

# Содержание

1 Билет 1	1
2 Билет 2	4
3 Билет 3	4
4 Билет 4	6
5 Билет 5	9
6 Билет 6	12
7 Билет 7	18
8 Билет 8	20
9 Билет 9	22
10 Билет 10	23
11 Билет 11	23
12 Билет 12	25
13 Билет 13	26
14 Билет 14	26
15 Билет 15	27
16 Билет 16	28

## 1 Билет 1

**Определение 1.1.** Подпрограммы в C++ называются функциями.

**Определение 1.2.** Блок — это набор функций, которые логически связаны между собой.

**Определение 1.3.** Код — это часть программы, которая выполняет действия, а данные представляют собой информацию, на которую направлены эти действия.

Объектно-ориентированное программирование характеризуется следующими тремя терминами.

**Определение 1.4.** Инкапсуляция — это механизм программирования, который связывает воедино код и данные, которые он обрабатывает, чтобы обезопасить их как от внешнего вмешательства, так и от неправильного использования.

**Определение 1.5.** Полиморфизм — это свойство, позволяющее использовать один интерфейс для целого класса действий.

**Определение 1.6.** Наследование — это процесс, благодаря которому один объект может приобретать свойства другого.

**Определение 1.7.** Объект — это конструкция, которая поддерживает инкапсуляцию.

**Определение 1.8.** Код называется закрытым, если он известен или доступен только частям этого объекта.

**Определение 1.9.** Код называется открытым, если он доступен любым другим частям программы.

**Определение 1.10.** Класс — это структура, которая содержит некоторый набор данных и также набор методов, которые могут выполнять работу с элементами этого класса.

**Определение 1.11.** Конструктор — это метод, отвечающий за то в каком состоянии будет создаваться класс при его объявлении в программе.

**Определение 1.12.** Деструктор — метод, отвечающий за уничтожение класса.

В языке C++ есть возможность создавать бинарные и унарные операции для заданных ранее объектов. Это называется переопределением (перегрузкой) операторов.

**Пример 1.** Введём операции сложения, вычитания и равенства для пар координат двух точек.

```
struct Point {  
    int x, y;  
    Point () {  
        x = 0;  
        y = 0;  
    }  
    Point (int a, int b) {  
        x = a;  
        y = b;  
    }  
    Point operator+ (const Point &b) {  
        Point res;  
        res.x = x+b.x;
```

```

    res.y = y+b.y;
    return res;
}
Point operator- (const Point &b) {
    Point res;
    res.x = x-b.x;
    res.y = y-b.y;
    return res;
}
void operator+= (const Point other) {
    *this = *this+other;
}
void operator-= (const Point other) {
    *this = *this-other;
}
void operator= (const Point &b) {
    x = b.x;
    y = b.y;
}
};

```

**Определение 1.13.** Исключение — объект произвольного типа данных, который передаёт информацию из места, в котором произошла ошибка, в код, который должен обработать эту ошибку

Ключевые слова:

1. *try* — функция, которая позволяет выделить в любом месте кода блок, который может генерировать исключения.
2. *catch* — функция, позволяющая находить исключения в блоке кода, который заключён в фигурных скобках.
3. *const* — функция, которая используется для указания, что переменная или метод не изменяет состояние объекта или не изменяет значение аргументов.
4. *static* — функция, которая используется для статических переменных, то есть переменных, которые инициализируются ровно 1 раз и не изменяются до конца работы программы.

5. *protected* — функция, которая используется для определения области видимости класса. В этом случае видны все элементы класса, а так же производные данного класса.
6. *friend* — функция, позволяющая взаимодействовать двум элементам двух разных закрытых классов друг с другом.
7. *template* — функция, позволяющая создать шаблон.

## 2 Билет 2

**Определение 2.1.** Стек — это некоторая динамическая структура, которая позволяет хранить, добавлять и изымать элементы по дисциплине *LIFO* (*Last In — First Out*).

**Определение 2.2.** Дек — это некоторая динамическая структура, которая позволяет хранить, добавлять и изымать элементы по дисциплине и из начала, и из конца.

**Определение 2.3.** Очередь — это некоторая динамическая структура, которая позволяет хранить, добавлять и изымать элементы по дисциплине *FIFO* (*First In — First Out*).

## 3 Билет 3

**Определение 3.1.** Структуру данных, обладающую следующими свойствами называют списком:

1. Элементы данных образуют линейную цепочку, то есть для каждого элемента существует "следующий" и "предыдущий" (за исключением первого и последнего элементов).
2. В каждый момент времени в этой линейной цепочке определена некоторая текущая позиция и разрешён доступ к элементу на этой позиции.
3. Положение текущей позиции можно изменить и тем самым получать доступ к значению любого элемента этой структуры.
4. Добавление и удаление элементов может происходить в окрестностях текущей позиции, причём эти операции не должны приводить к массовым перессылкам данных в памяти.

Различают два типа списков: однонаправленные и двунаправленные.

**Определение 3.2.** Однонаправленный список — список, в котором есть указатель только на следующий элемент.

**Определение 3.3.** Однонаправленный список — список, в котором есть указатель и на следующий, и на предыдущий элемент.

Для реализации двунаправленного списка нужно реализовать следующие операции:

1. Создать список
2. Уничтожить список
3. Добавить элемент справа
4. Удалить элемент справа
5. Добавить элемент слева
6. Удалить элемент слева
7. Выбрать элемент слева
8. Выбрать элемент справа
9. Сдвинуть указатель влево
10. Сдвинуть указатель вправо
11. Прочитать элемент справа
12. Прочитать элемент слева
13. Изменить элемент справа
14. Изменить элемент слева
15. Переместить указатель в начало
16. Переместить указатель в конец
17. Очистить список
18. Указатель в начале?
19. Указатель в конце?
20. Список пуст?

Для реализации однонаправленного списка нужно реализовать следующие операции:

1. Создать список
2. Уничтожить список
3. Добавить элемент справа
4. Удалить элемент справа
5. Выбрать элемент справа
6. Добавить текущий элемент
7. Удалить текущий элемент
8. Выбрать текущий элемент
9. Сдвинуть указатель вправо
10. Прочитать элемент справа
11. Изменить элемент справа
12. Прочитать текущий элемент
13. Изменить текущий элемент
14. Переместить указатель в начало
15. Очистить список
16. Указатель в конце?
17. Список пуст?

## 4 Билет 4

**Определение 4.1.** Дерево — связанный граф без циклов.

**Определение 4.2.** Расстоянием между вершинами  $A$  и  $B$  или длиной пути называется количество рёбер графа, содержащихся в связной цепочке рёбер, соединяющих  $A$  и  $B$ .

**Определение 4.3.** Вершина  $B$  называется потомком вершины  $A$ , если расстояние от  $A$  до  $B$  равно 1 и расстояние от вершины  $A$  до корня меньше чем расстояние о вершине  $B$  до корня.

**Определение 4.4.** Если вершина  $A$  находится на расстоянии  $k$  от корня, то говорят, что эта вершина находится на  $k$ -м уровне дерева.

**Определение 4.5.** Ветвью дерева назовём связную последовательность вершин, начинающуюся с корня и заканчивающуюся вершиной, не имеющей потомков.

**Реализация 4.1** (Произвольное дерево).

*template <typename T>*

```

struct Node {
    T value;
    vector<Node*> sons;
};

struct Tree {
    Node *root;
    Tree () {root == nullptr;}
};

void DFS (Node* v) {
    if (v == nullptr) {return;}
    for (int i = 0; i < (int) (v -> sons.size()); i++) {
        DFS(v -> sons[i]);
    }
}

```

**Определение 4.6.** Дерево, в котором у каждой вершины не больше двух потомков, называется бинарным.

**Реализация 4.2** (Бинарное дерево).

*template <typename T>*

```

struct Node {
    T value;
    Node *left, *right;
    Node(T x) {value = x; right = left = nullptr;}
};

struct BinTree {
    Node *root;
    BinTree () {
        root = nullptr;
    }
    ~BinTree () {
        destructor(root);
    }
};

```

```

        }
    }

void destructor (Node *v) {
    if (v != nullptr) {
        if (v -> left != nullptr) {destructor(v -> left);}
        if (v -> right != nullptr) {destructor(v -> right);}
    }
    delete v;
}

```

Существует 2 обхода произвольного дерева:

1. Обход сверху-вниз
2. Обход снизу-вверх

**Пример 2.** Пример обхода сверху-вниз произвольного дерева:

```

void Up_Down (Node *v) {
    if (!v) {return;}
    for (int i = 0; i < v -> sons.size(); i++) {
        Up_Down (v -> sons[i]);
    }
}

```

Существует 3 обхода бинарного дерева:

1. Обход сверху-вниз
2. Обход слева-направо
3. Обход снизу-вверх

**Реализация 4.3.**

```

void Up_Down (Node *pos) {
    if (!pos) {return;}
    Up_Down (pos -> left);
    Up_Down (pos -> right);
}

```

Остальные два алгоритма реализуются аналогично.

**Алгоритм 4.1.** 1. Стандартный обход от корня до всех вершин. Проверяем вершину  $u$ , пока левое поддерево не пусто, спускаемся влево. Если левое поддерево пусто или уже проверено, то смещаемся в правое поддерево.

2. Начинаем с самой левой вершины, затем, если у неё есть правое поддерево, то следующей вершиной будет самая левая вершина этого поддерева, или его корень, если такой нет. Если правого поддерева нет, то следующей вершиной является родитель данной, а дальше процесс повторяется для каждой последующей вершины.
3. Начинаем с самой левой вершины  $u$ , если есть правый потомок у родителя данной вершины, то переходим к самому левому из правых потомков данной вершины и повторяем итерацию, иначе поднимаемся вверх и повторяем итерации.

Любое произвольное дерево всегда можно представить в виде бинарного. Для этого расположим все потомки данной вершины в одностороннем списке в том же порядке, в котором они добавляются в это дерево. В данной вершине будем хранить указатель на первый элемент этого списка и на следующий элемент в списке.

**Пример 3.** *template <typename T>*

```
struct Node {
    T value;
    Node () {*next, *down;}
};

void Up_Down (Node *v) {
    if (!v) {return;}
    v = v -> down;
    while (v) {
        Up_Down (v);
        v = v -> next;
    }
}
```

## 5 Билет 5

**Определение 5.1.** Ключ вершины дерева — значение, которое однозначно вычисляется из информации, хранящейся в данной вершине, и связано отношением порядка с ключами данных вершин.

**Определение 5.2.** Бинарным деревом поиска называется бинарное дерево, в котором ключ любой вершины не меньше ключа любой вершины левого поддерева и строго меньшее ключа любой вершины правого поддерева.

**Реализация 5.1.**

```

struct Node {
    int id;
    Node *left, *right;
    Node (int x) {id = x; left = right = nullptr;}
};

void destructor (Node* v) {
    if (v != nullptr) {
        if (v -> right != nullptr){
            destructor (v -> right);
        }
        if (v -> left != nullptr) {
            destructor (v -> left);
        }
    }
    delete v;
}

struct BinTree {
    Node *root;
    BinTree () {
        root = nullptr;
    }
    ~BinTree () {
        destructor (root);
    }
    Node* _BinErase (Node *v, int x) {
        if (v == nullptr) {return nullptr;}
        if (v -> id < x) {
            v -> right = _BinErase (v -> right, x);
            return v;
        }
        if (v -> id > x) {
            v -> left = _BinErase (v -> left, x);
            return v;
        }
    }
}

```

```

    }
    else {
        Node *left = v -> left;
        Node *right = v -> right;
        delete v;
    }
}

int _BinFind (Node* v, int x) {
    if (v == nullptr) {return 0;}
    if (v -> id == x) {return 1;}
    if (v -> id > x) {return _BinFind(v -> left, x);}
    else {return _BinFind(v -> right, x);}
}

Node* _BinInsert (Node* v, int x) {
    if (v == nullptr) {
        Node* nnode = new Node(x);
        nnode -> id = x;
        return nnode;
    }
    if (v -> id == x) {return v;}
    if (v -> id < x) {
        v -> right = _BinInsert(v -> right, x);
        return v;
    }
    else {
        v -> left = _BinInsert(v -> left, x);
        return v;
    }
}
int BinFind (int x) {
    return _BinFind(root, x);
}
int BinInsert (int x) {
    if (BinFind(x)==1) {return 0;}
    root = _BinInsert(root, x);
    return 1;
}

```

```

int BinErase (int x) {
    if (root == nullptr) {return 0;}
    if (BinFind(x) == 0) {return 0;}
    root = _BinErase(root, x);
    return 1;
}

```

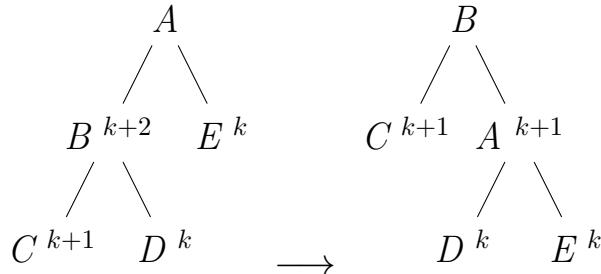
## 6 Билет 6

**Определение 6.1.** Глубиной дерева назовём максимальную длину ветви этого дерева.

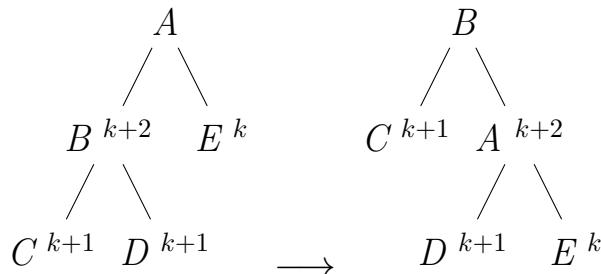
**Алгоритм 6.1.** Мы хотим добиться того, чтобы в каждой вершине модуль разности глубины левого и правого поддеревьев не превышал 1. Для этого при добавлении или удалении элемента будем совершать один из следующих поворотов.

1. Малый правый поворот:

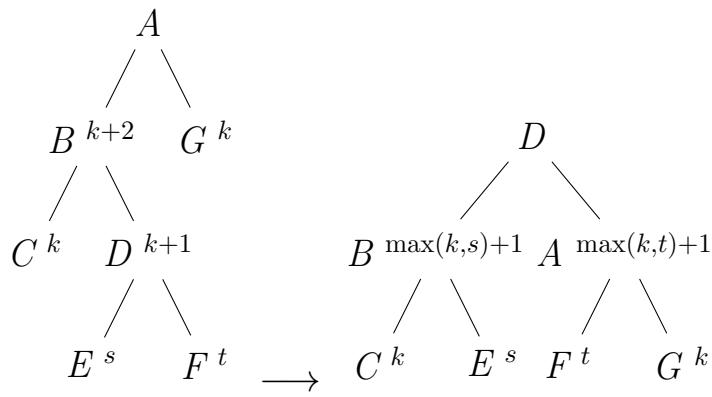
Если вес левого поддерева левого поддерева больше веса правого поддерева левого поддерева:



Если вес левого поддерева левого поддерева равен весу правого поддерева левого поддерева:

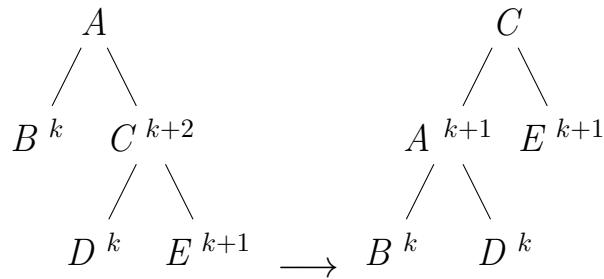


2. Большой правый поворот:

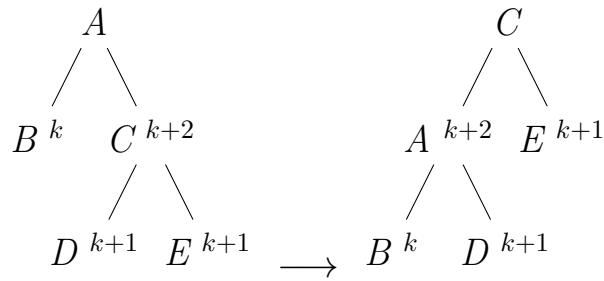


### 3. Малый левый поворот:

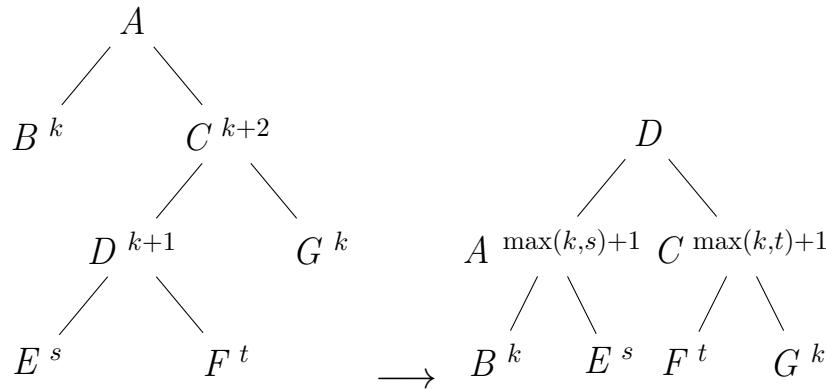
Если вес правого поддерева правого поддерева больше веса левого поддерева правого поддерева:



Если вес правого поддерева правого поддерева равен весу левого поддерева левого поддерева:



### 4. Большой левый поворот:



**Реализация 6.1.**

```
struct Node {
    int id;
    Node *left, *right;
```

```

    Node (int x) { id = x; left = right = nullptr; }
};

void destructor (Node* v) {
    if (v != nullptr) {
        if (v -> right != nullptr) {
            destructor (v -> right);
        }
        if (v -> left != nullptr) {
            destructor (v -> left);
        }
    }
    delete v;
}

struct AVL_Tree {
    Node *root;
    AVL_Tree () {
        root = nullptr;
    }
    ~AVL_Tree () {
        destructor (root);
    }
    Node* small_left (Node *v) {
        Node* A = v -> r;
        v -> r = A -> l;
        A -> l = v;
        return A;
    }
    Node* big_left (Node *v) {
        Node *B, *C;
        B = v -> r;
        C = B -> l;
        B -> l = C -> r;
        v -> r = C -> l;
        C -> l = v;
        C -> r = B;
        return C;
    }
}

```

```

Node* small_right (Node *v) {
    Node* A = v -> l;
    v -> l = A -> r;
    A -> r = v;
    return A;
}
Node* big_right (Node *v) {
    Node *B, *C;
    B = v -> l;
    C = B -> r;
    v -> l = C -> r;
    B -> r = C -> l;
    C -> l = B;
    C -> r = v;
    return C;
}
void insert (int x) {
    if (_find(root, x)!=1) {root = _insert(root, x);}
}
Node* _insert (Node *v, int x) {
    if (v == nullptr) {
        Node* nnode = new Node(x);
        nnode -> id = x;
        return nnode;
    }
    if (v -> id > x) {
        v -> l = _insert(v -> l, x);
        if (h(v -> l)-h(v -> r) == 2) {
            if (h(v -> l -> l) >= h(v -> l -> r)) {
                v = small_right(v);
            }
            else {
                v = big_right(v);
            }
        }
    }
}

```

```

        return v;
    } else {
        v -> r = _insert(v -> r, x);
        if (h(v -> l)-h(v -> r) == -2) {
            if (h(v -> r -> l) <= h(v -> r -> r)) {
                v = small_left(v);
            }
            else {
                v = big_left(v);
            }
        }
    }
    return v;
}
int erase (int x) {
    if (_find(root, x)==0) {return 0;}
    else {root = _erase(root, x); return 1;}
}
Node* _erase (Node* v, int x) {
    int t;
    if (v -> id == x) {
        if (v -> l == nullptr && v -> r == nullptr) {
            return nullptr;
        }
        if (v -> l == nullptr && v -> r != nullptr) {
            return v -> r;
        }
        if (v -> r == nullptr && v -> l != nullptr) {
            return v -> l;
        }
    }
    if (v -> id == x) {
        if (v -> l -> r != nullptr) {
            v -> id = max_id(v -> l);
            v -> l = _erase(v -> l, v -> id);
        }
    }
}

```

```

else {
    v -> id = v -> l -> id;
    v -> l = _erase(v -> l, v -> id);
}
if (h(v -> l) - h(v -> r) == -2) {
    v = small_left(v);
}
return v;
}
if (v -> id < x) {
    v -> r = _erase(v -> r, x);
    if (h(v -> l) - h(v -> r) == 2) {
        if (h(v -> l -> l) >= h(v -> l -> r)) {
            v = small_right(v);
        } else {
            v = big_right(v);
        }
    }
    return v;
}
if (v -> id > x) {
    v -> l = _erase(v -> l, x);
    if (h(v -> l) - h(v -> r) == -2) {
        if (h(v -> r -> l) <= h(v -> r -> r)) {
            v = small_left(v);
        } else {
            v = big_left(v);
        }
    }
}
return v;
}
int _find(Node* v, int x) {
    if (v == nullptr) {return 0;}
    if (v -> id == x) {return 1;}
    else if (v -> id > x) {return _find(v -> l, x);}
    else {return _find(v -> r, x);}
}

```

```

    }
    int find (int x) {return _find (root , x);}
};

```

**Определение 6.2.** Дерево назовём идеально сбалансированным, если для любой его вершины глубина левого поддерева совпадает с глубиной правого поддерева.

**Теорема 6.1.** Длина произвольного AVL дерева из  $N$  элементов не превосходит  $\frac{3}{2} \log_2(N)$ .

*Доказательство.* Рассмотрим дерево, в котором левое поддерево каждой вершины имеет глубину на 1 больше глубины правого поддерева этой вершины (данное дерево называется деревом Фибонacci). Обозначим за  $N(k)$  количество вершин в таком дереве, имеющем длину  $k$ .

$$N(0) = 0, N(1) = 1, N(k) = N(k - 1) + N(k - 2) + 1$$

Индукция по  $k$ .

База индукции:  $N(1) = 1 \leq \frac{3}{2} \cdot (2^1 - 1) = \frac{3}{2}$

Шаг индукции:  $N(k) = N(k - 2) + N(k - 1) + 1 \leq \frac{3}{2} \cdot (2^{k-2} - 1 + 2^{k-1} - 1) + 1 = 9 \cdot 2^{k-3} - 2 = 2^k + 2^{k-3} - 2 \leq 2^k + 2^{k-1} - \frac{3}{2} \leq \frac{3}{2} \cdot (2^k - 1)$

Из этого следует, что для любого сбалансированного дерева глубина не превышает  $\log_2(\frac{2}{3}N)$ , то есть поиск элемента имеет сложность  $O(\log_2(N))$ . Причём добавление и удаление элемента также можно совершить за логарифмическое время, так как нужно будет совершить не больше  $\log_2(N)$  поворотов, чтобы сбалансировать новое дерево. ■

## 7 Билет 7

**Определение 7.1.** Красно-чёрным деревом называется бинарное дерево поиска со следующими свойствами:

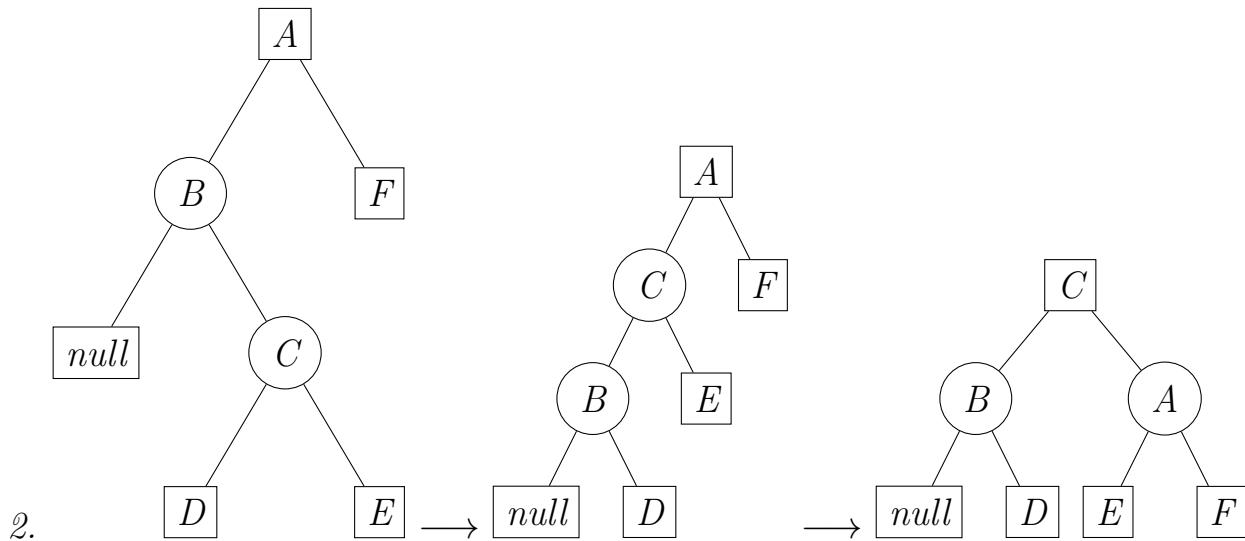
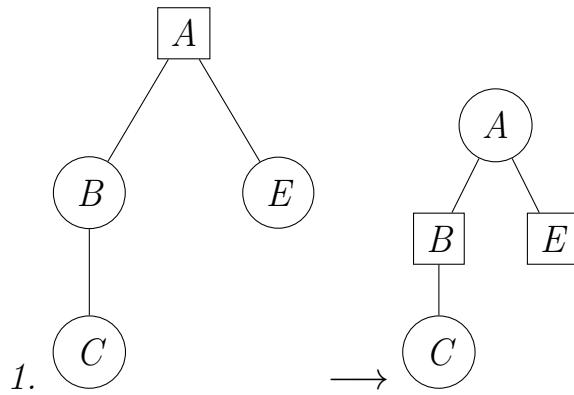
1. Каждая вершина имеет цвет (красный или чёрный), причём корень всегда чёрный.
2. Красная вершина может иметь только чёрных потомков.
3. В любой ветке от корня до концевой вершины содержится одинаковое число чёрных вершин.

**Теорема 7.1.** Глубина красно-чёрного бинарного дерева из  $N$  элементов не превосходит  $2 \log_2(N + 1)$ .

*Доказательство.* Пусть красно-чёрное дерево имеет глубину  $k$ . В силу свойства 2 длина любой ветви от корня не меньше  $\frac{k}{2} \implies N$  (количество вершин) не меньше  $2^{\frac{k}{2}} - 1 \implies \frac{k}{2} \leq \log_2(N + 1) \implies k \leq 2 \log_2(N + 1)$ . ■

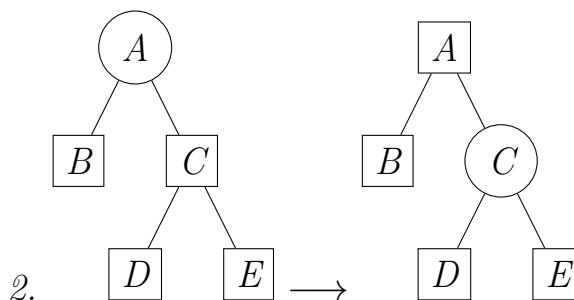
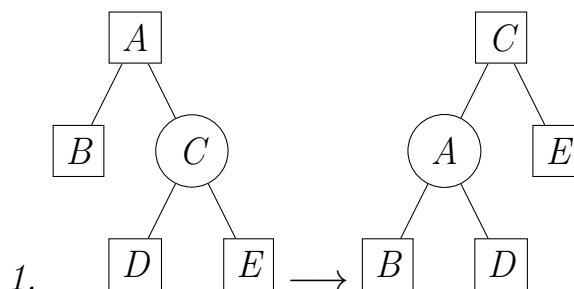
**Алгоритм 7.1.** Нужно реализовать следующие четыре поворота:

Добавление элемента: добавляемый элемент перекрашиваем в красный цвет и при необходимости совершать одно из следующих преобразований:



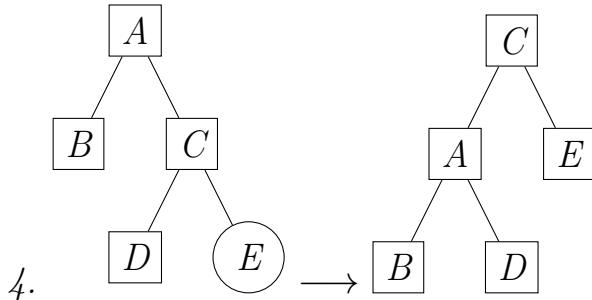
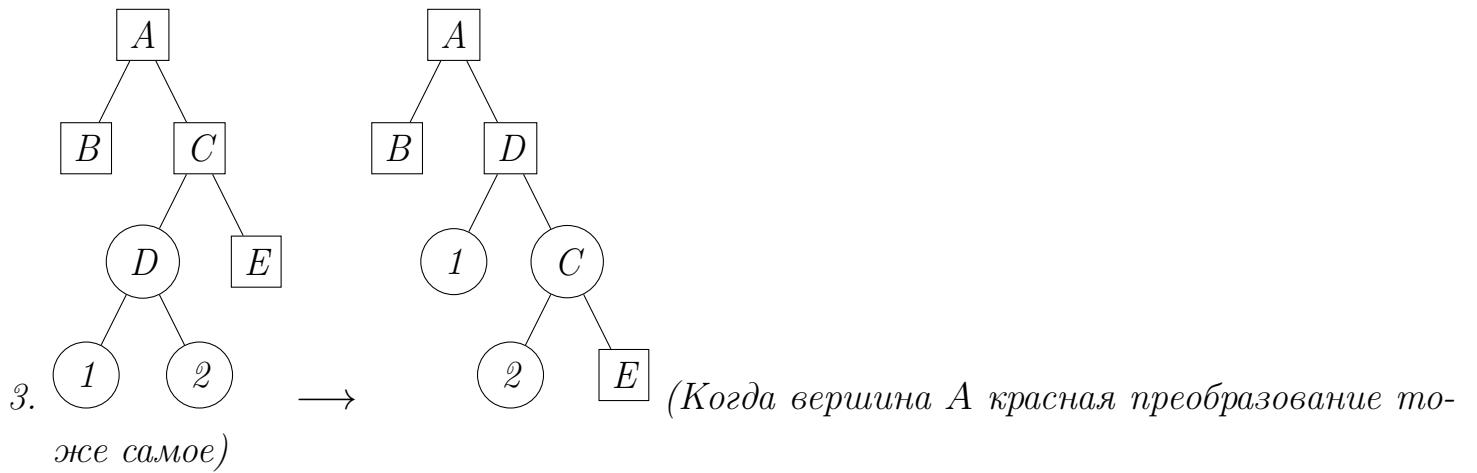
3. Справа симметрично

Удаление элемента:



И переходим к вершине A.

Если A чёрная, то перекрашиваем только C и переходим к вершине A.



Вершины  $A$  и  $D$  могут иметь и красный цвет, но только не обе вместе, тогда преобразование тоже самое.

## Реализация 7.1.

## 8 Билет 8

**Определение 8.1.**  $B$ -деревом порядка  $n$  называется древовидная структура, удовлетворяющая следующим условиям:

1. Вершинами являются массивы, способные вместить  $2n$  элементов.
2. В каждой вершине, являющейся массивом, элементы упорядочены по возрастанию ключей.
3. Каждая вершина, кроме корня, содержит не меньше  $n$ , но и не больше  $2n$  элементов.
4. Вершина, содержащая  $k$  элементов содержит ровно  $k+1$  потомка, либо является концевой вершиной.
5. Если вершина имеет  $k$  элементов, то ключи всех элементов поддерева  $i$ -го потомка меньше ключа элемента и больше ключа  $i-1$ -го элемента.
6. Все концевые вершины лежат на одном уровне дерева.

**Алгоритм 8.1** (Добавление элемента). 1. Найдём концевую вершину, в которую можно добавить элемент.

2. Если число элементов до добавления в этой вершине меньше  $2n$ , то просто добавляем элемент и процесс завершён.
3. Если число элементов до добавления в этой вершине ровно  $2n$ , то создаётся новая вершина, в исходной вершине остаются первые  $n$  элементов, а в новую добавляются последние  $n$  элементов из данной. Средний элемент добавляется в родительскую вершину, если она уже заполнена, то процесс повторяется и т.д. Если в результате мы достигли корня, то корень аналогично разделяется на две вершины по тому же принципу, а средний элемент переносится в новый корень.

**Алгоритм 8.2** (Удаление элемента). 1. Если удаляемый элемент находится в концевой вершине, то очевидно.

2. Если удаляемый элемент не в концевой вершине, то заменяем этот элемент максимальным элементом из поддерева, вершиной которого является левая ссылка удаляемого элемента.
3. Если в результате этого в концевой вершине останется  $n - 1$  элемент, то обединим множество элементов этой концевой вершины  $A$ , элемент  $y$ , левая ссылка от которого указывает на эту вершину и множество элементов концевой вершины  $B$ , на которую указывает правая ссылка элемента  $y$ . Теперь разделим это множество на 3 составляющие: средний элемент поместим в родительскую вершину вместе с элементом  $y$ , все элементы слева переместим в вершину  $A$ , а оставшиеся в вершину  $B$ .
4. Если в вершине  $B$  до этого преобразования было ровно  $n$  элементов, то все  $2n$  элементов перемещаются в вершину  $A$ , вершина  $B$ , элемент  $y$  удаляется из родительской вершины вместе со ссылкой на вершину  $B$ .
5. Если в родительской вершине число элементов равно  $n - 1$ , то повторяем операцию.
6. Если в результате таких операций мы дойдём до корня, то удаляем корень, а вершину, с которой была совершена последняя операция, становится новым корнем.

**Теорема 8.1.** Сложность поиска в  $B$ -дереве из  $N$  элементов и имеющего порядок  $p$  равна  $O(\log_p(N) \log_2(n))$ .

*Доказательство.* Глубина этого дерева равна  $O(\log_n(N))$ , в каждой вершине бинарным поиском либо находим элемент, либо указатель на потомка, что возможно за  $O(\log_2(n))$ , то есть поиск осуществляется за  $O(\log_n(N) \cdot \log_2(n))$ . ■

**Теорема 8.2.** Сложность алгоритмов добавления и удаления элемента из  $B$ -дерева равна  $O(\log_n(N) \log_2(n) + n)$ .

*Доказательство.* Из предыдущей теоремы имеем, что для того чтобы найти удаляемый элемент нужно  $O(\log_n(N) \cdot \log_2(n))$  времени, а для того чтобы сбалансировать полученное после удаления/добавления элемента дерево понадобится не более  $n$  операций, описанных выше, то есть сложность алгоритма добавления или удаления элемента составляет  $O(\log_n(N) \log_2(n) + n)$ . ■

## 9 Билет 9

**Определение 9.1.** Пусть  $N_{\max}$  — максимальное число в подмножестве натуральных чисел. Целочисленный массив размера  $[N_{\max}/32] + 1$  назовём битовым множеством, если каждое число  $k$  хранится в нём как битовое число, где 1 стоит на позиции с номером  $k \% 32$  и в ячейке с номером  $[k/32]$ .

**Пример 4.** Пусть множество состоит из элементов 0, 1, 2, 3, 4, 5, тогда  $[N_{\max}/32 + 1] = 1$ , тогда, например число 3 будет находиться в ячейке с номером 0, и на позиции, где в битовой записи 1 сдвинута на 3 позиции влево.

Операции:

1. Добавление элемента: с помощью побитовой дизъюнкции числа  $A[i]$  и числа  $1 << (a \% 32)$ .
2. Удаление элемента: аналогично, только вместо побитовой дизъюнкции выполняется побитовая конъюнкция, причём вместо элемента  $1 << (a \% 32)$  используется элемент  $\sim (1 << (a \% 32))$  (операция  $\sim$  — побитовое отрицание, то есть  $\sim 010 = 101$ ).
3. Поиск элемента: проверка, что  $A[a/32] \& (1 << (a \% 32)) != 0$ .
4. Пересечение: побитовая конъюнкция.
5. Объединение: побитовая дизъюнкция.
6. Инвертирование:  $\sim a$  (побитовое отрицание).

## 10 Билет 10

**Определение 10.1.** Пусть  $M$  — множество элементов,  $N \subset M$ ,  $f : M \rightarrow \{0, 1, \dots, p-1\}$  — некоторая функция, тогда функция  $f$  разбивает множество  $M$  на классы эквивалентности. Данная функция  $f$  называется хеш-функцией. А алгоритм называется хешированием.

**Примеры 1.** 1. Функция, сопоставляющая фамилии человека её первую букву.

2. Хеш-функция для строк:  $x$  — значение символа,  $p$  — простое число  $M$  — простое число (довольно большое).  $h = (h \cdot p + x) \% M$ , значение хеш-функции — сумма по всем символам строки.

**Алгоритм 10.1** (Метод списков). Пусть даны списки  $S_1, \dots, S_5$ , тогда сопоставим каждому  $S_i$  пересечение  $i$ -ого класса эквивалентности с данным множеством  $N$ . Затем вычисляем значение хеш-функции от интересующего нас элемента и переходим к изучению на добавление, поиск, удаление в соответствующий список.

Само сопоставление каждому списку значение хеш-функции осуществляется с помощью хеш-таблицы, которая представляет собой массив, в котором  $k$ -ый элемент содержит указатель на начало  $k$ -ого списка.

## 11 Билет 11

**Алгоритм 11.1** (Метод проб). Пусть  $N$  содержит не более  $n$  элементов,  $p$  — число возможных значений хеш-функции, несколько большее  $n$ . Пусть  $f$  — хеш-функция, составим массив из  $p$  ячеек и будем туда добавлять элементы данного множества. Положим  $h[k] = x$ , где  $k = f(x)$ .

1. Добавление элемента  $y$ :

- Если  $\exists x = h[k] = y$ , то просто добавляем элемент.
- Если  $\exists x = h[k] = y$ , то осуществляем такую же проверку для ячейки  $h[k+1]$ . При достижении ячейки  $h[p-1]$  переходим к ячейке  $h[0]$  и продолжаем проверку.

2. Удаление элемента  $y$ :

Пусть  $h[s] = y$ .

- Если следующая ячейка пустая, то просто удаляем элемент.

- (b) Если следующая ячейка не пустая, то ищем первое число со значением хеш-функции, которое меньше  $s$  и которое принадлежит данной цепочке. Если такого числа нет, то удаляем  $h[s]$ .
- (c) Если есть, то удаляем  $h[s]$  и перемещаем найденное число в ячейку  $s$  и повторяем процесс для ячейки с этим числом.

### 3. Поиск элемента:

- (a) Начинаем проверку с ячейки  $h[k]$ , где  $k = f(x)$ . Если она занята элементом, не равным исскомому, то переходим к ячейке  $h[k + 1]$  и повторяем проверку.
- (b) Далее осуществляем проверку пока не найдём данное число или пустую ячейку. Во втором случае делаем вывод, что число не найдено.

**Утверждение 11.1.** Вероятность расположить число с  $k$ -ой попытки равна  $\frac{p-m}{p-k+1} \cdot \prod_{i=0}^{k-2} \frac{m-i}{p-i}$ , где  $m$  — число занятых ячеек.

*Доказательство.*  $q_1 = \frac{p-m}{p}$ ,  $q_2 = \frac{p}{m} \cdot \frac{p-m}{p-1}$ ,  $q_3 = \frac{m}{p} \cdot \frac{m-1}{p-1} \cdot \frac{p-m}{p-2}$  и т.д. ■

**Теорема 11.1.** Количество проб, которое можно считать как случайную величину, принимющую значение  $k$  с вероятностью  $q_k = \frac{p-m}{p-k+1} \cdot \prod_{i=0}^{k-2} \frac{m-i}{p-i}$ , равно  $\frac{1}{1-a}$ , где  $a = \frac{m}{p+1}$ .

**Лемма 11.1.**  $\sum_{k=2}^{m+1} \frac{k \cdot (p-k)!}{(m-k+1)! \cdot (p-1)!} = \frac{2p-(m-1)}{(m-1)! \cdot (p-m+1) \cdot (p-m)}$

*Доказательство.*  $\frac{k \cdot (p-k)!}{(m-k+1)! \cdot (p-1)!} = \frac{A}{(m-k+1)!} + \frac{B}{(p-k)!}$   
 $(p-1)! \cdot A + (m-k+1)! \cdot B = k \cdot (p-k)!$

Возьмём  $A = \frac{1}{(m-1)! \cdot (p-m)}$ ,  $B = -\frac{1}{(m-1)! \cdot (p-m+1)}$ .

Тогда  $\frac{A}{(m-k+1)!} + \frac{B}{(p-k)!} = \frac{\frac{1}{(m-1)! \cdot (p-m)}}{(m-k+1)!} + \frac{-\frac{1}{(m-1)! \cdot (p-m+1)}}{(p-k)!} =$  ■

*Доказательство.* Ожидаемое количество проб вычисляется как математическое ожидание величины  $q_k$ , то есть  $\sum_{k=1}^{m+1} k q_k = \frac{p-m}{p} + \sum_{k=2}^{m+1} \left( k \cdot \frac{p-m}{p-k+1} \cdot \prod_{i=0}^{k-2} \frac{m-i}{p-i} \right) =$   
 $= (p-m) \left( \frac{1}{p} + \sum_{k=2}^{m+1} \left( \frac{k}{p-k+1} \cdot \frac{m!}{(m-k+1)!} \cdot \frac{(p-k+1)!}{p} \right) \right) = (p-m) \cdot \left( \frac{1}{p} + \sum_{k=2}^{m+1} \left( \frac{k \cdot m! \cdot (p-k)!}{(m-k+1)! \cdot p!} \right) \right) =$   
 $= \frac{p-m}{p} \cdot (1 + m! \cdot \sum_{k=2}^{m+1} \left( \frac{k \cdot (p-k)!}{(m-k+1)! \cdot (p-1)!} \right))$

По лемме имеем

$$\frac{p-m}{p} \cdot (1 + m! \cdot \sum_{k=2}^{m+1} \left( \frac{k \cdot (p-k)!}{(m-k+1)! \cdot (p-1)!} \right)) = \frac{p-m}{p} \cdot \left( 1 + \frac{m! \cdot (2p-(m-1))}{(m-1)! \cdot (p-m+1) \cdot (p-m)} \right) =$$

$$= \frac{p-m}{p} \cdot \frac{(p-m)^2 + (p-m) + m(2p-m+1)}{(p-m+1) \cdot (p-m)} = \frac{p^2 + p}{p \cdot (p-m+1)} = \frac{p+1}{p-m+1}$$

Обозначим  $a = \frac{m}{p+1}$ , тогда формула имеет вид:  $\frac{\frac{m}{a}}{\frac{m}{a}-m} = \frac{1}{1-a}$ . ■

## 12 Билет 12

**Определение 12.1.** Пусть  $M$  — множество всех возможных элементов,  $t$  — подмножество заданных фиксированных ключей. Хеш-функция  $h(x)$  называется совершенной, если  $\forall x_1, x_2 \in M, x_1 \neq x_2 \implies h(x_1) \neq h(x_2)$ .

**Определение 12.2.** Хеш-функция  $h(x)$  называется минимальной, если  $h : M \rightarrow \{0, 1, \dots, |t|\}$ .

**Определение 12.3.** Пусть на множестве ключей введено отношение порядка " $<$  тогда если  $h(x)$  — хеш-функция,  $x_1 < x_2 \implies h(x_1) < h(x_2)$  для любых  $x_1, x_2 \in M$ , то говорят, что  $h$  сохраняет порядок.

**Алгоритм 12.1** (Построение минимальной совершенной хеш-функции). Будем искать функцию  $h(x)$  в виде:  $g(f_1(x)) + g(f_2(x))$ , где  $f_1(x), f_2(x)$  — хеш-функции на основе случайных таблиц, а  $g$  — функция, которую предстоит построить.

1. Выберем  $n = c \cdot |t|$ , где  $c > 2$  — некоторая константа. Возьмём две случайные хеш-функции  $f_1, f_2$ , с диапазоном значений  $\{0, \dots, n-1\}$ .
2. Каждому ключу  $x$  сопоставим пару  $(f_1(x), f_2(x))$ . Теперь построим граф, рёбрами которого будут пары  $(f_1(x), f_2(x))$ , каждое такое ребро будет соединять вершины  $f_1(x), f_2(x)$ .
3. Если в таком графе будет цикл, то генерируем новые функции  $f_1, f_2$ . И начинаем процесс заново.

**Утверждение 12.1.** Вероятность получить ациклический граф имеет асимптотику  $p = \sqrt{\frac{c-2}{c}}$ , где  $n = c \cdot |t|$ .

**Утверждение 12.2.** Ожидаемое число попыток, которое потребуется для получения ациклического графа, составляет  $\sqrt{\frac{c}{c-2}}$ , то есть в среднем потребуется не более двух попыток.

*Доказательство.* Вычислим математическое ожидание:

$$\sum_{k=1}^{\infty} kp(1-p)^{k-1} = p \cdot \left(-\sum_{k=1}^{\infty}(1-p)^k\right)' = p \cdot \left(\frac{p-1}{p}\right)' = \frac{1}{p} = \sqrt{\frac{c}{c-2}}$$

## 13 Билет 13

**Определение 13.1.** Хеш-функция  $h(s)$  называется циклической, если  $h(s[i+1 \dots i+m]) = (h(s[i \dots i+m-1]) - s[i] \cdot p^{m-1}) \cdot p + s[i+m]$ , где  $p$  простое.

**Определение 13.2.** Хеш-функция вида:  $P_s(x) = (s_0 \cdot x^{n-1} + s_1 \cdot x^{n-2} + \dots + s_{n-1})$  — называется полиномиальной хеш-функцией.

**Алгоритм 13.1** (Рабина-Карпа). Пусть  $s$  — строка,  $t$  — её подстрока,  $h$  — хеш-функция.

1. Вычисляем значение хеш-функции от подстроки  $t$ ,  $h(t) = t_0$  (работает за  $O(m)$ , где  $m$  — длина подстроки) и вычисляем значение  $h(s[0 \dots m-1]) = s_0$ , также за  $O(m)$ . Вычислим  $p^m$  для некоторого простого  $p$ .
2. Для каждого  $i$  от 0 до  $n-m$  сравниваем значения  $t_0$  и  $s$ , где  $s$  — значение хеш-функции на  $[i \dots i+m-1]$ .  $s = (p \cdot s - h(i) \cdot p^m + h(i+m)) \% mod$ , где  $mod$  — некоторое большое простое число.

Данный алгоритм будет работать за  $O(n+m)$ .

**Алгоритм 13.2** (Кнута-Морриса-Пратта). 1.

## 14 Билет 14

**Определение 14.1.** Граф — множество вершин и рёбер.

**Определение 14.2.** Ориентированный граф — граф, в котором на каждом ребре задано направление. Иначе граф называется неориентированным.

Два способа хранения:

1. Хранение графа в виде двумерного массива. В нём элемент  $a_{ij} = 1$ , если существует ребро между вершинами  $i$  и  $j$ .
2. Хранить граф как список смежности (массив списков, где каждый список хранит соседей вершины (т. е. указатели на них)).

**Замечание 1.** Неориентированный граф можно хранить как верхнетреугольную матрицу смежности.

**Алгоритм 14.1** (Обход в ширину). 1. Каждую вершину обозначим 0, как не посещённую.

2. Стартовую вершину переобозначим 1.

3. Для каждой вершины, являющейся соседом уже прочитанной вершины  $x$ , если она обозначена 1, то переходим к следующей, иначе повторяем операцию для всех её соседей, а затем обозначаем эту вершину 1 (как прочитанную).

**Алгоритм 14.2** (Обход в глубину). Пусть все вершины имеют индекс 0 (не посещены).

1. Если стартовая вершина совпадает с конечной, то процесс завершается и возвращает true.
2. Если вершина имеет индекс не 0, то процесс завершается и возвращает false.
3. Изменим индекс данной вершины на 1.
4. Приемим данную функцию ко всем соседям данной вершины.
5. Изменим индекс данной вершины на 2.

## 15 Билет 15

**Алгоритм 15.1** (Максимальный поток (алгоритм Форда-Фалкерсона)). На каждом ребре графа напишем число человек, которое пропускает данное ребро. Теперь перебираём все возможные пути от стартовой вершины до конечной. Для каждого пути найдём минимальное число людей, которое он пропускает, и преобразуем данный график, вычитя из числа над каждым ребром данного пути данное минимальное число.

**Алгоритм 15.2** (Максимальное паросочетание (алгоритм Куна)). Начинаем с пустого множества  $M$  пар. Разделим множество на два класса. Для каждой вершины первого класса, начиная с первой, смотрим с какой вершиной из второго класса у неё есть общее ребро. Выбираем такую вершину, которая не была выбрана ранее. Если таких вершин не оказалось, то проверяем, можно ли выбрать другую вершину на одном из предыдущих шагов алгоритма. Каждую найденную пару добавляем в множество  $M$ . Количество пар в множестве  $M$  и есть максимальное паросочетание.

**Определение 15.1.** Оставное дерево — связный подграф данного графа, являющийся деревом и имеющий столько же вершин сколько и у данного графа.

**Определение 15.2.** Оставное дерево называется минимальным, если сумма длин его рёбер минимальна.

**Алгоритм 15.3** (Минимальное оставное дерево (алгоритм Прима)). Выберем произвольную вершину в данном графе. Теперь для каждой соседней вершины запишем в неё вес, равный длине соответствующего ребра. Теперь для каждой из оставшихся не пройденных вершин повторим процесс. Если длина какого ребра, соединяющего уже пройденную

вершину и исследоваемую, окажется меньше известного веса этой пройденной вершины, то перезапишем его. Иначе не учитываем. Таким образом получим минимальное оставное дерево.

## 16 Билет 16

**Определение 16.1.** Контейнер — програмная реализация, предназначенная для хранения данных в памяти в течении некоторого времени.

Система предписаний:

1. Создать контейнер.
2. Расположить данные.
3. Удалить данные.
4. Очистить контейнер.
5. Завершить работу.

**Пример 5.** Функции *malloc* и *free*.