается на объект в динамической памяти и при выходе из области видимости уничтожает хранимый объект. Взглянем на пример кода ниже:



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.

## 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), 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 в тип bool, возвращая true, если std::unique\_ptr владеет ресурсом.

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:



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:



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

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

Так можно реализовать аналог класса std::unique\_ptr.

template<class T>

class unique\_ptr

{

T\* m\_ptr;

public:

operator bool() const

{

return m\_ptr != nullptr;

}

unique\_ptr(T\* ptr = nullptr)

:m\_ptr(ptr)

{

}

// Деструктор позаботится об удалении указателя

~unique\_ptr()

{

delete m\_ptr;

}

// Конструктор перемещения, который передает право собственности на x.m\_ptr в m\_ptr

unique\_ptr(unique\_ptr&& x)

: m\_ptr(x.m\_ptr)

{

x.m\_ptr = nullptr; // мы поговорим об этом чуть позже

}

unique\_ptr(const unique\_ptr& x) = delete;

// Оператор присваивания, который реализовывает семантику перемещения

unique\_ptr& operator=(unique\_ptr& a) = delete;

// Оператор присваивания перемещением, который передает право собственности на x.m\_ptr в m\_ptr

unique\_ptr& operator=(unique\_ptr&& x)

{

// Проверка на самоприсваивание

if (&x == this)

return \*this;

// Удаляем всё, что к этому моменту может хранить указатель

delete m\_ptr;

// Передаем право собственности на x.m\_ptr в m\_ptr

m\_ptr = x.m\_ptr;

x.m\_ptr = nullptr; // мы поговорим об этом чуть позже

return \*this;

}

T& operator\*() const { return \*m\_ptr; }

T\* operator->() const { return m\_ptr; }

bool isNull() const { return m\_ptr == nullptr; }

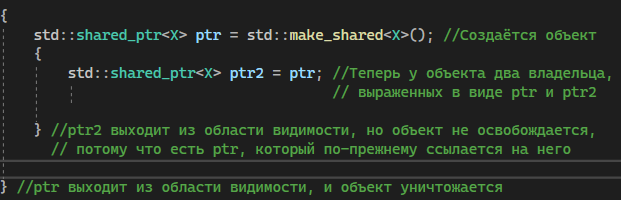
};

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

Shared\_ptr и 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. 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:



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.

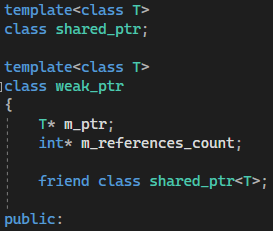
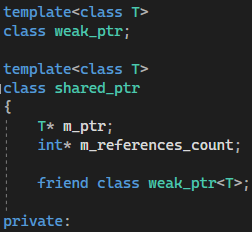
Чтобы разорвать цикличность, необходимо использовать std::weak\_ptr. Это фактически умный указатель non owning, предназначенный для использования именно с std::shared\_ptr. Копирование std::weak\_ptr не увеличивает счётчик в std::shared\_ptr, а значит и не защищает объект от уничтожения. При этом всегда имеется возможность проверить, существует ли ещё объект, на который ссылается std::weak\_ptr, или нет.

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

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

Для реализации аналога shared\_ptr, за основу можно взять класс unique\_ptr.

1. Необходимо добавить блок управления, а именно счетчик ссылок на выделенную память. К примеру: int\* m\_references\_count; Этот указатель далее можно передавать при копировании, инкрементируя его.
2. Можно снова разрешить конструкторы копирования, которые будут выполнять поверхностное копирование, а после инкрементировать счетчик.
3. При вызове деструктора, или при копировании поверх уже имеющихся данных, вызывается функция удаления, которая декрементирует счетчик, и удаляет значения и сам счетчик, если количество ссылок нулевое.
4. Теперь надо реализовать класс weak\_ptr и подружить наши классы. Для этого надо перед определением классов, добавить определение другого без реализации.

1. Класс weak\_ptr не имеет переопределенных операторов разыменовывания, а так же в нем не вызывается функция удаления, т.к он является не владеющим.
2. В обоих классах необходимо реализовать конструкторы копирования, и операторы присваивания копированием, принимающими в качестве аргумента своего друга.

shared\_ptr(weak\_ptr<T>& x);

shared\_ptr& operator=(weak\_ptr<T>& x)

weak\_ptr(shared\_ptr<T>& x)

weak\_ptr& operator=(shared\_ptr<T>& x)

1. Осталось реализовать метод lock() для weak\_ptr, который будет возвращать shared\_ptr, из которого уже можно обращаться к данным.

shared\_ptr<T> lock()

{

return shared\_ptr<T>(\*this);

}

1. Для удобства можно определить оператор приведения к типу bool, который определяет, не является ли указатель nullptr. Так же придется сделать метод сравнивающий два объекта, к примеру:



При дальнейшем использовании этих классов у меня возникла проблема с тем, что при попытке сравнить два объекта компилятор сначала преобразовывал оба объекта к типу bool, а потом сравнивало две единички, что вылилось в пол часа активного пошагового тестирования.

Что такое дерево?

Дерево — одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы.

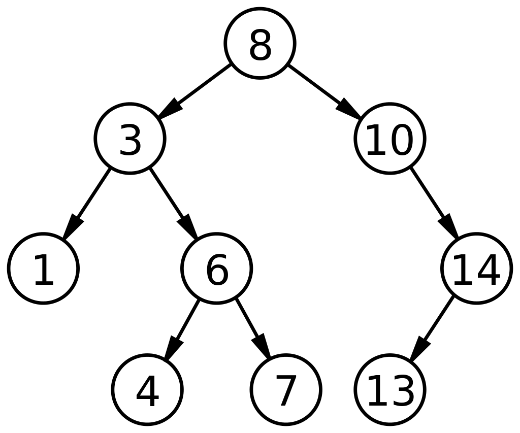
## Справочник терминов

* **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

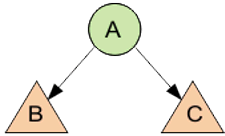
Теперь рассмотрим особый тип деревьев, называемых бинарными или двоичными деревьями.

“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.

Определение ноды

struct Node

{

Node(int nKey, std::string nValue);

Node();

shared\_ptr<Node> newNode(int nKey, std::string nValue);

shared\_ptr<Node> left, right;

int key;

std::string value;

};

Добавление ноды

shared\_ptr<::Node> insert(shared\_ptr<Node> node, int key, str value)

{

/\* 1. Perform the normal BST rotation \*/

if (!node)

return(newNode(key, value));

if (key < node->key)

node->left = insert(node->left, key, value);

else if (key > node->key)

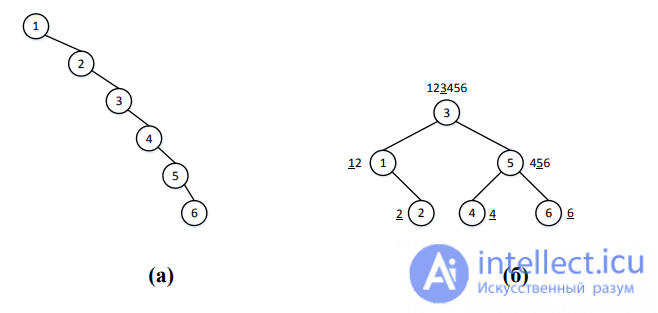
node->right = insert(node->right, key, value);

else // Equal keys not allowed

return node;

}

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).

**АВЛ-дерево**

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.

Реализация:

int getHeight(shared\_ptr<Node> node) // Высота дерева

{

if (!node)

return 0;

return node->height;

}

int max(int a, int b)

{

return (a > b) ? a : b;

}

// Правый поворот

shared\_ptr<Node> rightRotate(shared\_ptr<Node> node)

{

shared\_ptr<Node> x = node->left;

shared\_ptr<Node> T2 = x->right;

// Perform rotation

x->right = node;

node->left = T2;

// Update heights

node->height = max(getHeight(node->left),

getHeight(node->right)) + 1;

x->height = max(getHeight(x->left),

getHeight(x->right)) + 1;

// Return new root

return x;

}

// Левый поворот

shared\_ptr<Node> leftRotate(shared\_ptr<Node> node)

{

shared\_ptr<Node> y = node->right;

shared\_ptr<Node> T2 = y->left;

// Perform rotation

y->left = node;

node->right = T2;

// Update heights

node->height = max(getHeight(node->left),

getHeight(node->right)) + 1;

y->height = max(getHeight(y->left),

getHeight(y->right)) + 1;

// Return new root

return y;

}

// Разность высот поддеревьев

int getBalance(shared\_ptr<Node> node)

{

if (!node)

return 0;

return getHeight(node->left) -

getHeight(node->right);

}

// Вставка

shared\_ptr< Node> insert(shared\_ptr<Node> node, int key, str value)

{

/\* 1. Perform the normal BST rotation \*/

if (!node)

return(newNode(key, value));

if (key < node->key)

node->left = insert(node->left, key, value);

else if (key > node->key)

node->right = insert(node->right, key, value);

else // Equal keys not allowed

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(getHeight(node->left),

getHeight(node->right));

/\* 3. Get the balance factor of this

ancestor node to check whether

this node became unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced,

// then there are 4 cases

// Left Left Case

if (balance > 1 && key < node->left->key)

return rightRotate(node);

// Right Right Case

if (balance < -1 && key > node->right->key)

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node->left->key)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// Минимальное значение в поддереве с корнем node.

shared\_ptr< Node> minValueNode(shared\_ptr<Node> node)

{

shared\_ptr<Node> current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left)

current = current->left;

return current;

}

// Удаление узла

shared\_ptr< Node> deleteNode(shared\_ptr<Node> root, int key)

{

// STEP 1: PERFORM STANDARD BST DELETE

if (!root)

return root;

// If the key to be deleted is smaller

// than the root's key, then it lies

// in left subtree

if (key < root->key)

root->left = deleteNode(root->left, key);

// If the key to be deleted is greater

// than the root's key, then it lies

// in right subtree

else if (key > root->key)

root->right = deleteNode(root->right, key);

// if key is same as root's key, then

// This is the node to be deleted

else

{

// node with only one child or no child

if ((!root->left) ||

(!root->right))

{

shared\_ptr<Node> temp = root->left ?

root->left :

root->right;

// No child case

if (!temp)

{

temp = root;

root = shared\_ptr<Node>();

}

else // One child case

\*root = \*temp; // Copy the contents of

// the non-empty child

}

else

{

// node with two children: Get the inorder

// successor (smallest in the right subtree)

shared\_ptr<Node> temp = minValueNode(root->right);

// Copy the inorder successor's

// data to this node

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right,

temp->key);

}

}

// If the tree had only one node

// then return

if (!root)

return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

root->height = 1 + max(getHeight(root->left),

getHeight(root->right));

// STEP 3: GET THE BALANCE FACTOR OF

// THIS NODE (to check whether this

// node became unbalanced)

int balance = getBalance(root);

// If this node becomes unbalanced,

// then there are 4 cases

// Left Left Case

if (balance > 1 &&

getBalance(root->left) >= 0)

return rightRotate(root);

// Left Right Case

if (balance > 1 &&

getBalance(root->left) < 0)

{

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Right Case

if (balance < -1 &&

getBalance(root->right) <= 0)

return leftRotate(root);

// Right Left Case

if (balance < -1 &&

getBalance(root->right) > 0)

{

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

2-3 дерево и Красно-черное дерево

### **Два-три дерево**

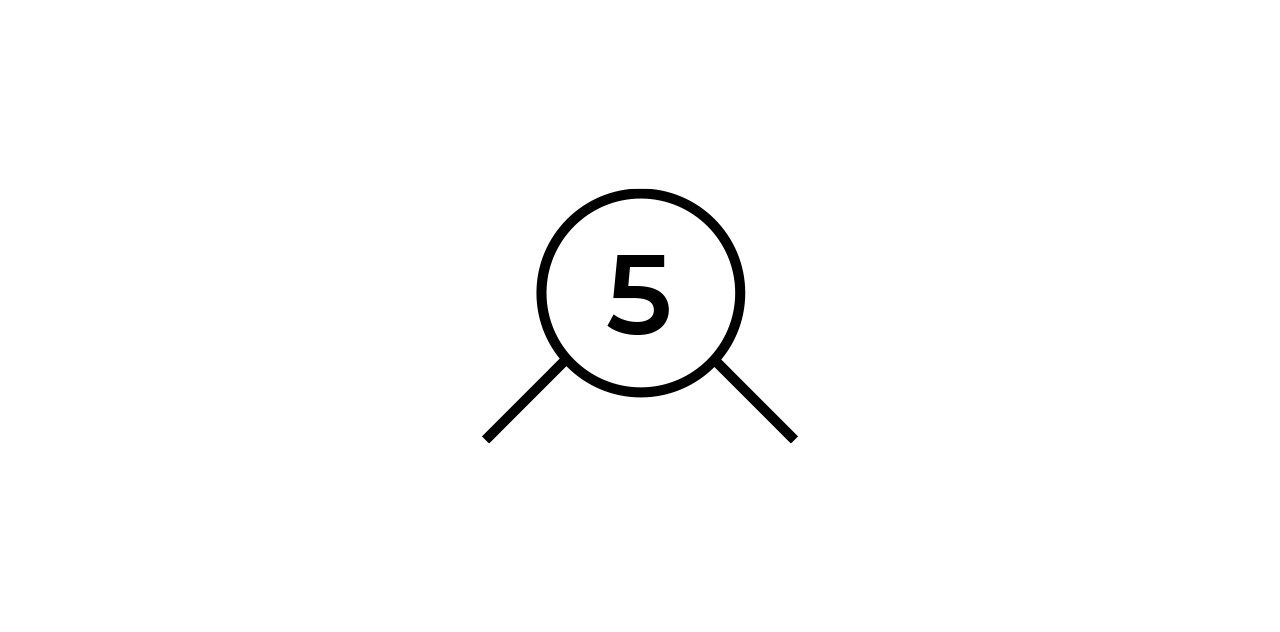
Чтобы понять красно-черное дерево, нужно понять два-три дерево

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.

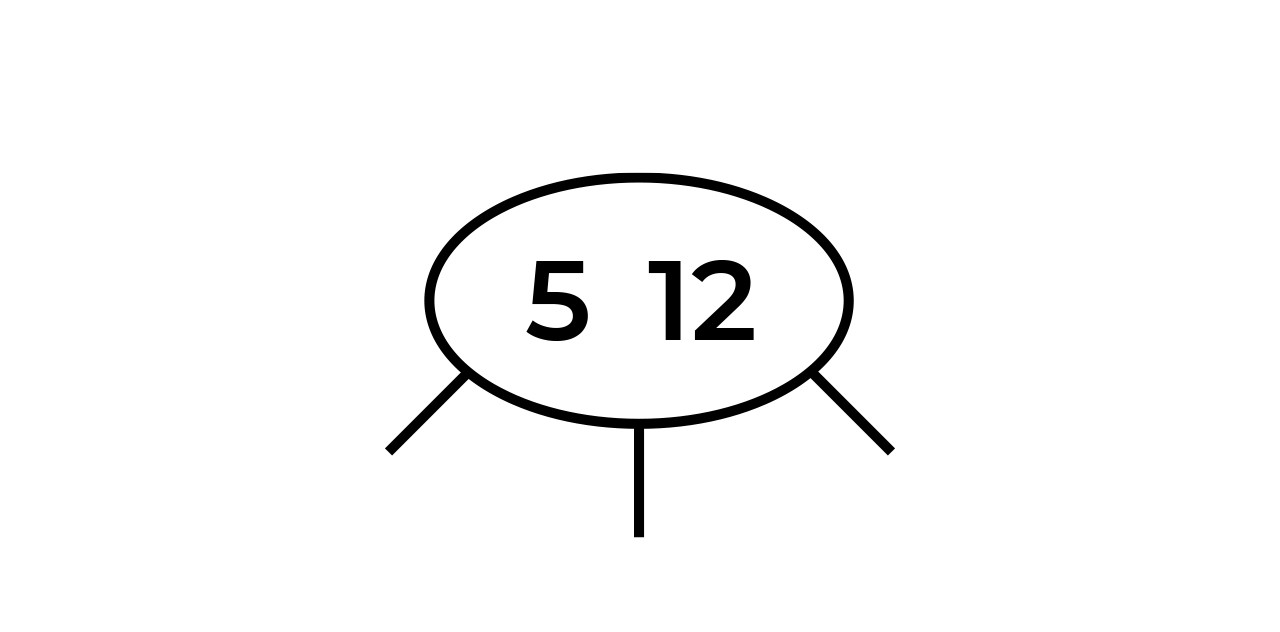
**Два-три дерево** - абстрактный тип данных, напоминающий по структуре дерево. В нодах два-три дерева может быть одно или два значения и два или три потомка (от чего зависит количество значений и потомков ноды, узнаем ниже). Ноду с одним значением и двумя потомками будем называть 2-нода, ноду с двумя 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.

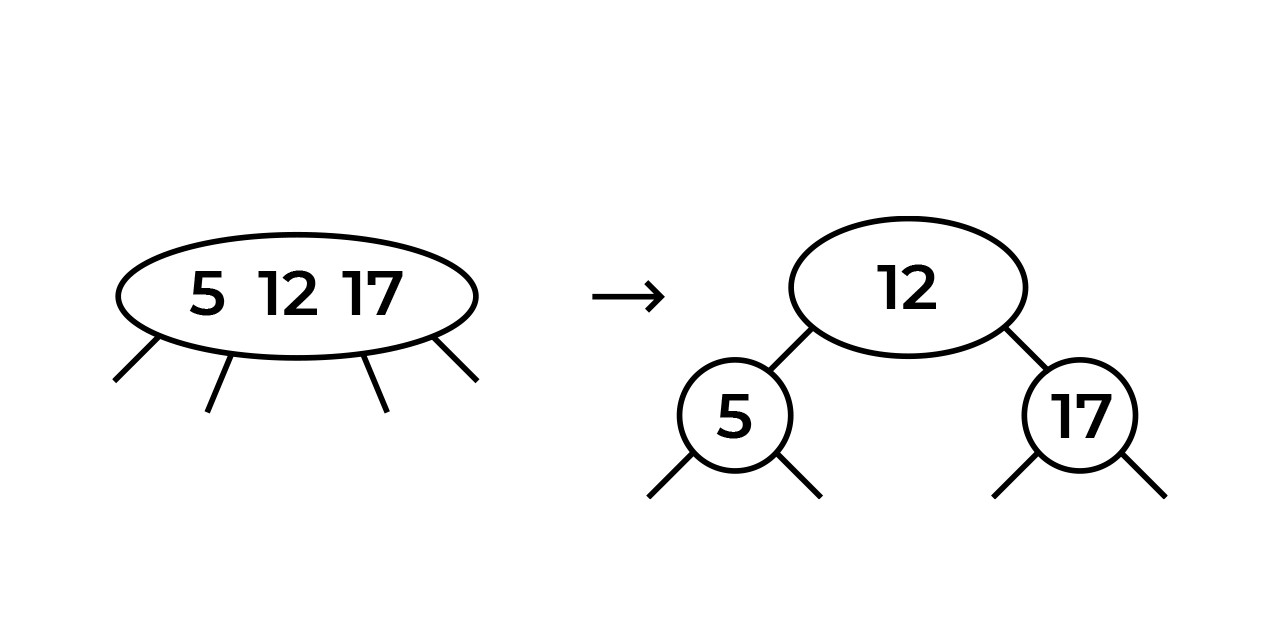


Добавим число 12. Число 12 мы так же добавляем в корень (помним, что нода может иметь два значения), но теперь нам нужно "отсортировать" нашу ноду (сортировать два элемента, ха), т.е. уложить их в порядке возрастания. В итоге получается нода 5-12.



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.

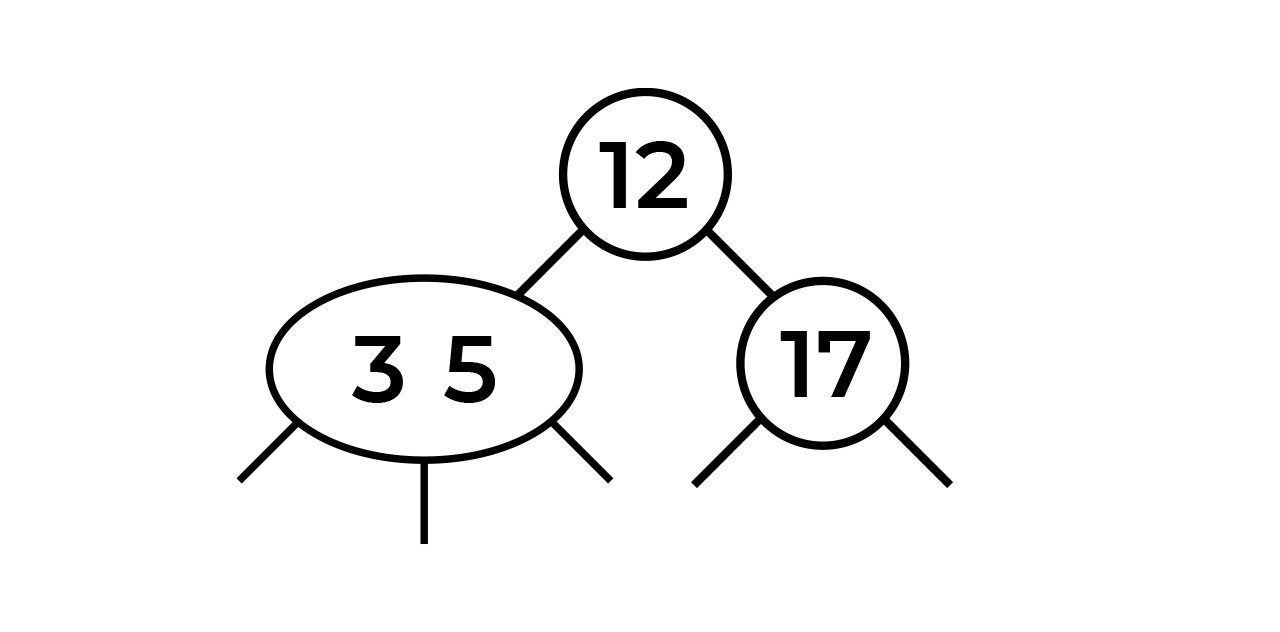


То, что мы сделали выше, можно назвать балансировкой два-три дерева. Правило в этой балансировке простое: если в ноде оказывается три значения, среднее значение мы "просачиваем" вверх. Алгоритм действий зависит от 3 условий:

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.

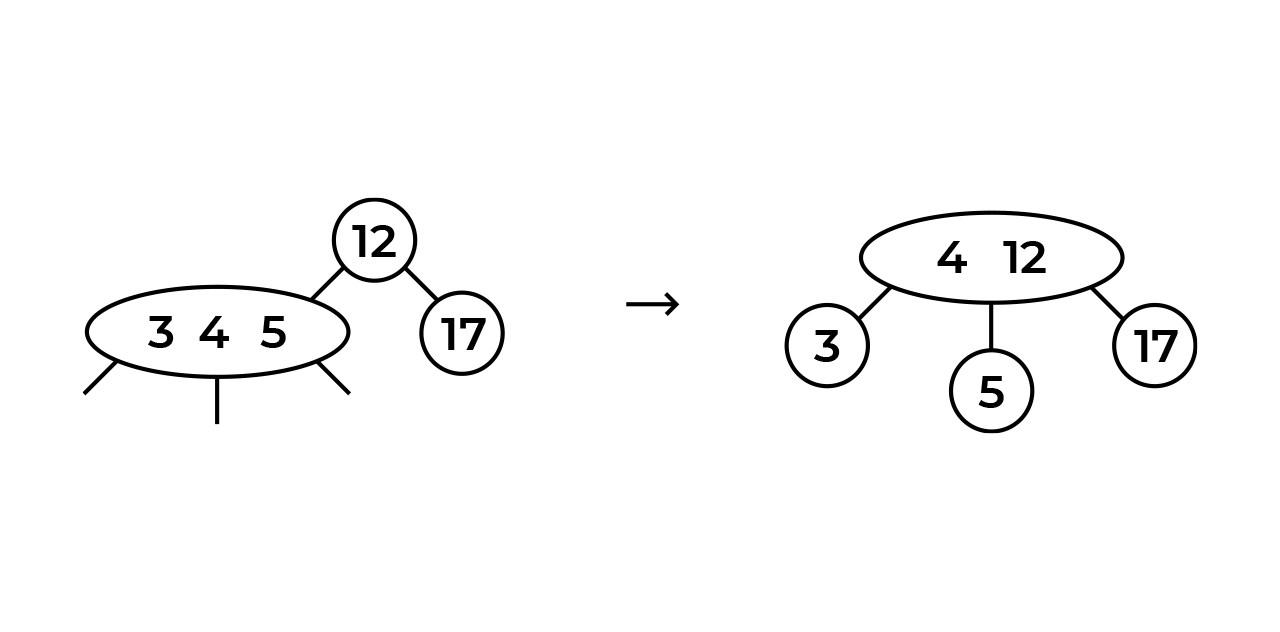
Окей, идем дальше. Давайте добавим число 3. Так как теперь мы не ограничиваемся одной нодой, спускаемся вниз. Понятно, что спуститься надо влево и добавить 3 к ноде со значением 5. Не забываем расставить 3 и 5 в нужном порядке. В итоге получилась нода 3-5.



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.

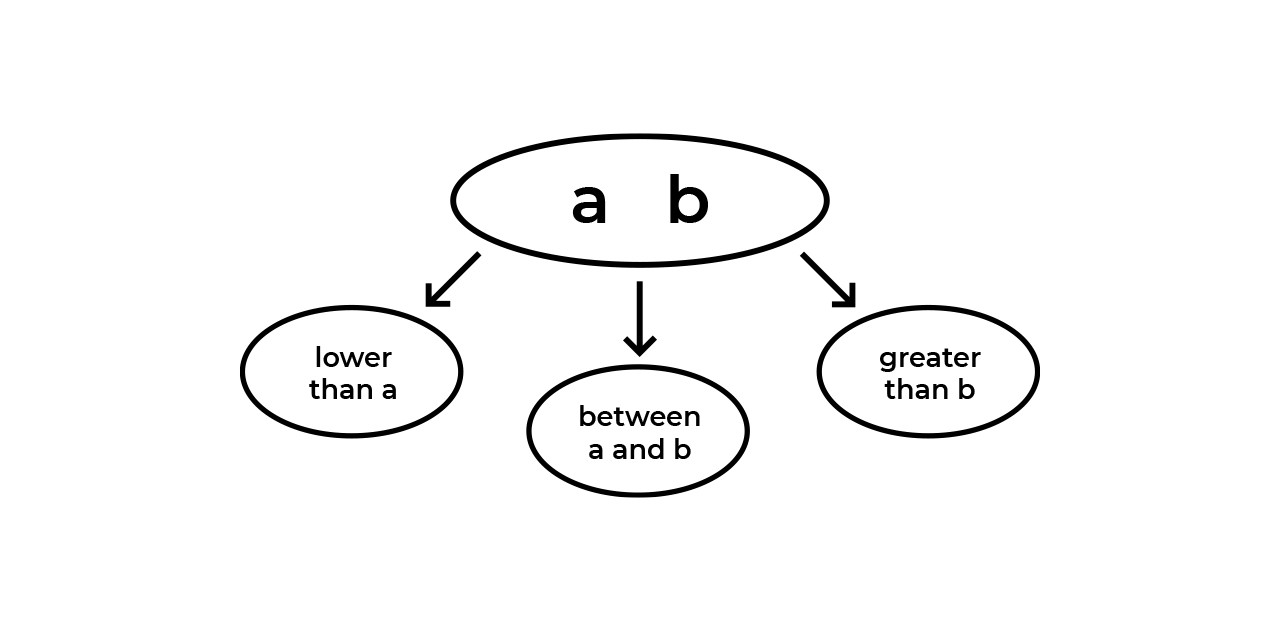
Теперь самое интересное. Помимо того, что мы добавим 4 к корню, мы так же добавим корню третьего потомка - это будет нода, которая была больше 4. В нашем случае это 5. Картина будет выглядеть так:



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?).

*В 2-3 дереве, при вставке, мы рекурсивно возвращаемся назад и разделяем ноды с 3 значениями, например, одна из последних нод (3, 4, 5). Эту структуру можно улучшить, не разделяя ноды с 3 значениями после вставки. Вместо этого, мы позволим 4-нодам находиться в нашем дереве, и разъединять их будем только по мере необходимости, а именно только при спуске от корня к листу во время вставки.*

*Такое дерево называется 2-4 деревом.*

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 языка C++[3], класс TreeMap языка Java[4], так же, как и многие другие реализации ассоциативного массива 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, в котором каждый узел имеет атрибут цвета. При этом:

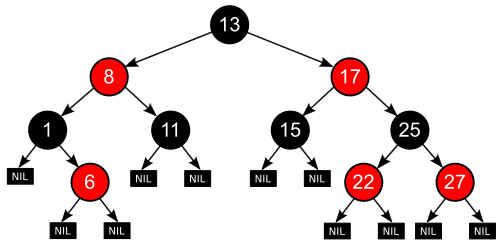
1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла —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

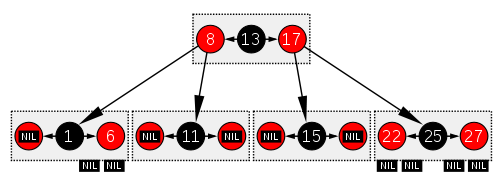
Благодаря этим ограничениям путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более 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 *2-4 дерево, модифицированное 2-3 дерево*, 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.

Определение структуры узла может выглядеть следующим образом:

struct Node

{

Node(T nKey, T1 nValue);

Node();

shared\_ptr<Node> left, right;

weak\_ptr<Node> parent; // weak\_ptr во избежания циклических ссылок

bool color; // red - false, black - true

T key;

T1 data;

};

Структура же самого класса дерева может выглядеть так:

template<class T, class T1>

class RBT

{

public:

RBT();

~RBT();

void add(T key, T1 value);

void remove(T key);

T1 findValueWithKey(T key);

bool contains(T key);

T1& operator[](T key);

void clear();

protected:

struct Node; // определение выше.

shared\_ptr<Node> grandparent(shared\_ptr<Node> node);

shared\_ptr<Node> uncle(shared\_ptr<Node> node);

shared\_ptr<Node> sibling(shared\_ptr<Node> node);

void RotateLeft(shared\_ptr<Node> node);

void RotateRight(shared\_ptr<Node> node);

void InsCase1(shared\_ptr<Node> node);

void InsCase2(shared\_ptr<Node> node);

void InsCase3(shared\_ptr<Node> node);

void InsCase4(shared\_ptr<Node> node);

void InsCase5(shared\_ptr<Node> node);

void DelCase1(shared\_ptr<Node> node);

void DelCase2(shared\_ptr<Node> node);

void DelCase3(shared\_ptr<Node> node);

void DelCase4(shared\_ptr<Node> node);

void DelCase5(shared\_ptr<Node> node);

void DelCase6(shared\_ptr<Node> node);

shared\_ptr<Node> find(shared\_ptr<Node> node, T key);

shared\_ptr<Node> min(shared\_ptr<Node> node);

shared\_ptr<Node> max(shared\_ptr<Node> node);

protected:

shared\_ptr<Node> root;

}

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

template<class T, class T1>

shared\_ptr<typename RBT<T, T1>::Node> RBT<T, T1>::grandparent(shared\_ptr<Node> node)

{

shared\_ptr<Node> p = node->parent; // Это присваивание делается для того, что бы не вызывать постоянно weak\_ptr<T>::lock(), для конвертации в shared\_ptr

if (p && p->parent)

return p->parent;

else

return shared\_ptr<Node>();

}

template<class T, class T1>

shared\_ptr<typename RBT<T, T1>::Node> RBT<T, T1>::uncle(shared\_ptr<Node> node)

{

shared\_ptr<Node> tmp = grandparent(node);

if (tmp)

{

if (tmp->left.equals(node->parent))

return tmp->right;

else

return tmp->left;

}

else

return shared\_ptr<Node>();

}

Левый и правый поворот дерева может быть выполнен так:

template<class T, class T1> // Так-же, как и в АВЛ-дереве

void RBT<T, T1>::RotateLeft(shared\_ptr<Node> node)

{

shared\_ptr<Node> right = node->right;

right->parent = node->parent;

if (node->parent)

{

shared\_ptr<Node> p = node->parent;

if (p->left.equals(node))

p->left = right;

else

p->right = right;

}

else

root = right;

node->right = right->left;

if (right->left)

right->left->parent = node;

node->parent = righ;

right->left = node;

}

template<class T, class T1> // Так-же, как и в АВЛ-дереве

void RBT<T, T1>::RotateRight(shared\_ptr<Node> node)

{

shared\_ptr<Node> left = node->left;

left->parent = node->parent;

if (node->parent)

{

shared\_ptr<Node> p = node->parent;

if (p->left.equals(node))

p->left = left;

else

p->right = left;

}

else

root = left;

node->left = left->right;

if (left->right)

left->right->parent = node;

node->parent = left;

left->right = node;

}

Вставка происходит, так, как это делается и в обычном дереве поиска, но с последующей балансировкой. Сначала мы спускаемся по дереву, запоминая предка, а далее, вставляем новый красный узел и рассматриваем возможные нарушения свойств красно-черного дерева.

template<class T, class T1>

void RBT<T, T1>::add(T key, T1 value)

{

shared\_ptr<Node> tmp = root, p = shared\_ptr<Node>();

while (tmp) // Спускаемся по дереву, ищя, куда вставить новый узел

{

p = tmp;

if (tmp->key > key)

tmp = tmp->left;

else if (tmp->key < key)

tmp = tmp->right;

else

return;// Если такой узел уже есть, то прото выходим

}

tmp = shared\_ptr<Node>(new Node(key, value));

tmp->parent = p;

if (p)

{

if (tmp->key > p->key)

p->right = tmp;

else

p->left = tmp;

}

else

root = tmp;

InsCase1(tmp);

return;

}

**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.

template<class T, class T1>

void RBT<T, T1>::InsCase1(shared\_ptr<Node> node)

{

if (!node->parent)

node->color = 1;

else

InsCase2(node);

}

**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.

template<class T, class T1>

void RBT<T, T1>::InsCase2(shared\_ptr<Node> node)

{

if (node->parent.lock()->color)

return;

else InsCase3(node);

}

*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. |

template<class T, class T1>

void RBT<T, T1>::InsCase3(shared\_ptr<Node> node)

{

shared\_ptr<Node> u = uncle(node);

if (u && !u->color)

{

node->parent.lock()->color = 1;

u->color = 1;

shared\_ptr<Node> g = grandparent(node);

g->color = 0;

InsCase1(g);

}

else

InsCase4(node);

}

*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. | |
| [Схема случая 5](https://commons.wikimedia.org/wiki/File:Red-black_tree_insert_case_5.png?uselang=ru)  template<class T, class T1>  void RBT<T, T1>::InsCase4(shared\_ptr<Node> node)  {  shared\_ptr<Node> g = grandparent(node), tmp = node;  shared\_ptr<Node> p = node->parent;  if (node.equals(p->right) && p.equals(g->left))  {  RotateLeft(p);  tmp = node->left;  }  else if (node.equals(p->left) && p.equals(g->right))  {  RotateRight(p);  tmp = node->right;  }  InsCase5(tmp);  }  **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. |

template<class T, class T1>

void RBT<T, T1>::InsCase5(shared\_ptr<Node> node)

{

shared\_ptr<Node> g = grandparent(node);

shared\_ptr<Node> p = node->parent;

p->color = 1;

g->color = 0;

if (node.equals(p->left))

RotateRight(g);

else

RotateLeft(g);

### }

### Удаление

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.

***В эту часть с дядями, тетями и остальной родней можно сильно не вникать, и быстро пробежав переходить к случаям с картинками, потому что запутаться тут довольно просто. После можете обращаться сюда, если совсем не будете понимать рассматриваемые случаи.***

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:

template<class T, class T1>

shared\_ptr<typename RBT<T, T1>::Node> RBT<T, T1>::sibling(shared\_ptr<Node> node)

{

if (!node->parent) return shared\_ptr<Node>();

shared\_ptr<Node> p = node->parent;

if (node.equals(p->left))

return p->right;

else

return p->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.

template<class T, class T1>

void RBT<T, T1>::remove(T key)

{

shared\_ptr<Node> tmp = root ? find(root, key) : shared\_ptr<Node>(); // Ищем узел с нужным ключем

if (!tmp) // Повезло повезло, ниче делать не надо если такого узла нет

{

return;

}

if (!tmp->parent && !tmp->left && !tmp->right) // Случай, когда дерево состоит из одного узла

{

root = shared\_ptr<Node>(); // Обнуляем корень

return;

}

if (tmp->left && tmp->right) // Если у узла есть и левй и правый потомок

{

// Ищем самый правый узел в левом поддереве

// Этот узел будет ближайшим меньшим к удаляемому, по значнию ключа

// Что дает нам возможность переместить его значение в удаляемый узел, и удалять уже новый, но у которого будет либо один, либо ноль потомков

// При таком способе прийдется рассматривать намного меньшее количество вариантов

shared\_ptr<Node> removed = tmp->left;

while (removed->right)

removed = removed->right;

tmp->data = removed->data;

tmp->key = removed->key;

tmp = removed;

}

shared\_ptr<Node> child = tmp->left ? tmp->left : tmp->right;

// Ищем потомка, если таково имеется

if (!child) // Случай, когда потомков нет

{

if (tmp->color)

// Удаление красного узла не привело бы к разбалансировке дерева

DelCase1(tmp); // А вот удаление черного узла нарушает свойства красно-черного дерева об одинаковой длинне черных путей

shared\_ptr<Node> p = tmp->parent;

if (p->left.equals(tmp)) // Меняем связи, тем самым удаляя узел, после чего тот выйдет из области видимости

p->left = child;

else

p->right = child;

}

else // Случай, коогда потомок есть

{

child->parent = tmp->parent; // Меняем связи

shared\_ptr<Node> p = tmp->parent;

if (tmp->parent) // Меняем связи

{

if (tmp.equals(p->left))

p->left = child;

else

p->right = child;

}

else

root = child; // Если надо, меняем корень дерева

if (tmp->color) // Если удаляемы узел был красным, то никакие свойства не нарушаются

{

if (!child->color) // Если удаляемый узел был черным, а потомок красный, то можем сохранить свойстав дерева, перекрасив потомка в черныйё

child->color = 1;

else

DelCase1(child);

}

}

}

*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.

template<class T, class T1>

void RBT<T, T1>::DelCase1(shared\_ptr<Node> node)

{

if (node->parent)

DelCase2(node);

}

*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 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**.  template<class T, class T1>  void RBT<T, T1>::DelCase2(shared\_ptr<Node> node)  {  shared\_ptr<Node> s = sibling(node);  shared\_ptr<Node> p = node->parent;  bool scol = s ? s->color : 1;  if (!scol)  {  p->color = 0;  s->color = 1;  if (node.equals(p->left))  RotateLeft(p);  else  RotateRight(p);  }  DelCase3(node);  } | | | | |
| [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.  template<class T, class T1>  void RBT<T, T1>::DelCase3(shared\_ptr<Node> node)  {  shared\_ptr<Node> s = sibling(node);  shared\_ptr<Node> p = node->parent;  bool scol = s ? s->color : 1;  bool lcol = s && s->left ? s->left->color : 1;  bool rcol = s && s->right ? s->right->color : 1;  if (p->color && scol && lcol && rcol)  {  if (s) s->color = 0;  DelCase1(p);  }  else  DelCase4(node);  } | | | |
| [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.  template<class T, class T1>  void RBT<T, T1>::DelCase4(shared\_ptr<Node> node)  {  shared\_ptr<Node> s = sibling(node);  shared\_ptr<Node> p = node->parent;  bool scol = s ? s->color : 1;  bool lcol = s && s->left ? s->left->color : 1;  bool rcol = s && s->right ? s->right->color : 1;  if (!p->color && scol && lcol && rcol)  {  if (s) s->color = 0;  p->color = 1;  }  else  DelCase5(node);  } | | |
| [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**.) | |
| [Диаграмма случая 6](https://commons.wikimedia.org/wiki/File:Red-black_tree_delete_case_6.png?uselang=ru)  template<class T, class T1>  void RBT<T, T1>::DelCase5(shared\_ptr<Node> node)  {  shared\_ptr<Node> s = sibling(node);  shared\_ptr<Node> p = node->parent;  bool lcol = s->left ? s->left->color : 1;  bool rcol = s->right ? s->right->color : 1;  if (s->color)  {  if (node.equals(p->left) &&  rcol &&  !lcol)  {  s->color = 0;  s->left->color = 1;  RotateRight(s);  }  else if (node.equals(p->right) &&  lcol &&  !rcol)  {  s->color = 0;  s->right->color = 1;  RotateLeft(s);  }  }  DelCase6(node);  }  **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.  template<class T, class T1>  void RBT<T, T1>::DelCase6(shared\_ptr<Node> node)  {  shared\_ptr<Node> s = sibling(node);  shared\_ptr<Node> p = node->parent;  s->color = p->color;  p->color = 1;  if (node.equals(p->left))  {  if (s->right) s->right->color = 1;  RotateLeft(p);  }  else  {  if (s->left) s->left->color = 1;  RotateRight(p);  }  } |

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.

Итерирование по Красно-черному дереву

Для итерирования добавим класс итератор, хранящий узел, а для узла реализуем два метода для поиска ближайшего меньшего, и ближайшего большего.

shared\_ptr<Node> next(shared\_ptr<Node> node);

shared\_ptr<Node> prev(shared\_ptr<Node> node);

class iterator

{

public:

iterator(shared\_ptr<Node> node, RBT<T, T1>\* own);

iterator(const iterator& it);

bool operator!=(iterator const& other) const;

bool operator==(iterator const& other) const;

T1& operator\*() const;

iterator& operator++();

iterator& operator--();

private:

shared\_ptr<Node> iter;

RBT<T, T1>\* owner;

};

template<class T, class T1>

typename RBT<T, T1>::iterator& RBT<T, T1>::iterator::operator++()

{

iter = owner->next(iter);

return \*this;

}

template<class T, class T1>

typename RBT<T, T1>::iterator& RBT<T, T1>::iterator::operator--()

{

iter = owner->prev(iter);

return \*this;

}

Этого достаточно для итерирования, но как мы помним, итераторы мы в основном используем, перебирая все элементы нашего дерева, что жвляется очень удобной функцией.

Для этого нам надо реализовать последний фантомный элемент, который будет возвращаться нам, если вызвать метод next() для самого большого элемента(метод prev() для самого маленького).

Тут есть два стула. На одном надо добавить параметр приоритета в узел, который у всех будет 1, у фантомного элемента слева, 0, а у фантомного элемента справа 2. Теперь ключи будут сравниваться сначала по этим приоритетам, а потом уже по ключам, для этого надо будет много где менять код, но в принципе, схема рабочая, и даже в стандартной библиотеке, как мне известно оно реализовано именно так.

*Так же, реализовать это будет не сложно, так что советую именно этот вариант.*

Второй вариант, который пришел мне в голову при огромном нежелании садиться на стул, это добавить родительский элемент к корню, но не добавлять этому родительскому элементу никаких потомков.

Почему это работает?

Для того, чтобы это понять, надо посмотреть, что будет происходить при вызове next() из крайнего правого узла:

template<class T, class T1>

shared\_ptr<typename RBT<T, T1>::Node> RBT<T, T1>::next(shared\_ptr<Node> node)

{

shared\_ptr<Node> tmp = node->right; // tmp == nullptr, т.к. у крайнего правого

// узла нет правого потомка

if (tmp) // это мимо, т.к. tmp == nullptr

{

while (tmp->left) tmp = tmp->left;

return tmp;

}

tmp = node; // теперь tmp это текущий узел.

while (tmp->parent && tmp.equals(tmp->parent.lock()->right)) tmp = tmp->parent;

// В цикле выше мы доходим до корня, т.к у нашего фантомного узла нулевые

// потомки

return tmp->parent.lock(); // Вовзвращаем родителя, в данном случае, фантомный

// элемент

}

Теперь получаем, что при крайних вызовах следующим элементом является наш фантомный узел, что позволит нам написать заветную строку;

for (auto j = rbt.begin(); j != rbt.end(); ++j)

И перебрать все элементы дерева. Функция begin() будет возвращать итератор с крайним левым элементом, а функция end(), итератор с нашим фантомным элементом. Инициализировать фантомный элемент можно рядом с корнем.

Но, даже в этом случае, придется добавить немного кода. Во многих наших методах у нас есть проверка на наличие родителя, ведь именно так мы отличаем корень от остальных узлов.

Да, можно добавить этот код, но есть и второй возможной вариант, которым я воспользовался. Каждый раз, когда я собираюсь добавлять или удалять элемент, я отсоединяю этот фантомный элемент от корня, а в конце функции обратно присоединяю.

if (root) root->parent = shared\_ptr<Node>(); // отсоединяем

if (root) root->parent = \_end; //присоединяем

Так же интересным способом итерирования является другой тип итератора. В этом итераторе , каждый узел так же является узлов двусвязного списка. Это позволит проитерироваться по всему дереву за время O(n), в то время, как первый вид итератора делал это за O(n\*log(n)).