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.

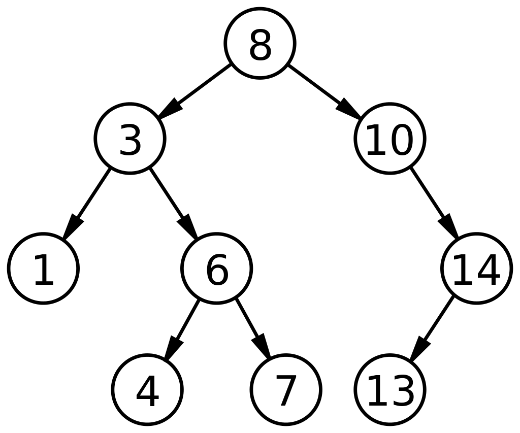
## Справочник терминов

* **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

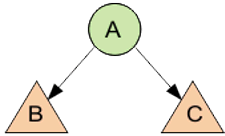
Теперь рассмотрим особый тип деревьев, называемых бинарными или двоичными деревьями.

“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;  
  }  
}

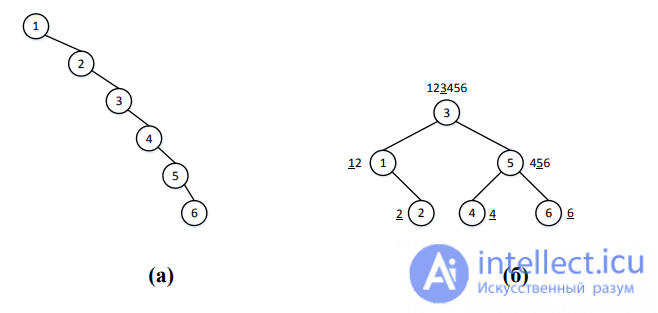
//////////////////////////////////////

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

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

2-4 дерево и Красно-черное дерево

Итерирование по Красно-черному дереву .