Константные экземпляры классов могут вызывать только константные методы

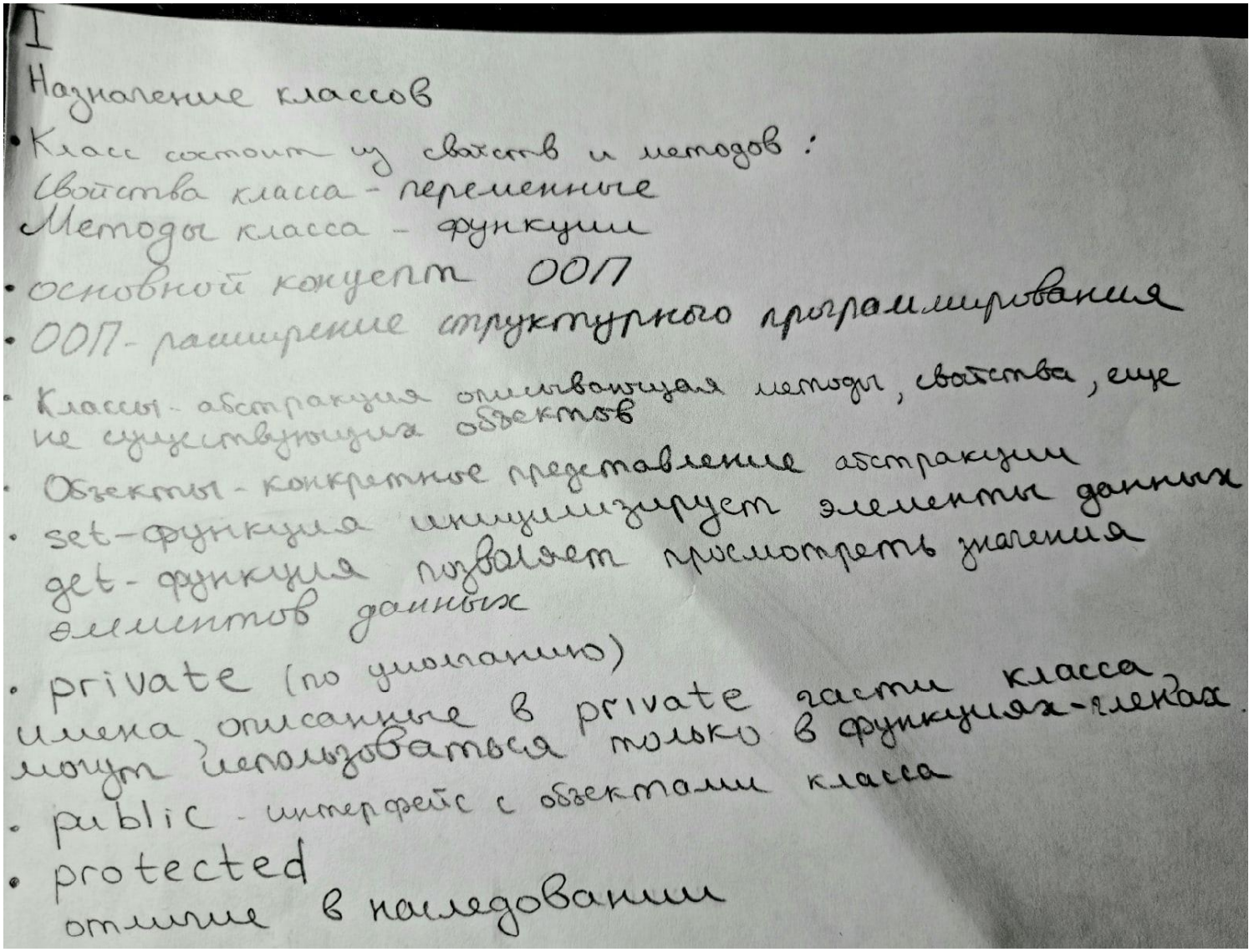
Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую вы хотите сделать дружественной классу.

```
int value = 7;

const int *ptr1 = &value;

 int *const ptr2 = &value;

const int *const ptr3 = &value;
```

```
std::vector<int> C = {1, 2 ,3 ,4, 5};

std::vector<int>::iterator a = C.begin();
std::cout << *(++a);


std::vector<int> merge(const std::vector<int> &a, const std::vector<int> &b){
std::vector<int> c;
c.insert(c.begin(), a.begin(), a.end());
c.insert(c.end(), b.begin(), b.end());
return c;
}

int main() {
std::vector<int> a = { 1 , 2 , 3};
std::vector<int> b = { 9 , 8 , 6};

// std::vector<int> c = merge(a, b);
std::vector<int> c(merge(a, b));
for (auto elem : c) {
std::cout « elem « ", ";
}
return 0;
}
```

Билет №1 Назначение классов. Члены класса и доступ к ним (public, private, protected).

Основные задачи **ООП** — структурировать код, повысить его читабельность и ускорить понимание логики **программы**. Косвенно выполняются и другие задачи: например, повышается безопасность кода и сокращается его дублирование.



Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом. Интерфейсом класса являются заголовки его методов

Билет №2 Конструктор, деструктор

Конструктор — специальная функция, которая выполняет начальную инициализацию элементов данных, причем имя конструктора обязательно должно совпадать с именем класса. Важным отличием конструктора от остальных функций является то, что он не возвращает значений.

В любом классе должен быть конструктор, даже если явным образом конструктор не объявлен.

Деструктор — это специальный тип метода класса, который выполняется при удалении объекта класса.

Когда объект автоматически выходит из области видимости или динамически выделенный объект явно удаляется с помощью ключевого слова delete, вызывается деструктор класса (если он существует) для выполнения необходимой очистки до того, как объект будет удален из памяти.

Однако, если объект класса содержит любые ресурсы (например, динамически выделенную память или файл/базу данных), или, если вам необходимо выполнить какие-либо действия до того, как объект будет уничтожен.

деструктор должен иметь тоже имя, что и класс, со знаком тильда (~) в самом начале;

деструктор не может принимать аргументы;

деструктор не имеет типа возврата.

Билет №3 Указатель this. Статические компоненты класса.

Статические переменные

Статические члены принадлежат классу, а не объектам этого класса.

Статические переменные и методы относят в целом ко всему классу. Для их определения используется ключевое слово *static*.

Когда мы объявляем статическую переменную-член внутри тела класса, то мы сообщаем компилятору о существовании статической переменной-члене, но не о её определении. Поскольку статические переменные-члены не являются частью отдельных объектов класса (они обрабатываются аналогично глобальным переменным и инициализируются при запуске программы), то вы должны явно определить статический член вне тела класса — в глобальной области видимости.

Для объявления переменной типа *static* необходимо проинициализировать (тип “класс”:: C = …, C - статическая переменная) ее в глобальной области, после класса и обращаться: “класс”.C

```
class Anything
{
public:
    static int s_value;
};
int Anything::s_value = 3;
```

Статические методы

Статические методы не привязаны к какому-либо одному объекту класса

У статических методов есть две особенности:

Во-первых, поскольку статические методы не привязаны к объекту, то они не имеют скрытого указателя *this! Здесь есть смысл, так как указатель *this всегда указывает на объект, с которым работает метод. Статические методы могут не работать через объект, поэтому и указатель *this не нужен.

Во-вторых, статические методы могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к не статическим членам. Это связано с тем, что нестатические члены принадлежат объекту класса, а статические методы — нет.

Указатель this — это скрытый константный указатель, содержащий адрес объекта, который вызывает метод класса.

У функции-члена есть дополнительный скрытый параметр, являющийся указателем на объект, для которого вызывалась функция. Можно явно использовать этот скрытый параметр под именем this. Считается, что в каждой функции-члене класса X указатель this описан неявно как и инициализируется, чтобы указывать на объект, для которого функция-член вызывалась. Этот указатель нельзя изменять, поскольку он постоянный (*const).

В основном this используется в функциях-членах, непосредственно работающих с указателями.

Билет №4 Наследование. Виртуальные функции. Абстрактные классы.

Наследование

Наследование является одним из основополагающих принципов ООП. В соответствии с ним, класс может использовать переменные и методы другого класса как свои собственные.

```
/*  ТИПЫ НАСЛЕДОВАНИЯ
 *  public
 * parent       -       child
 * private      ->      private унаследуется, но потомку изменять нельзя
 * protected    ->      protected
 * public       ->      public
 *
 *  protected
 * parent       -       child
 * private      ->      private унаследуется, но изменять нельзя
 * protected    ->      protected
 * public       ->      protected
 *
 *  private
 * parent       -       child
 * private      ->      private унаследуется, но изменять нельзя
 * protected    ->      private, унаследуется и можно изменять потомку
 * public       ->      private, унаследуется и можно изменять потомку
 */
```

Для установки отношения наследования после название класса ставится двоеточие, затем идет название класса, от которого мы хотим унаследовать функциональность.(class Employee : public Person)

Конструкторы при наследовании не наследуются. И если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса.(*Employee(std::string n, **int** a, std::string c):Person(n, a)*)

Абстрактный класс

```
class Shape {
public:
    virtual void show()const = 0;
    virtual void hide()const = 0;
    virtual ~Shape() { std::cout << " - delete shape" << std::endl; }
};

 class Point : public Shape {
 protected:
     int x0, y0;
 public:
     Point(int x, int y) :x0(x), y0(y) {
         std::cout << " + create point";
         show();
     }
     void show()const {
         std::cout << "Show point . (" << x0 << "," << y0 << ")" << std::endl;
     }
     void hide()const {
         std::cout << "Hide point . (" << x0 << "," << y0 << ")" << std::endl;
     }
     ~Point() { hide(); }
 };
```

Виртуальные функции

В производном классе можно переопределить функцию, в базовом классе подобная функция должна быть объявлена с ключевым словом *virtual*

При переопределении виртуальных функций следует учитывать, что переопределенная версия функции в производном классе должна иметь тот же набор параметров и возвращать объект того же типа, что и виртуальная функция в базовом классе.

Абстрактные классы

Может потребоваться определить для всех классов какой-то общий класс, который будет содержать общую для всех функциональность. И для описания подобных сущностей используются абстрактные классы.

Абстрактные классы - это классы, которые содержат или наследуют без переопределения хотя бы одну чистую виртуальную функцию. Чтобы определить виртуальную функцию как чистую, ее объявление завершается значением "*=0*". Абстрактный класс определяет интерфейс для переопределения производными классами.

Билет №5 Операторы приведения типа: const_cast, dynamic_cast, reinterpret_cast, static_cast

Операторы приведения типа

- const_cast<new_type>(exp)

Операция служит для удаления или добавления модификатора const. Используется при передачи в функцию константного указателя на место формального параметра не имеющего модификатора const. Квалификаторы const можно удалить или добавить только с помощью оператора приведения const_cast и приведения типов в стиле языка С.

! обозначенный тип должен быть таким же как и у выражения за исключением указанного const

int i;

const int * pi = &i;

// *pi имеет тип const int,

// но pi указывает на int, который константным не является

int* j = const_cast<int *> (pi); // получаем ссылку, по которой мы можем изменить переменную

- static _cast<new_type>(exp)

  На этапе компиляции (Если приведение не удалось, возникнет ошибка на этапе компиляции.)

  Может быть использован для приведения одного типа к другому.

  Применяется для преобразования указателей родственных классов иерархии, в основном - указателя базового класса в указатель на производный класс при этом во время выполнения программы производится проверка допустимости преобразования

```
class B{};
class C: public B{};

int main()
{
    C c;
    B *bp = static_cast<B*>(&c);
    B b;
    C &cp = static_cast<C&>(b); // так пидорасы делают
     return 0;
}
```

```
1   double res = static_cast<double>(13)/7;
```

- dynamic_cast <new_type>(exp)

  (преобразование во время выполнения программы)

Безопасное приведение по иерархии наследования, в том числе и для виртуального наследования.

dynamic_cast<derv_class *>(base_class_ptr_expr)

Классы должны быть полиморфными, то есть в базовом классе должна быть хотя бы одна виртуальная функция. Если эти условие не соблюдено, ошибка возникнет на этапе компиляции. Если приведение невозможно, то об этом станет ясно только на этапе выполнения программы и будет возвращен NULL.

dynamic_cast работает только для указателей и ссылок и только для полиморфных классов, связанных открытым наследованием; при попытке использовать его с чем-нибудь другим произойдет ошибка компиляции.

- reinterpret_cast<new_type>(exp)

Не портируемо, результат может быть некорректным, никаких проверок не делается. Не может быть приведено одно значение к другому значению. Обычно используется, чтобы привести указатель к указателю, указатель к целому, целое к указателю. Умеет также работать со ссылками.

применяется для преобразования не связанных между собой типов, например указателей в целые или наоборот

```
struct {double x; float y; } S;
S.x = 1; S.y = 2;

double* p;
p = reinterpret_cast<double*>(&S); // p =(double*)(&S);
std::cout << "*p=" << *p << std::endl; // *p = 1

unsigned int X;
X = reinterpret_cast<unsigned int> (&S);
std::cout << "&S = " << &S << std::endl;// выводится 16 сс
std::cout << "X = " << X << std::endl;//выводится 10 сс
X += 8;// переходим к y
float* d;
d = reinterpret_cast<float*>(X);
std::cout << "*d = " << *d << std::endl;// выводится "*d = 2"
```

Билет №6 Структуры данных: вектор

Вектор

Вектор представляет контейнер, который содержит коллекцию объектов одного типа. Для работы с векторами необходимо включить заголовок: #include <vector>

std::vector<int> v1; // пустой вектор

std::vector<int> v2(v1); // вектор v2 - копия вектора v1

std::vector<int> v3 = v1; // вектор v3 - копия вектора v1

std::vector<int> v4 (5); // вектор v4 состоит из 5 чисел

std::vector<int> v5 (5, 2); // вектор v5 состоит из 5 чисел, каждое число равно 2

std::vector<int> v6{1, 2, 4, 5}; // вектор v6 состоит из чисел 1, 2, 4, 5

При этом можно хранить в векторе элементы только одного типа, который указан в угловых скобках.

Обращение к элементам и их перебор

[index]: получение элемента по индексу (также как и в массивах), индексация начинается с нуля

at(index): функция возвращает элемент по индексу

front(): возвращает первый элемент

back(): возвращает последний элемент

Данные расположены в памяти рядом, за счёт чего они хорошо кешируются.

Методы

v.(r)begin()- если мы по массиву идем, то можно с конца пойти

v.push_back(1) - добавляет в конец вектора элемент равный 1

v.size() - возвращает размер вектора

можно менять размер и длину вектора

v.resize(5) укоротит вектор до пяти элементов

v.reserve(15) - создаст вектор длинной 15

v.insert(куда?позиция, что)

v.assign(31, 4) - создаст вектор длины 31, заполняет 4, позволяет переиспользовать данный вектор

v.clear() - позволяет очистить вектор

v.pop_back()/_front-удаляет ячейку в конце/начале

Билет №7 Структуры данных: списки

Списки

Контейнер list представляет двусвязный список. Для его использования необходимо подключить заголовочный файл list.

Для контейнера list можно использовать функции front() и back(), которые возвращают соответственно первый и последний элементы.

Чтобы обратиться к элементам придется выполнять перебор элементов с помощью циклов или итераторов

Для получения размера списка можно использовать функцию size()

Функция empty() позволяет узнать, пуст ли список.

С помощью функции resize() можно изменить размер списка.

resize(n): оставляет в списке n первых элементов. Если список содержит больше элементов, то он усекается до первых n элементов. Если размер списка меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию

resize(n, value): также оставляет в списке n первых элементов. Если размер списка меньше n, то добавляются недостающие элементы со значением value

insert(pos, val): вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент

insert(pos, n, val): вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.

insert(pos, begin, end): вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.

insert(pos, values): вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

clear(p): удаляет все элементы

pop_back(): удаляет последний элемент

pop_front(): удаляет первый элемент

erase(p): удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент

erase(begin, end): удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент.

```
int main()
{
    std::list<int> numbers = { 1, 2, 3, 4, 5 };

    int first = numbers.front();  // 1
    int last = numbers.back();  // 5

    // перебор в цикле
    for (int n : numbers)
        std::cout << n << "\t";
    std::cout << std::endl;

    // перебор с помощью итераторов
    for (auto iter = numbers.begin(); iter != numbers.end(); iter++)
    {
        std::cout << *iter << "\t";
    }
    std::cout << std::endl;
    return 0;
}
```

Билет №8 Структуры данных: дерево. Сбалансированные деревья поиска.

Деревья. Сбалансированные деревья поиска.

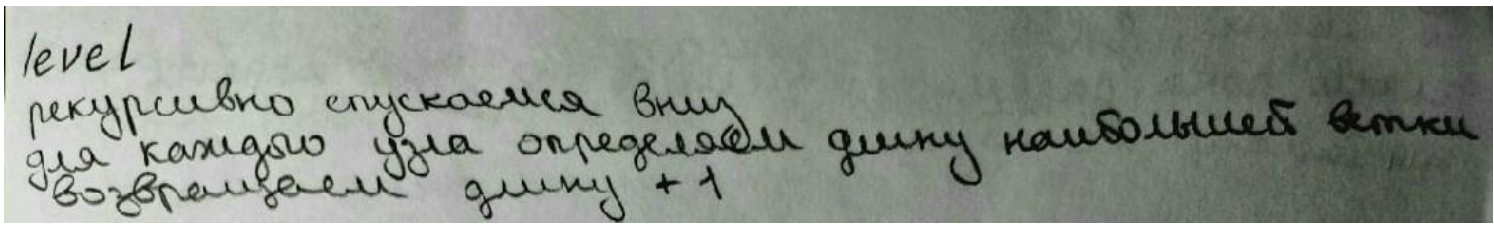
Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

template <typename ElementType>

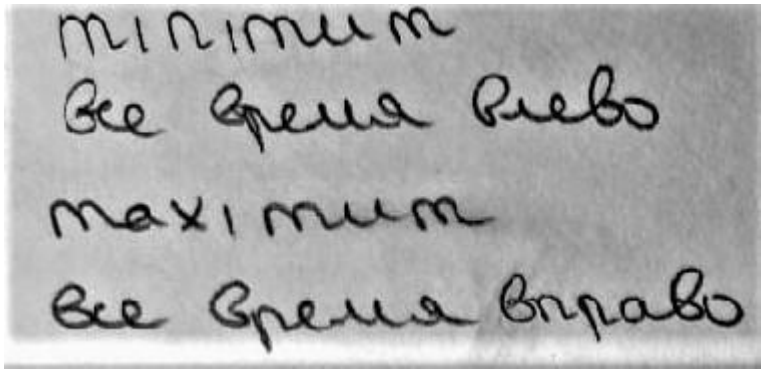
```
class BinTree {
    struct Node {
        Node *left, *right, *parent;
        ElementType data;
        Node(const ElementType& X) : right(nullptr), left(nullptr), parent(nullptr), data(X) {}
        ~Node() {
            if (left) delete left;
            if (right) delete right;
            right = left = parent = nullptr;
        }
```

```
        int level() const {
            int L = (left != nullptr) ? left->level() : 0;
            int R = (right != nullptr) ? right->level() : 0;
            return L > R ? L + 1 : R + 1;
        }
```

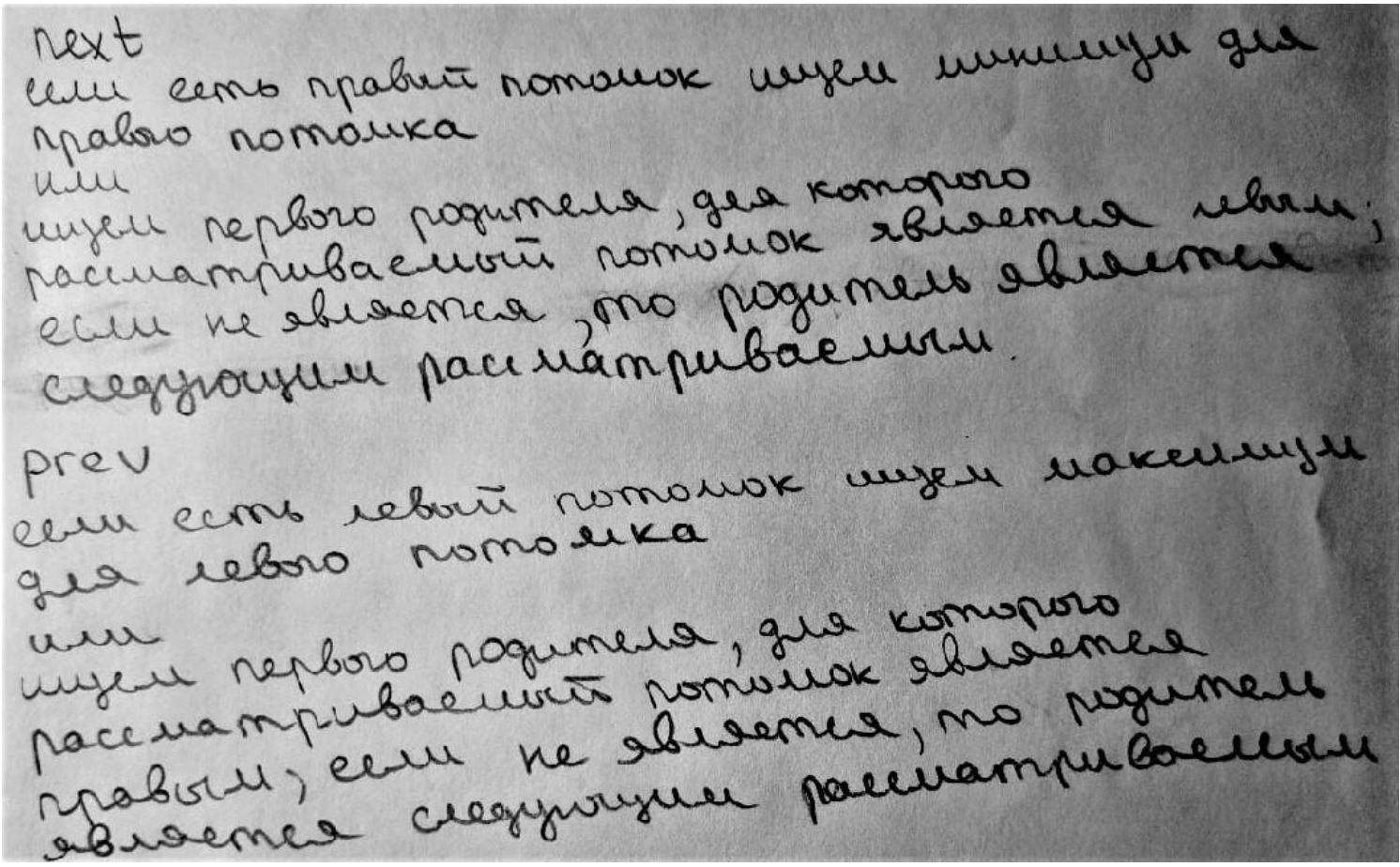
Вычисляем какая из веток(левая или правая) длиннее.



```
        Node* minimum() {
            Node *now = this;
            while (now->left != nullptr) now = now->left;
            return now;
        }
        Node* maximum() {
            Node *now = this;
            while (now->right != nullptr) now = now->right;
            return now;
        }
```



```
Node* next() {
    if (right != nullptr) return right->minimum();
    Node *p = parent, *now = this;
    while (p != nullptr) {
        if (now == p->left) break;
        now = p;
        p = p->parent;
    }
    return p;
}
Node* prev() {
    if (left != nullptr) return left->maximum();
    Node *p = parent, *now = this;
    while (p != nullptr) {
        if (now == p->right) break;
        now = p;
        p = p->parent;
    }
    return p;
}
```



```
    };
    Node * root;
    unsigned int count;
public:
```

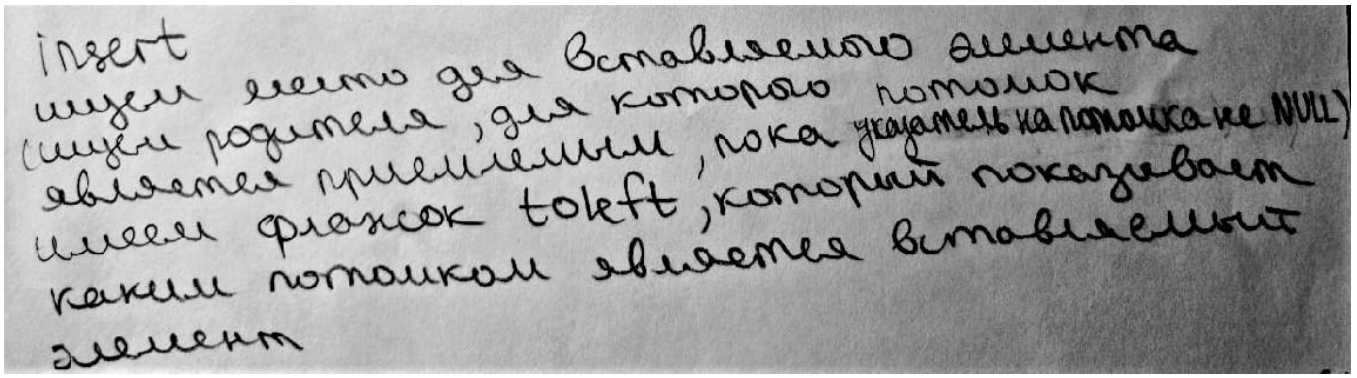
```
BinTree() : root(nullptr), count(0) {}
```

```
~BinTree() { clear(); }
```

```
void clear() { if (root) delete root; root = nullptr; }
```

```
    void insert(const ElementType &X) {
        ++count;
        if (root == nullptr) { root = new Node(X); return; }
        Node *now, *p;
        bool toLeft;
        now = root;
        do {
            p = now;
```

```
        if (X < now->data) {
            now = now->left;
            toLeft = true;
        }else {
            now = now->right;
            toLeft = false;
        }
    } while (now); // now - указатель на структуру Node
    now = new Node(X);
    if (toLeft)
        p->left = now;
    else
        p->right = now;
    now->parent = p;
}
```



```
unsigned int size() const { return count; }
```

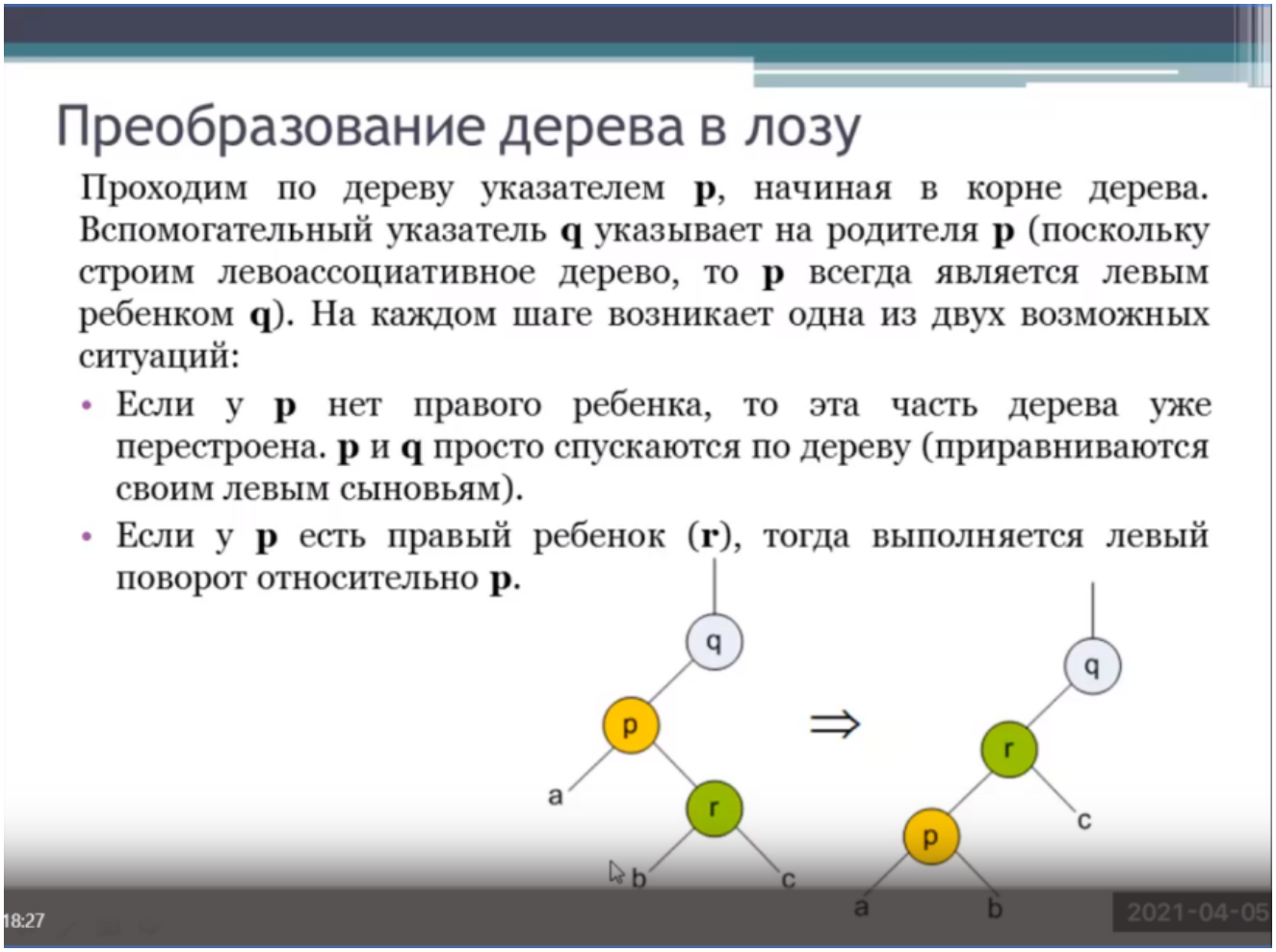
```
int height()const { return (root != nullptr) ? root->level() : 0; }
```

```
class iterator { // для обхождения коллекции данных, не вдаваясь в подробности реализации
    Node * now;
public:
    iterator(Node *p = nullptr) : now(p) {}
    const ElementType& operator*()const { return now->data; } // разыменование указателя на узел
    Возвращает константную ссылку на элемент
    bool operator==(const iterator &p) const { return now == p.now; }
    bool operator!=(const iterator &p) const { return now != p.now; }
    iterator& operator++() { // ++iter
```

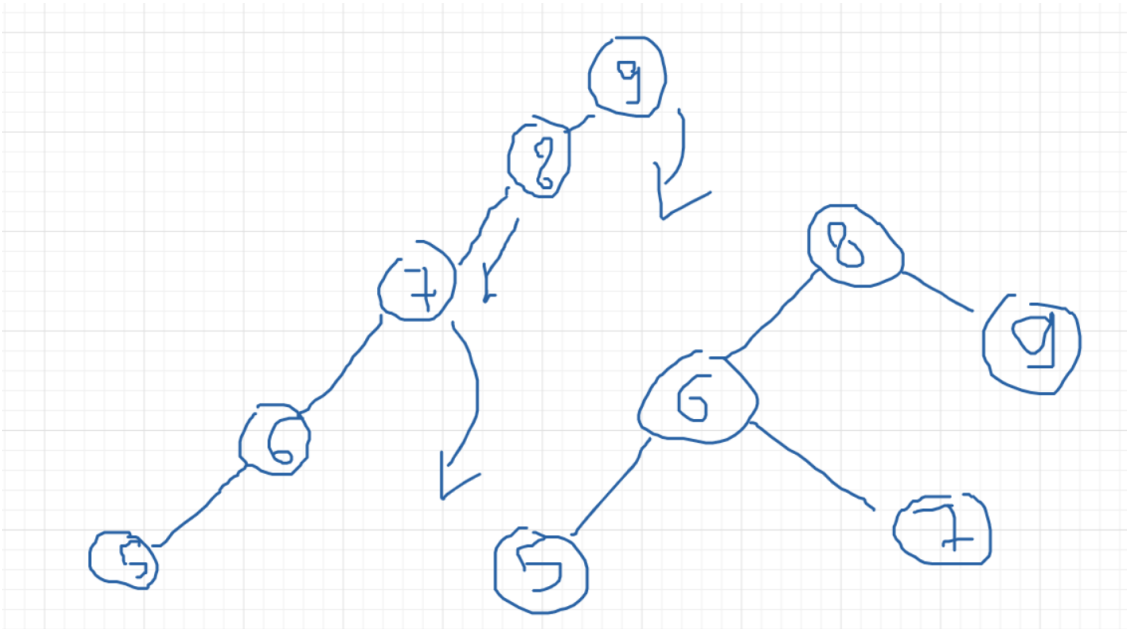
```
            if (now) now = now->next();
            else throw "Iterator error";
            return *this;
        }
        iterator operator++(int) { // iter++
            Node *tmp = now;
            if (now) now = now->next();
            else throw "Iterator error";
            return iterator(tmp);
        }
        friend class BinTree;
    };
    iterator begin()const {
        if (root) return iterator(root->minimum());
        return end();
    }
    iterator end()const { return iterator(nullptr); }

    iterator find(const ElementType& X) {
        Node *now = root;
        while (now) {
            if (X == now->data) break;
            if (X < now->data) now = now->left;
            else now = now->right;
        }
        return iterator(now);
    }
    void balance();
    void erase(iterator);
};
```

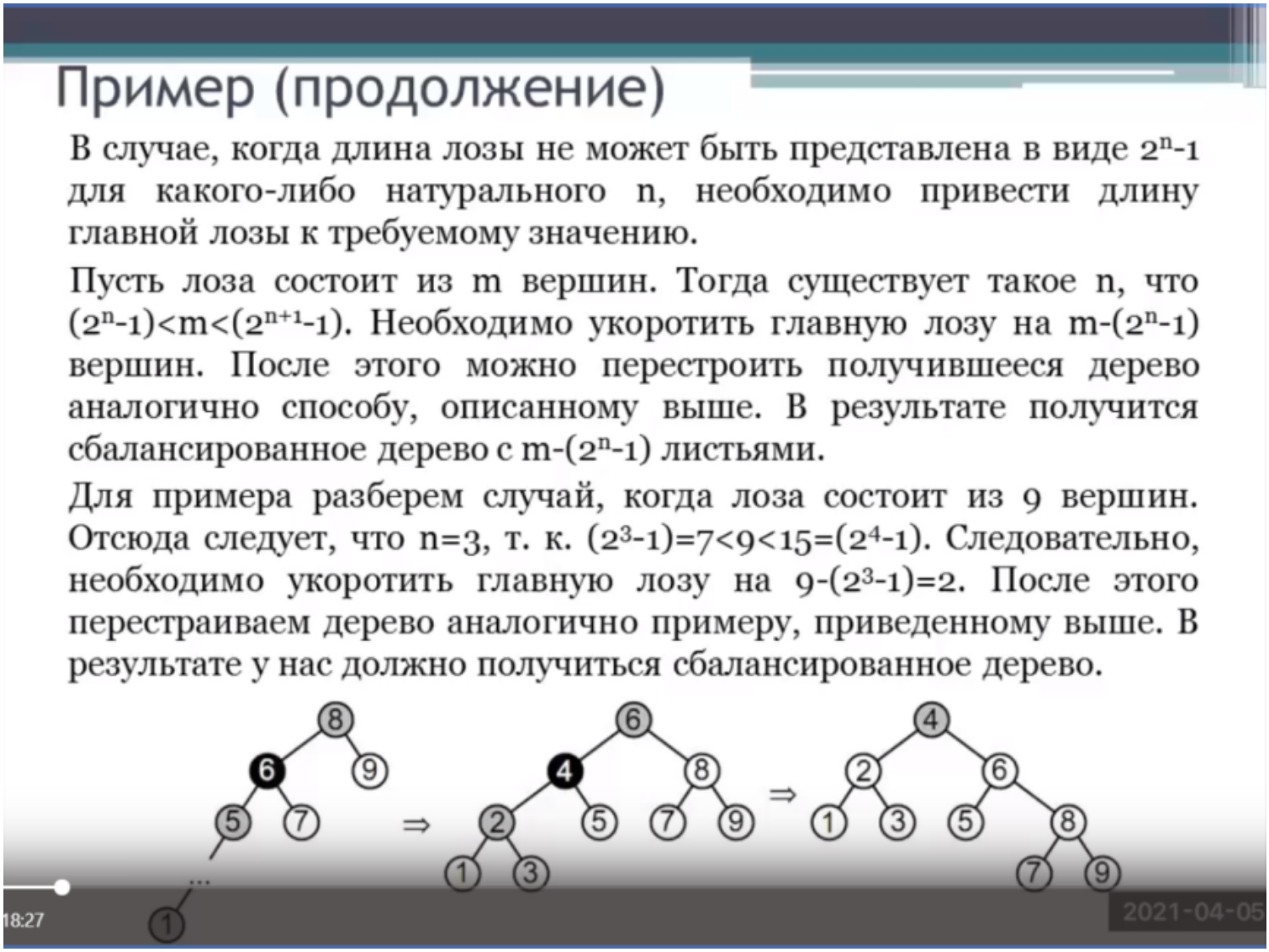
```
template <typename ElementType>
void BinTree<ElementType>::balance(){
    if (root == nullptr) return; // if (count<2)
    // Вытянуть в лозу
    Node *p, *q, *r;
```

```
p = root;
while (p) {
    if (p->right == nullptr)
        p = p->left;
    else {
        r = p->parent;
        q = p->right;

        if (r) r->left = q; else root = q;
        q->parent = r;

        p->right = q->left;
        if (q->left) q->left->parent = p;

        q->left = p;
        p->parent = q;

        p = q;
    }
}
```



```
// Сломать лишнее
int n = 0, n2 = 1;
for (int i = count + 1; i > 1; i >>= 1, ++n, n2 <<= 1); // i=i/2, n=n+1, n2=n2*2
int Lishnee = count - (n2 - 1);
q = root;
while (Lishnee > 0) {
    p = q->left;
    p->right = q;

    p->parent = q->parent;
    if (q->parent == nullptr) root = p;
    else q->parent->left = p;

    q->parent = p;
    q->left = nullptr;

    q = p->left;
    --Lishnee;
}
```



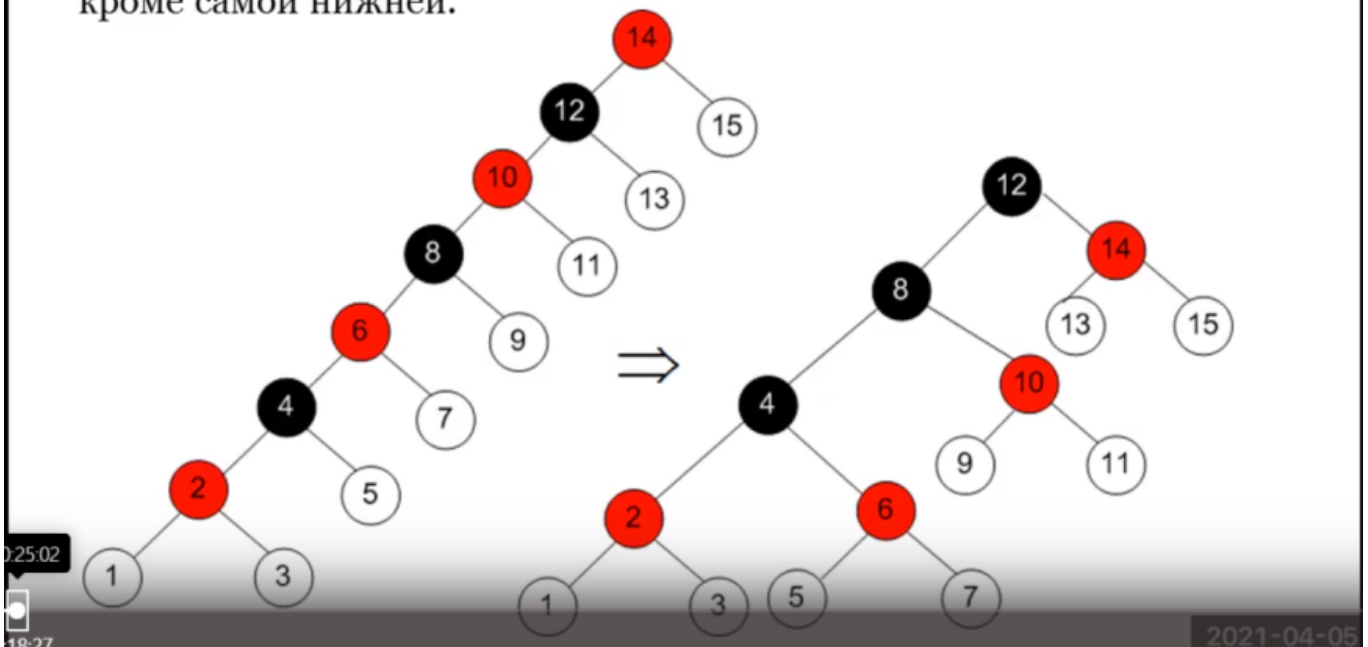


```
// Ломать лозу для балансировки
Node *red, *black;
int BlackInStep = (n2 - 1) / 2;

for (int step = 1; step < n; ++step, BlackInStep /= 2) {
    red = root;
    for (int i = 1; i <= BlackInStep; ++i) {
        black = red->left;
        black->parent = red->parent;

        if (red->parent == nullptr) root = black;
        else red->parent->left = black;

        red->left = black->right;
```

```
            if (black->right) black->right->parent = red;

            black->right = red;
            red->parent = black;
            red = black->left;
        }
    }
}
```

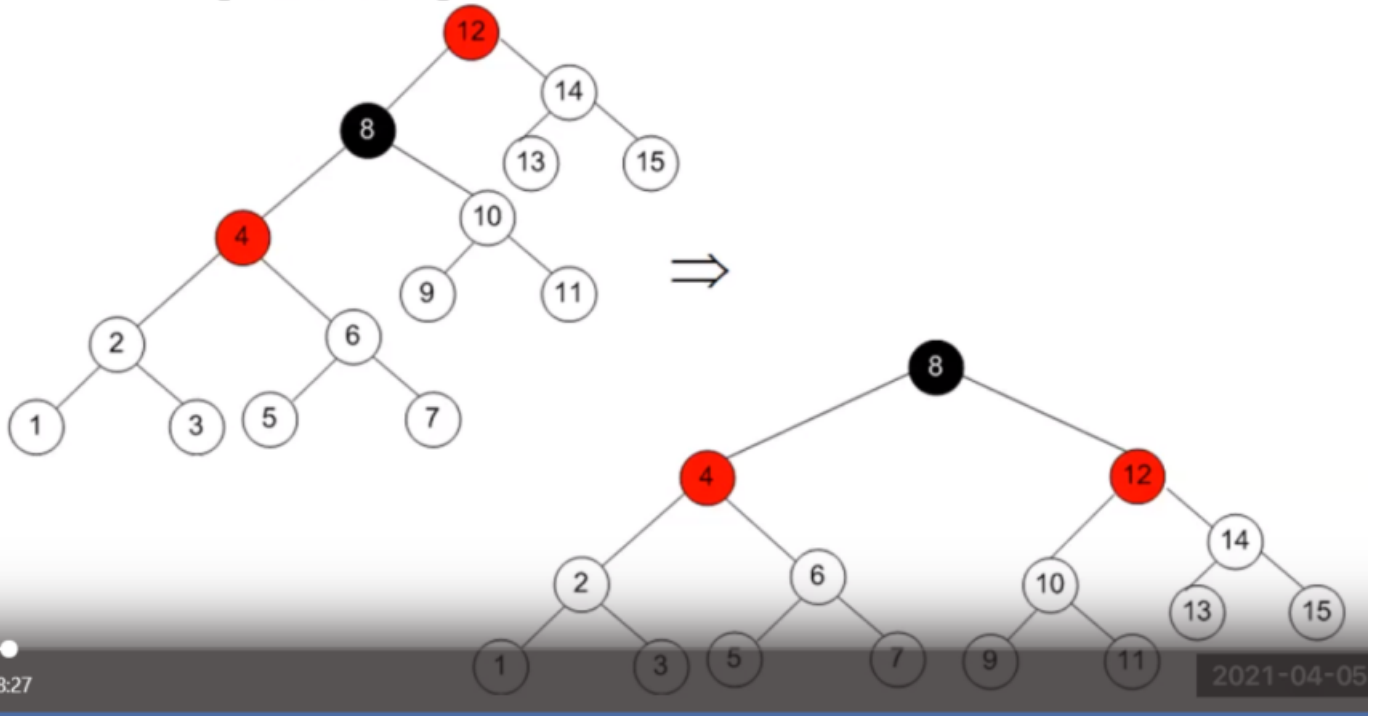
Пример (продолжение)

Для второго перестроения сначала перекрасим красные вершины в белые. Далее перекрасим каждую вторую черную вершину в красный цвет, начиная в корне. Теперь, как и раньше, выполним малый правый поворот относительно каждой красной вершины, кроме самой нижней.



Пример (продолжение)

Третье перестроение аналогично первым двум. Вершины 12 и 4 перекрашиваются в серый цвет, затем выполняется малый правый поворот относительно вершины 12. В результате получается сбалансированное дерево.



Удаление элемента

```
template <typename ElementType>
void BinTree<ElementType>::erase(typename BinTree<ElementType>::iterator pos) {
    Node *toDelete = pos.now;
    if (toDelete == nullptr) return;

    Node *Alt;
    if (toDelete->left == nullptr) //если нет левого потомка берем правого потомка
        Alt = toDelete->right;
    else if (toDelete->right == nullptr)  //если нет правого потомка берем левого потомка
        Alt = toDelete->left;
    else {                                      //   (right != nullptr , left != nullptr)
        Alt = toDelete->right->minimum(); // Находим минимального потомка в правом поддереве
        if (Alt->parent != toDelete) { // Правый потомок toDelete не является минимальным
            Alt->parent->left = Alt->right;
            if (Alt->right) Alt->right->parent = Alt->parent;
            Alt->right = toDelete->right;
            toDelete->right->parent = Alt;
        }
        Alt->left = toDelete->left; //перекидываем потомков toDelete
        toDelete->left->parent = Alt;
    }

    if (toDelete->parent == nullptr) // toDelete == root
        root = Alt;
    else { // Меняем потомков у родителей toDelete
        if (toDelete->parent->left == toDelete)
            toDelete->parent->left = Alt;
        else
            toDelete->parent->right = Alt;
    }
    if (Alt!=nullptr) Alt->parent = toDelete->parent;

    toDelete->right = nullptr; toDelete->left = nullptr;
    delete toDelete;
    --count;
}
```

Итог:

- Определение бинарного дерева
- Структура бинарного дерева

  класс bin_tree:

  - структура node (левый, правый потомки, родитель)
  - свойства BinTree (root, count)
- Методы(level, min/max, next/prev,insert, erase для node)
- Класс итератор (указатель + методы)
- Удаление элемента из дерева (1. Находим альтернативу 2. Переписываем связи 3. Удаляем элемент to_delete)
- Балансировка (вытянуть лозу(лево-ассоциативное дерево) -> сломать(нечетные) до кол-ва 2^n -1 элементов в лозе -> раскрашиваем в Ч и К, начиная с красного -> поворачиваем вправо относительно красных вершин, кроме последней красной -> все красные окрашиваем в белый -> повторяем процесс)

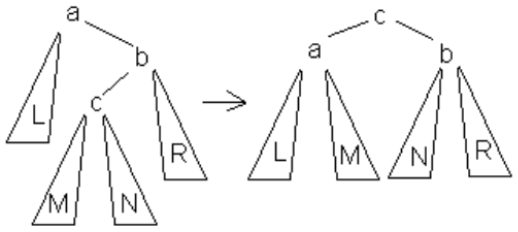
Билет №9. Сбалансированное двоичное дерево (AVL tree).

AVL-деревья

```
void insert(const ElementType &X) {
    ++count;
    if (root == nullptr) { root = new Node(X); return; }
    Node *now, *p;
    bool toLeft;
    now = root;
// Поиск места для нового элемента
    do {
      p = now;
      if (X < now->data) {
        now = now->left;
        toLeft = true;
      }
      else {
        now = now->right;
        toLeft = false;
      }
    } while (now);
    now = new Node(X);
    if (toLeft)
```

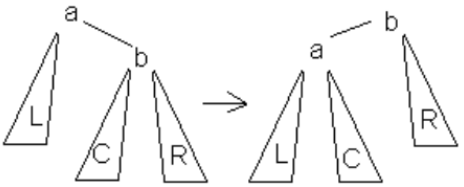
```
        p->left = now;
    else
        p->right = now;
    now->parent = p;
    // Восстановление баланса
    do {
        if (now->parent->left == now) now->parent->balance -= 1;
         //если now - левый потомок, вычитаем 1 из баланса родителя
        else now->parent->balance += 1;

        now = now->parent;
        switch (now->balance) { проверяем баланс родителя
        case 2: // баланс родителя 2
            if (now->right->balance == -1) // баланс правого потомка -1
                now = _BigLeftRotate(now);
```

Большое левое вращение



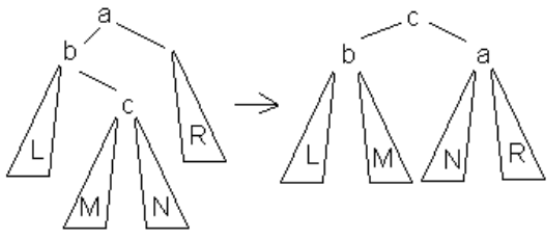

```
            else
                now = _LeftRotate(now);
```

Малое левое вращение



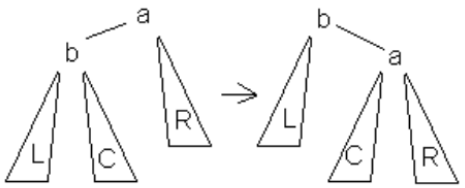

```
            break;
        case -2:  // баланс родителя -2
            if (now->left->balance == 1) // баланс левого потомка 1
                now = _BigRightRotate(now);
```

Большое правое вращение




```
else
                now = _RightRotate(now);
```

.Малое правое вращение

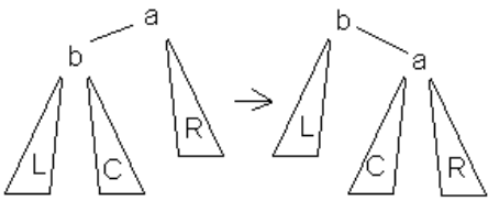



```
            break;
        }
    } while (now!=root && now->balance!=0);// проверяем баланс родителей до корня или делаем балансировку какого-то родителя нулевой
```

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

```
template <typename ElementType>
typename AVLTree<ElementType>::Node*
AVLTree<ElementType>::_RightRotate(typename AVLTree<ElementType>::Node * c) {
    if (c->balance != -2) return c;
    Node *a = c->left;
    if (a->balance == 1) return c;
    c->left = a->right;
    if (a->right) a->right->parent = c;
    a->parent = c->parent;
    if (c->parent) {
        if (c->parent->left == c) c->parent->left = a;
```

```
        else c->parent->right = a;
    }
    else
        root=a;
    a->right = c;
    c->parent = a;

    if (a->balance == -1) {
        a->balance = 0; c->balance = 0;
    } else {
        a->balance = 1; c->balance = -1;
    }
    return a;
}
```

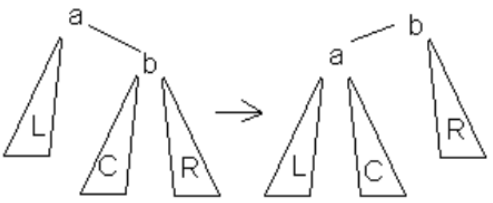
3.**Малое правое вращение** 

Данное вращение используется тогда,

когда высота b-поддерева — высота R = 2 и высота C <= высота L.

```
template <typename ElementType>
typename AVLTree<ElementType>::Node*
    AVLTree<ElementType>::_LeftRotate(typename AVLTree<ElementType>::Node * a) {
    if (a->balance != 2) return a;
    Node *c = a->right;
    if (c->balance == -1) return a;
    a->right = c->left;
    if (c->left) c->left->parent = a;
    c->parent = a->parent;
    if (a->parent) {
        if (a->parent->left == a) a->parent->left = c;
        else a->parent->right = c;
```

```
    }
    else
        root=c;
    c->left = a;
    a->parent = c;

    if (c->balance == 1) {
        a->balance = 0; c->balance = 0;
    }
    else {
        a->balance = 1; c->balance = -1;
    }
    return c;
}
```

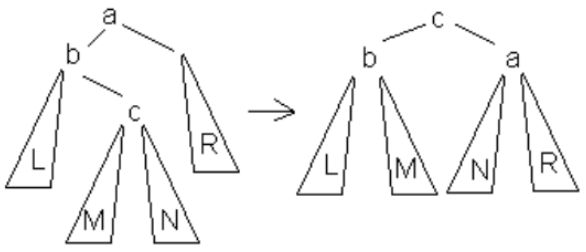
1.**Малое левое вращение** 

Данное вращение используется тогда,

когда высота b-поддерева — высота L = 2 и высота C <= высота R.

```
//-----------------------------------------------------------------------------------------------------------------------------------------
template <typename ElementType>
typename AVLTree<ElementType>::Node*
    AVLTree<ElementType>::_BigRightRotate(typename AVLTree<ElementType>::Node * c) {
    if (c->balance != -2) return c;
    Node *a = c->left;
    if (a->balance != 1) return c;
    Node * b = a->right;

    a->right = b->left; if (b->left) b->left->parent = a;
    c->left = b->right; if (b->right) b->right->parent = c;

    b->parent = c->parent;
    if (c->parent) {
```

```
        if (c->parent->left == c) c->parent->left = b;
        else c->parent->right = b;
    }
    else
        root = b;

    b->left = a; a->parent = b;
    b->right = c; c->parent = b;

    if (b->balance == 0) {
        a->balance = 0; c->balance = 0;
     } else if (b->balance==1) {
         a->balance = -1; c->balance = 0;
     } else {
         a->balance = 0; c->balance = 1;
     }
     b->balance = 0;
     return b;
}
```

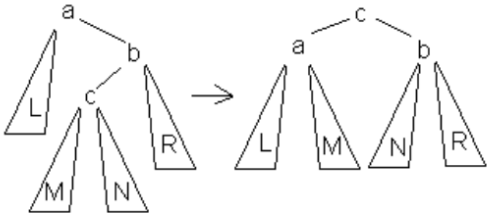
4.**Большое правое вращение** 

Данное вращение используется тогда, когда высота b-поддерева — высота R = 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 можно заметить, что большое вращение это комбинация правого и левого малого вращения. Из-за условия балансированности высота дерева O(log(N)), где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций.

```
//-----------------------------------------------------------------------------
template <typename ElementType>
typename AVLTree<ElementType>::Node*
    AVLTree<ElementType>::_BigLeftRotate(typename AVLTree<ElementType>::Node * a) {
```

```
    if (a->balance != 2) return a;
    Node *c = a->right;
    if (c->balance != -1) return a;
    Node * b = c->left;

    a->right = b->left; if (b->left) b->left->parent = a;
    c->left = b->right; if (b->right) b->right->parent = c;

    b->parent = a->parent;
    if (a->parent) {
        if (a->parent->left == a) a->parent->left = b;
        else a->parent->right = b;
    }
    else
        root = b;

    b->left = a; a->parent = b;
    b->right = c; c->parent = b;

    if (b->balance == 0) {
        a->balance = 0; c->balance = 0;
    } else if (b->balance == 1) {
        a->balance = -1; c->balance = 0;
    } else {
        a->balance = 0; c->balance = 1;
    }
    b->balance = 0;
    return b;
}
```

2.**Большое левое вращение** 

Данное вращение используется тогда, когда высота b-поддерева — высота L = 2 и высота c-поддерева > высота R.

Итог:

Бинарное дерево + добавлено поле balance в node

Добавление балансировки с помощью вращений в операции insert и erase

Вращения:

- BigLeftRotate [баланс родителя = 2, баланс правого потомка = -1]
- LeftRotate [баланс родителя = 2, баланс правого потомка = 0 or 1],
- BigRightRotate[баланс родителя = -2, баланс левого потомка = 1]
- RightRotate [баланс родителя = -2, баланс левого потомка 0 or -1]

Билет №10. Инфиксная и постфиксная формы записи алгебраических выражений. Дерево разбора выражения.

Инфиксная нотация — это форма записи математических и логических формул, в которой операторы записаны в инфиксном стиле между операндами, на которые они воздействуют (например, 2 **+** 2).

Постфиксная форма - форма записи математических и логических выражений, в которой операнды расположены перед знаками операций (2 2 **+**).

InfixFilter:

- Удаляет пробелы и табуляции из строки
- Унарный минус превращаем в 0-
- Добавляем в буфер элементы до того, как не появится символ отличный от букв или цифр
- Если в буфер вошла функция с количеством символов больше 1, то ищем ее короткое имя

Infix2Postfix: (преобразует инфиксное выражение в постфиксное)

- Строим таблицу приоритетов операций

по столбцам - токены во входной строке

по строкам - токены в стеке

- ActionsColNumber: действия по колонке; выбирается в зависимости от входного символа

  (возвращает номер колонки)

- ActionsRowNumber: если стек не пустой, вызываем от последнего элемента в стеке

  (возвращает номер строки)

- В зависимости от номеров row и col вызываем действие в таблице приоритетов операций

Классы для дерева разбора выражения:

Класс FormulaNode - абстрактный(calc()-считает выражение, str()-выводит выражения)

Производные для FormualNode: NumNode(число), BinNode(бинарная функция), производные от BinNode, UnarNode(Родитель для всех унарных функций)

Класс Formula(для каждого выражения)

Postfix2Tree

- В зависимости от типа элемента в строке создается соответствующий узел
- Узлы складываем в стек для их связи
- Если узел является операцией, то вытаскиваем из стека переменные/числа и присваиваем их свойствам узла
- Результат - самый верхний узел в стеке

Билет №11 . Стандартная библиотека шаблонов (STL). Контейнеры, алгоритмы.

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multlmap), множества (set), множества с дубликатами (multiset).

multimap - допускает дублирование ключей

mp1[0] = 7; // ключ - число

mp2["zero"] = 0; // ключ - строка

pair <int, int> p = make_pair(1, 3);

mp3[p] = 3; // ключ - значение

```
map <string, string> book = {{"Hi", "Привет"},
                             {"Student", "Студент"}
                             {"!", "!"}};

cout << book["Hi"];
```

При добавлении нового элемента контейнер будет отсортирован по возрастанию.

map < <L>, <R> > <имя>;

<L> — этот тип данных будет относиться к значению ключа.

<R> — этот тип данных соответственно относится к значению.

- обращение к элементам map:

```
void PrintMap(const map<int, string>& m) {
  cout << "Size = " << m.size() << endl;
  for (auto item: m) {
    cout << item.first << ": " << item.second << endl;
  }
}
```

- удаление элемента по ключу

```
events.erase(1970);
```

словарь с заранее известными результатами

```
map<string, int> m = {{"one", 1}, {"two" ,2}, {"three", 3}};
```

Find:

map <string, string> :: iterator it, it_2;

it = book.find("book");

set -множество-контейнер в котором хранятся уникальные элементы, повторение запрещено

- insert - добавление элемента
- erase

multiset - повторения

Последовательные контейнеры Векторы (vector), двусторонние очереди (deque) и списки (list) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они могут быть реализованы с разной эффективностью:

Итак, *вектор* — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

Стеки (stack) В стеке допускаются только две операции, изменяющие его размер — добавление элемента в вершину стека и выборка из вершины. Стек можно реализовать на основе любого из рассмотренных контейнеров: вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся, поэтому он называется адаптером контейнера.

Алгоритмы

Таблица 14.1. Немодифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
adjacent_find	Нахождение пары соседних значений
count	Подсчет количества вхождений значения в последовательность
count_if	Подсчет количества выполнений условия в последовательности
equal	Попарное равенство элементов двух последовательностей
find	Нахождение первого вхождения значения в последовательность
find_end	Нахождение последнего вхождения одной последовательности в другую
find_first_of	Нахождение первого значения из одной последовательности в другой
find_if	Нахождение первого соответствия условию в последовательности
for_each	Вызов функции для каждого элемента последовательности
mismatch	Нахождение первого несовпадающего элемента в двух последовательностях
search	Нахождение первого вхождения одной последовательности в другую
search_n	Нахождение n-го вхождения одной последовательности в другую

Алгоритм	Выполняемая функция
copy	Копирование последовательности, начиная с первого элемента
copy_backward	Копирование последовательности, начиная с последнего элемента
fill	Замена всех элементов заданным значением
fill_n	Замена первых n элементов заданным значением
generate	Замена всех элементов результатом операции
generate_n	Замена первых n элементов результатом операции
iter_swap	Обмен местами двух элементов, заданных итераторами
random_shuffle	Перемещение элементов в соответствии со случайным равномерным распределением
remove	Перемещение элементов с заданным значением
remove_copy	Копирование последовательности с перемещением элементов с заданным значением
remove_copy_if	Копирование последовательности с перемещением элементов при выполнении предиката
remove_if	Перемещение элементов при выполнении предиката
replace	Замена элементов с заданным значением
replace_copy	Копирование последовательности с заменой элементов с заданным значением
replace_copy_if	Копирование последовательности с заменой элементов при выполнении предиката
replace_if	Замена элементов при выполнении предиката
reverse	Изменение порядка элементов на обратный
reverse_copy	Копирование последовательности в обратном порядке
rotate	Циклическое перемещение элементов последовательности
rotate_copy	Циклическое копирование элементов
swap	Обмен местами двух элементов
swap_ranges	Обмен местами элементов двух последовательностей
transform	Выполнение заданной операции над каждым элементом последовательности
unique	Удаление равных соседних элементов
unique_copy	Копирование последовательности с удалением равных соседних элементов

Algorithm — заголовочный файл в стандартной библиотеке языка программирования C++, включающий набор функций для выполнения алгоритмических операций над контейнерами и над другими последовательностями.

Функция std::find() выполняет поиск первого вхождения заданного значения в контейнере. В качестве аргументов std::find() принимает 3 параметра:

- итератор для начального элемента в последовательности;
- итератор для конечного элемента в последовательности;
- значение для поиска.

- find_if вместо значения для поиска - лямбда функция или ссылка на функцию
- count, count_if - считают количество вхождений элемента
- for_each | first, last, function | Применяет function ко всем объектам (от first до last)

```
std::array arr{ 1, 2, 3, 4 };

std::for_each(arr.begin(), arr.end(), doubleNumber);
```
(выведет 2 4 6 8)

- swap | a, b | Заменяет один объект другим
- move()

  после использования функции x и y поменяются местами (меняются ссылки)

```
void swap(T& x, T& y)
{
  T tmp { std::move(x) }; // вызывает конструктор перемещения
  x = std::move(y); // вызывает оператор присваивания перемещением
  y = std::move(tmp); // вызывает оператор присваивания перемещением
}
```

- copy(откуда, докуда)
- make_heap | first, last | Создает кучу из значений диапазона first last
- pop_heap | first, last | Меняет значения в first и last-1. Помещает диапазон first last-1 в кучу
- push_heap | first, last | Помещает значение из last-1 в результирующую кучу (heap, область динамической памяти) диапазон от first до last
- sort_heap | first, last | Упорядочивает элементы в куче first last

*Чтобы отсортировать массив нам нужно использовать схему ниже:

2) sort(<имя массива>, <имя массива> + <размер массива>, компаратор);

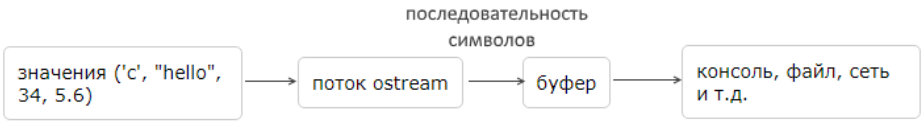
Билет №12 Стандартная библиотека шаблонов (STL). Потоки ввода-вывода.

Потоки ввода - вывода

Для использования объектно-ориентированного консольного ввода-вывода с помощью потоков (stream) STL в программу необходимо включить заголовочный файл <iostream>, а для файлового ещё и <fstream>.

При запуске консольного приложения неявно открываются четыре потока: cin — для ввода с клавиатуры, cout — для буферизованного вывода на монитор, cerr — для небуферизованного вывода на монитор сообщений об ошибках и clog — буферизованный аналог cerr. Эти четыре символа определены посредством <iostream>.

Для ввода-вывода сначала необходимо создать поток — экземпляр соответствующего класса STL, а затем связать его с файлом. Для потока вывода используется класс ofstream, для потока ввода — ifstream, для потока ввода-вывода — fstream. В каждом из этих классов есть метод open(), который связывает поток с файлом. Методу передаются два параметра: имя файла и режим открытия файла. Второй параметр представляет собой набор битовых флагов, определяющих режим открытия файла (чтение, запись и пр.) и способ работы с данными (текстовый или двоичный режим).



Операции чтения и записи в поток, связанный с файлом, осуществляются либо с помощью операторов << и >>, перегруженных для классов потоков ввода-вывода, либо с помощью любых других методов классов потоков ввода-вывода.

Некоторые наиболее употребляемые методы:

Чтение данных:

getline() *// читает строку из входного потока.*

get() *// читает символ из входного потока.*

ignore() *// пропускает указанное число элементов от текущей позиции чтения.*

read() *// читает указанное количество символов из входного потока и сохраняет их в буфере (неформатированный ввод).*

Запись данных:

flush() *// вывод содержимого буфера в файл (при буферизованном вводе-выводе)*

put() *// выводит символ в поток.*

write() *// выводит в поток указанное количество символов из буфера (неформатированный вывод)*

```
#include <iostream>
#include <fstream>
using namespace std;
const char *filename = "testfile2.txt";
int main() {
    ofstream ostr;
    ostr.open(filename);
    if (ostr) {
        for (int i = 0; i < 16; i++) {
            ostr << i*i << endl;
            if (ostr.bad()) {
                cerr << "Unrecoverable write error" << endl;
                return 1;
            }
        }
        ostr.close();
    }
```

```
    else {
        cerr << "Output file open error \"" << filename << "\"" << endl;
        return 1;
    }
    // открытие файла (в конструкторе) для чтения в текстовом режиме,
    // чтение данных, форматированный вывод на консоль, закрытие файла.
    int data;
    int counter = 0;
    ifstream istr(filename);
    if (istr) {
        while (!(istr >> data).eof()) {
            if (istr.bad()) {
                cerr << "Unrecoverable read error" << endl;
                return 2;
            }
            cout.width(8);
            cout << data;
            if (++counter % 4 == 0) {
                cout << endl;
            }
        }
        istr.close();
    }
    else {
        cerr << "Input file open error \"" << filename << "\"" << endl;
        return 2;
    }
    return 0;
}
```

```
std::wifstream inFile(fname);
            if (inFile) {
                    while (inFile.peek() != EOF)
                      }
```

Оператор <<

Первый параметр оператора << представляет ссылку на неконстантный объект ostream, так как запись в поток изменяет его состояние и нельзя копировать объект класса ostream.

Второй параметр оператора определяется как ссылка на константу объекта класса, который надо вывести в поток.

```
std::ostream& operator << (std::ostream &os, const Person &p)
{
    return os << p.name << "  " << p.age;
}
```

```
std::istream& operator >> (std::istream& in, Person& p)
{
    in >> p.name >> p.age;
    if (!in)
    {
        p = Person();
    }
    return in;
}
```

Билет №13 Лямбда выражения, auto.

Лямбда выражения

```
/* lambda function */
/*  [объявление лямбда выражений](список аргументов)->возвращаемый_тип {тело лямбда выражения}
 * int total, factor, x; ...
 * [&total, factor] - передаём переменную total по ссылке, factor по значению
 * [&total, &factor, x] - передаём две переменные по ссылке, а переменную x по значению
 * [factor, &total, x] - factor по значению, total по ссылку
 * [&, factor] - передаём все переменные кроме factor по сслыке, а factor по значению
 * [-, &factor] - передаём все переменные по значению, а factor по ссылке
 * [-] - передаём все пременные по значению
 * [&] - передаём все переменные по ссылке
 *
 * */
```

Компилятор создает структуру типа comparator:

```
std::function lambda = [ ](int a, int b)->bool {return a > b;};
 std::cout << lambda(1 ,5);
```

Auto

Ключевое слово auto при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен присвоить тип переменной исходя из инициализируемого значения.

Билет №14 Rvalue-ссылки, семантика перемещения.

Rvalue - ссылка - техническое расширение языка C++. Они позволяют программистам избегать логически ненужного копирования и обеспечивать возможность идеальной передачи.

R-value ссылка ведет себя точно так же, как и L-value ссылка, за исключением того, что она может быть связана с временным объектом, тогда как L-value связать с временным (не константным) объектом нельзя.

Возможность реализовать семантику перемещения с R-value ссылкой

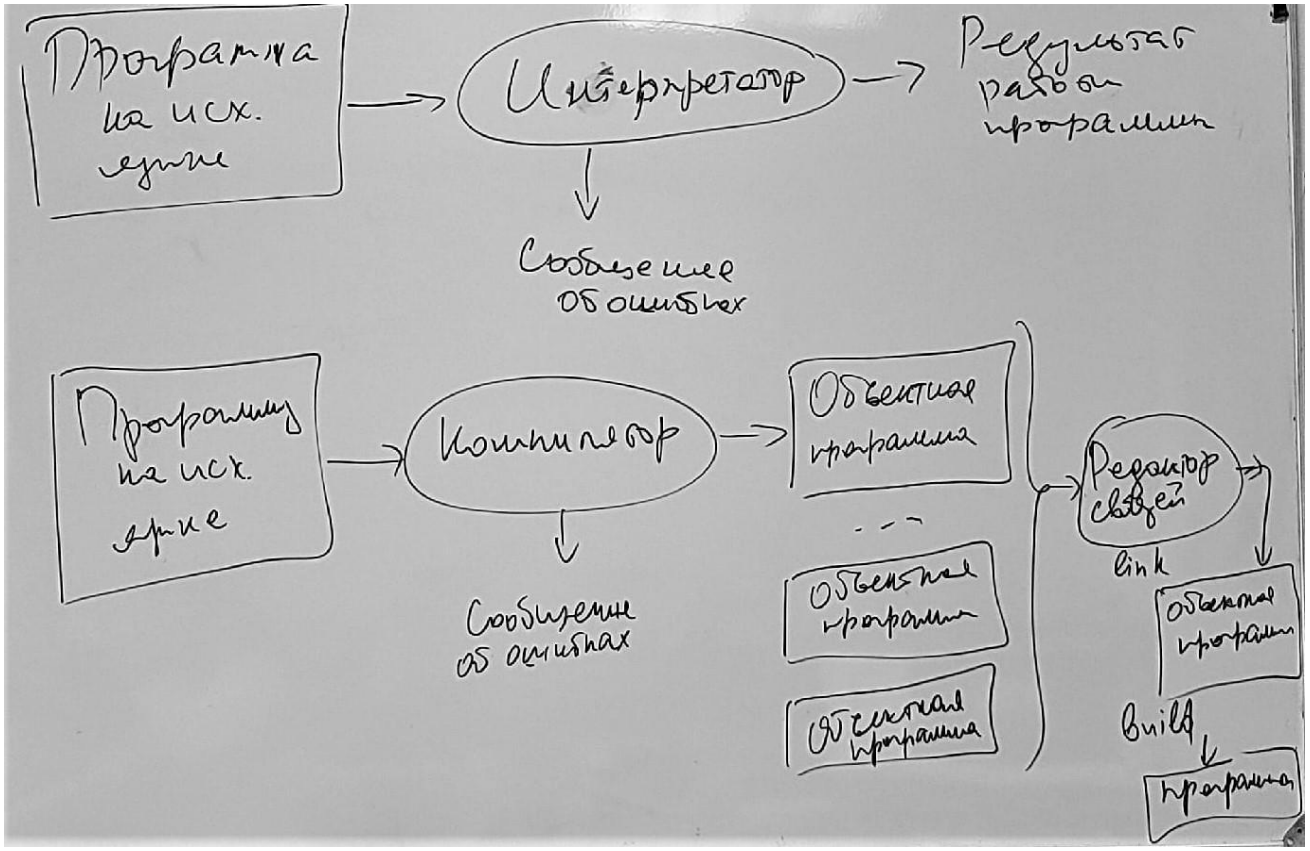
Семантика перемещения

Семантика перемещения -средство языка, предназначенное для осуществления перемещения данных во время инициализации и конструирования новых объектов. Для практического осуществления семантики перемещения в синтаксис C++ введены R-value ссылки, а также конструкторы перемещения и перемещающий оператор присваивания.

```
class D{
   public:
      int _numOfa;
      int* _a;
      D(): _a(nullptr), _numOfa(0){
      }
      void addNum(int a){
         if (_numOfa == 0) {
            _a = new int(a);
            _numOfa = 1;
         }else{
            int *tmp = new int[_numOfa];
            for(int i = 0, i < _numOfa, ++i){
               tmp[i] = _a[i];
            }
            delete[] _a;
            _a = tmp;
            ++_numOfa;
         }
      }
      D(D &a){
         this->_a = new int[a._numOfa];
         this->_numOfa = a._numOfa;
         for(int i = 0, i < _numOfa, ++i){
               this->_a[i] = a._a[i];
         }
      }
      D(D &&a){ //&& - по R-value ссылке
         this->_a = a._a;
         this->numOfa = a.numOfa;
         a._a = nullptr;
      }
         D& operator= ( D &&a){
                  this->_a = a._a;
                  this->numOfa = a.numOfa;
                  a._a = nullptr;
                  return *this;
      }

}
};
```

Билет №15 Компиляторы и интерпретаторы. Фазы компиляции.

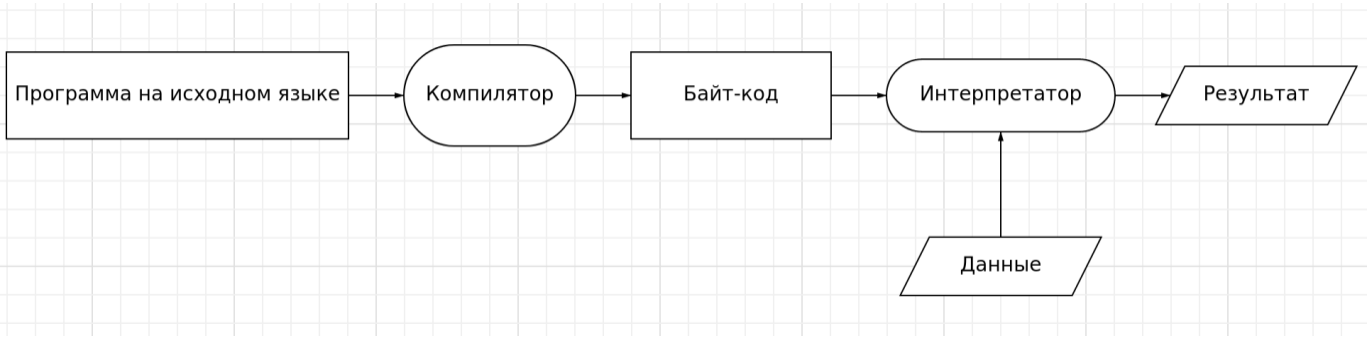
Компиляторы и интерпретаторы

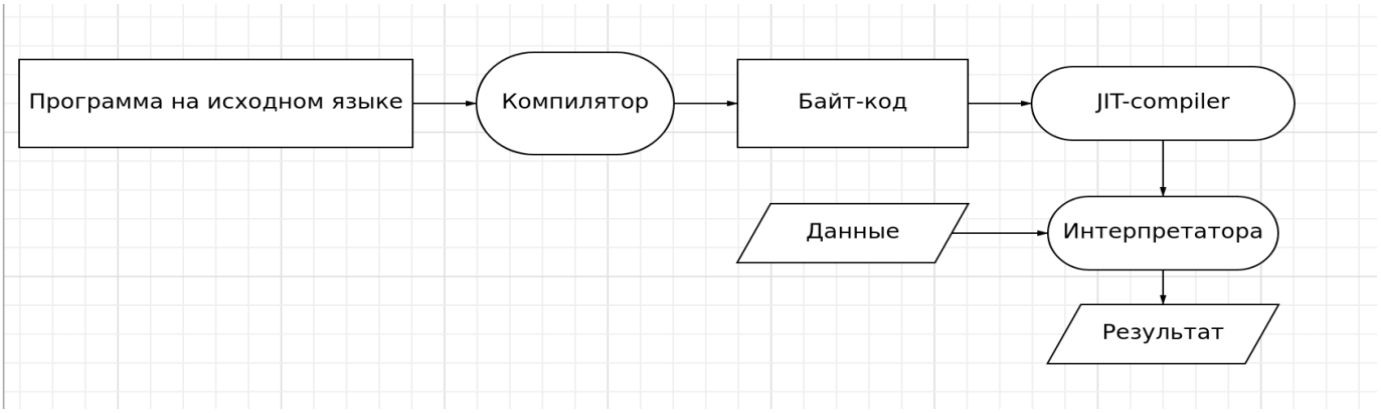


Интерпретатор переводит исходный код на команды процессора.

Компилятор выявляет ошибки, проводит анализ, генерирует код(перестройка в объектные программы).

Виртуальная машина:





Фазы компиляции

1. Лексический анализ (проверка правильности написания ключевых слов)
2. Синтаксический анализ (проверка на правильность логики построения кода)
3. Семантический анализ (построения последовательности выполнения действий)
4. Оптимизация
5. Генерация кода