

## Math

1.

Найдем производные, подставим в изначальное уравнение, сравним с нулем. Если не равно нулю, тогда данное уравнение не является решением диффура.

$$\begin{aligned}y &= xe^{-x} \\y'' - 2y' + y &= 0 \\[10pt]y' &= e^{-x} + (-xe^{-x}) = e^{-x} - xe^{-x} \\[10pt]y'' &= -e^{-x} - (e^{-x} - xe^{-x}) = xe^{-x} - 2e^{-x} \\[10pt]& xe^{-x} - 2e^{-x} - 2(e^{-x} - xe^{-x}) + xe^{-x} = \\& = 2xe^{-x} + 2xe^{-x} - 2e^{-x} - 2e^{-x} = \\& = 4xe^{-x} - 4e^{-x} \neq 0\end{aligned}$$

2.

Это однородный диффур второго порядка с постоянными коэффициентами, решается через составление характеристического уравнения, нахождением лямбда, и подстановкой в общее решение уравнения.

$$y'' - 2y' + y = 0$$

Составим характерист. ур-е

$$\lambda^2 - 2\lambda + 1 = 0$$
$$(\lambda - 1)^2 = 0$$
$$\lambda_1 = 1$$

Общее решение:

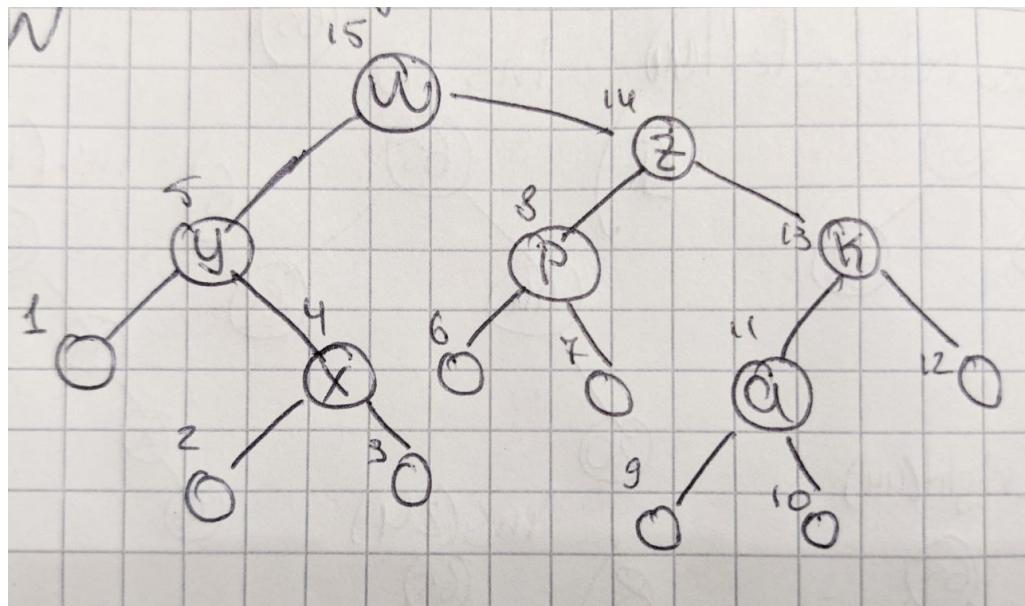
$$y = C_1 e^{\lambda_1 x} + C_2 e^{\lambda_2 x}$$

отвейн:

$$y = C_1 e^x + C_2$$

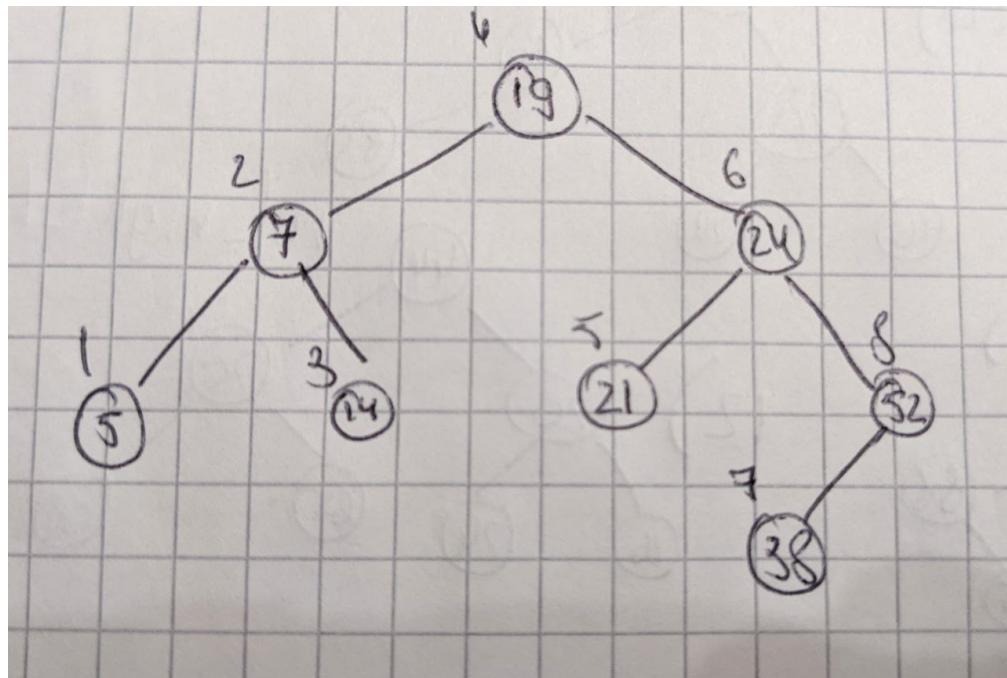
## Data structures and Algorithms

1. LRN or postorder



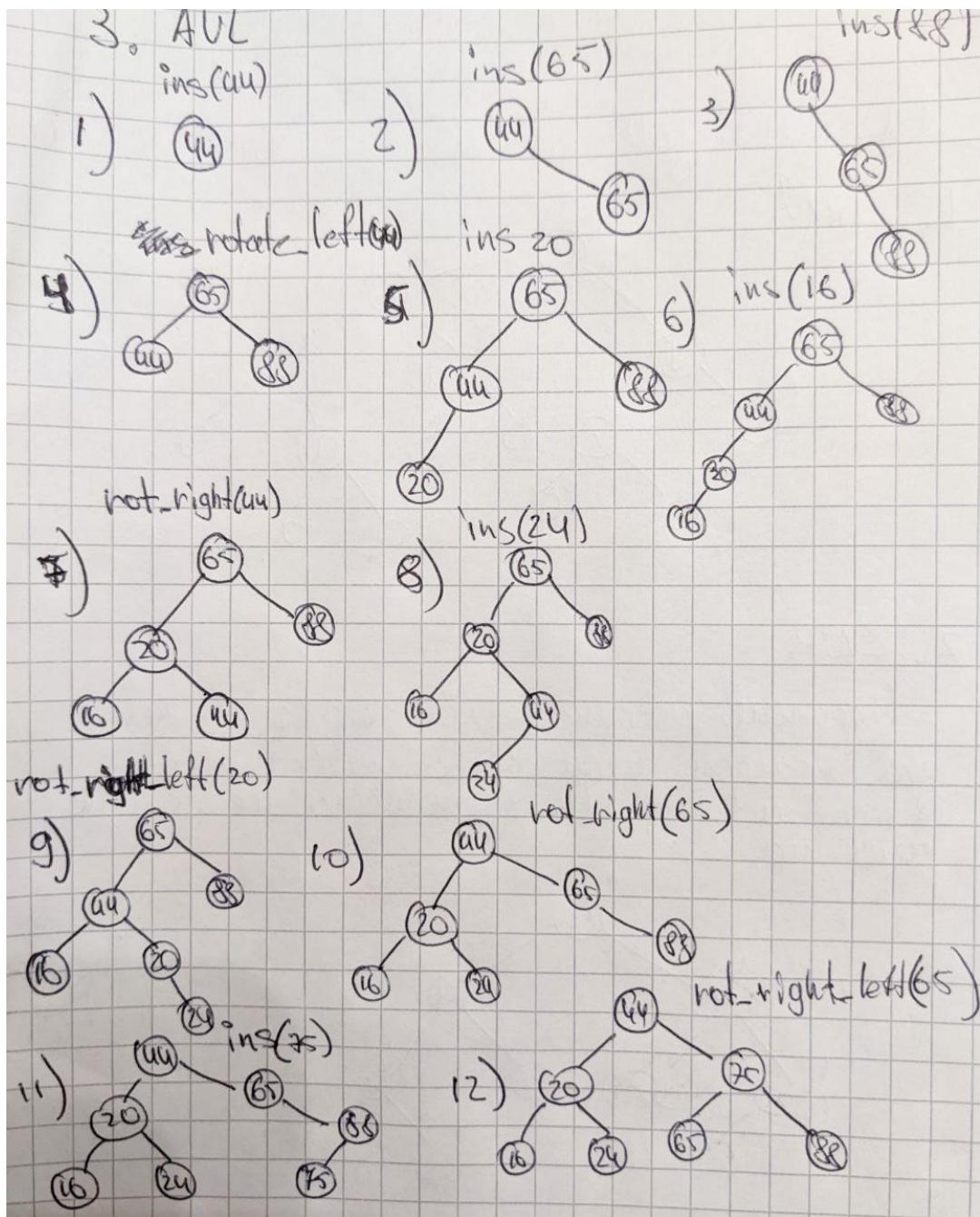
2. LNR or inorder

Идеально сбалансированное дерево - это дерево, в котором для каждого узла количество узлов левом и правом поддереве отличаются не больше чем на 1.

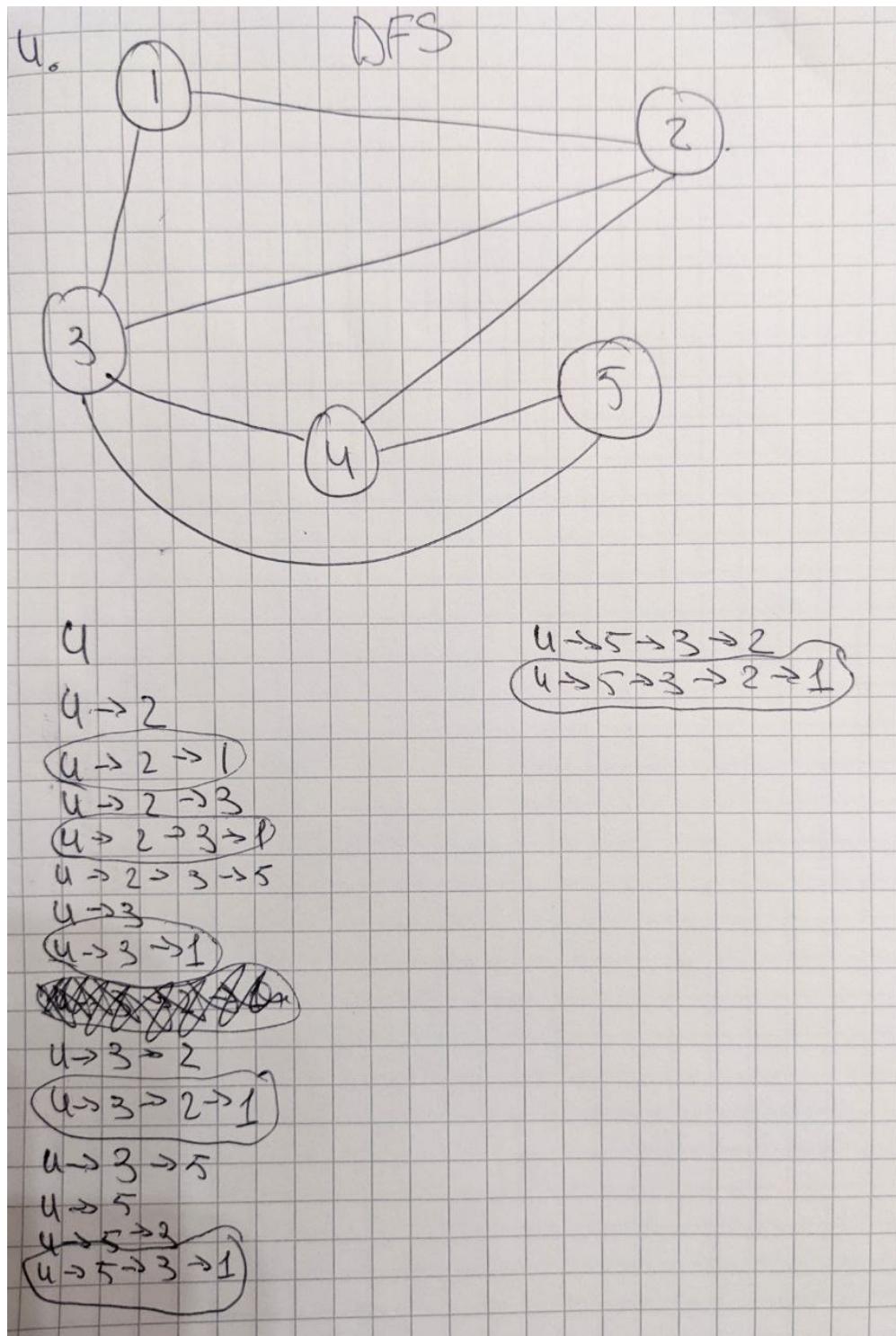


3.

AVL дерево - это самобалансирующееся бинарное дерево поиска, где баланс достигается с помощью некоторых поворотов вершин вокруг себя. Повороты бывают 4 типов: правый, левый, правый-левый и левый-правый, которые запускаются для каждой вершины в ветке после добавления новой вершины, которая нарушает "идеальность" этого дерева.



#### 4. DFS или поиск в глубину



5.

Сортировка Шелла - это метод сортировки массивов путем замены местами стоящих элементов на некоторой заданной длине друг от друга, в порядке возрастания. То есть алгоритм, используя заданную последовательность расстояний, сравнивает элементы между собой и расставляет их в правильном порядке.

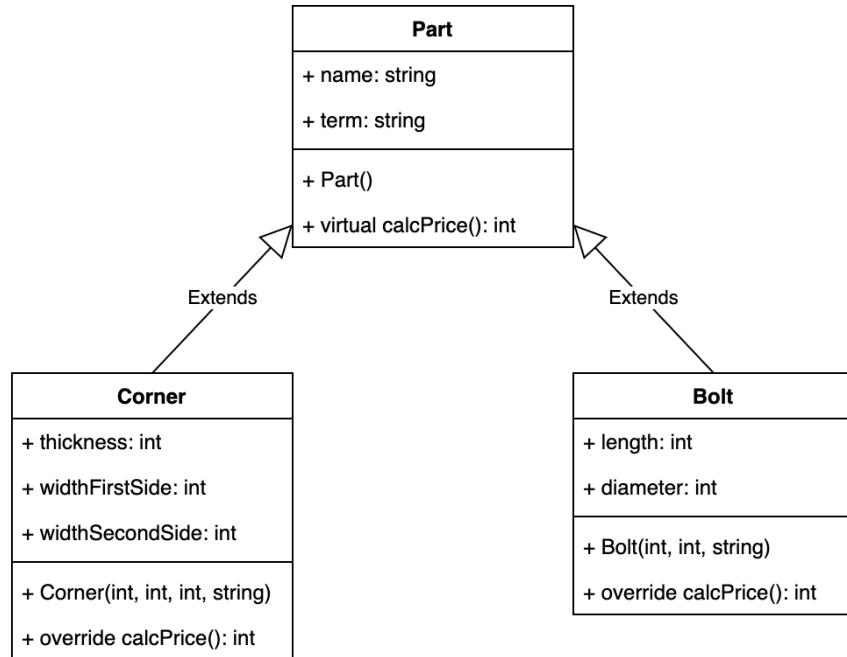
$n=8 \Rightarrow 4, 2, 1$	45 29 16 124 67 35 12 38
4	45 29 12 124 67 35 16 38
-	45 29 12 38 67 35 16 124
2	12 29 45 35 67 38 16 124
-	12 29 45 35 66 38 67 124
1	12 16 29 35 45 38 67 124
	12 16 29 35 38 45 67 124

Здесь я выбрал стандартную последовательность расстояний  $n/2, n/4, n/8 \dots$

Общее число сравнений равно 22.

# OOP

## 1.1. UML



## 1.2. C++ code

```
class Part {
public:
    string name;
    string term;

    Part(string name, string term) {
        this->name = name;
        this->term = term;
    }

    virtual int calcPrice() = 0;
};

class Bolt : public Part {
public:
    int length;
    int diameter;
```

```
Bolt(int length, int diameter, string term) : Part("Bolt", term) {
    this->length = length;
    this->diameter = diameter;
}

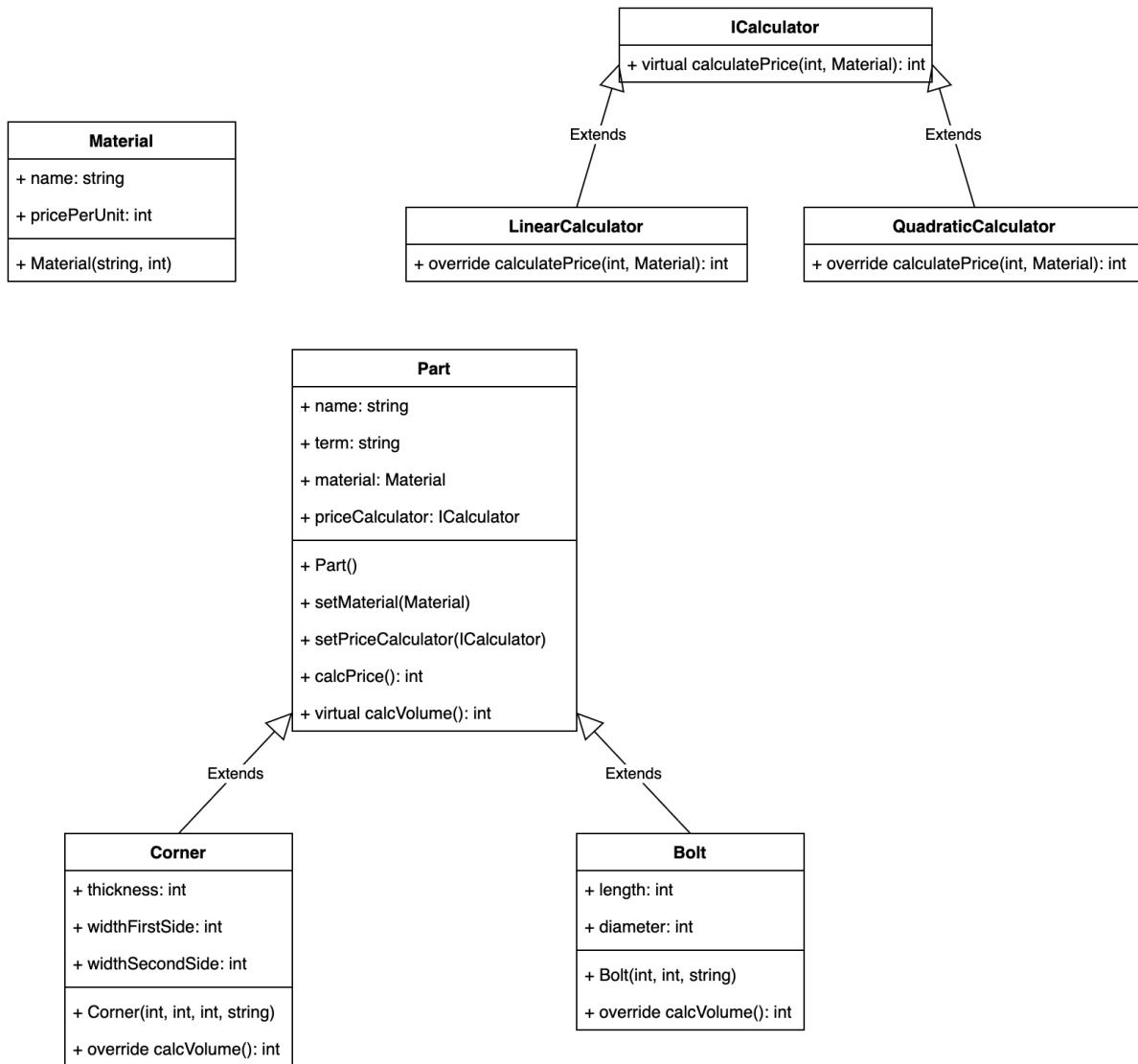
int calcPrice() override {
    return diameter * length;
}
};

class Corner : public Part {
public:
    int thickness;
    int widthFirstSide;
    int widthSecondSide;

    Corner(int thickness, int widthFirstSide, int widthSecondSide, string
term) : Part("Corner", term) {
        this->thickness = thickness;
        this->widthFirstSide = widthFirstSide;
        this->widthSecondSide = widthSecondSide;
    }

    int calcPrice() override {
        return thickness * widthFirstSide * widthSecondSide;
    }
};
```

## 2.1. UML



## 2.2. C++ code

```

class Material {
public:
    string name;
    int pricePerUnit;

    Material() {}

    Material(string name, int pricePerUnit) {
  
```

```

        this->name = name;
        this->pricePerUnit = pricePerUnit;
    }
};

class ICalculator {
public:
    virtual int calcPrice(int volume, Material* material) = 0;
};

class LinearCalculator : public ICalculator {
public:
    int calcPrice(int volume, Material* material) override {
        return volume * material->pricePerUnit;
    }
};

class QuadraticCalculator : public ICalculator {
public:
    int calcPrice(int volume, Material* material) override {
        return volume * volume * material->pricePerUnit;
    }
};

class Part {
public:
    string name;
    string term;
    Material* material;
    ICalculator* priceCalculator;

    Part(string name, string term) {
        this->name = name;
        this->term = term;
    }

    void setMaterial(Material* material) {
        this->material = material;
    }

    void setPriceCalculator(ICalculator* priceCalculator) {
        this->priceCalculator = priceCalculator;
    }
};

```

```
    }

    int calcPrice() {
        return priceCalculator->calcPrice(calcVolume(), material);
    }

    virtual int calcVolume() = 0;
};

class Bolt : public Part {
public:
    int length;
    int diameter;

    Bolt(int length, int diameter, string term) : Part("Bolt", term) {
        this->length = length;
        this->diameter = diameter;
    }

    int calcVolume() override {
        return diameter * length;
    }
};

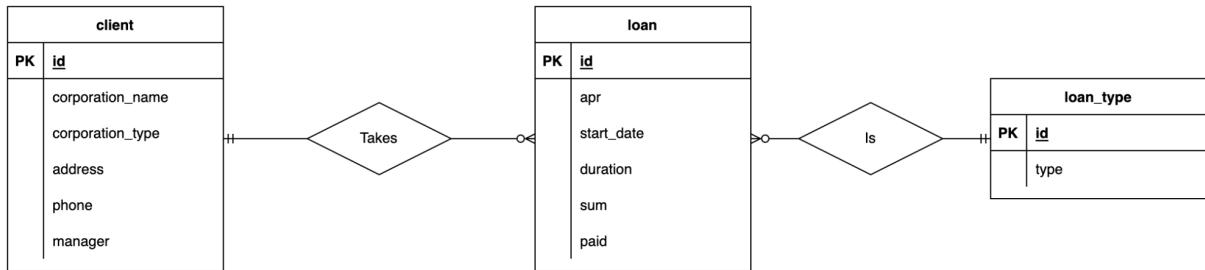
class Corner : public Part {
public:
    int thickness;
    int widthFirstSide;
    int widthSecondSide;

    Corner(int thickness, int widthFirstSide, int widthSecondSide, string
term) : Part("Corner", term) {
        this->thickness = thickness;
        this->widthFirstSide = widthFirstSide;
        this->widthSecondSide = widthSecondSide;
    }

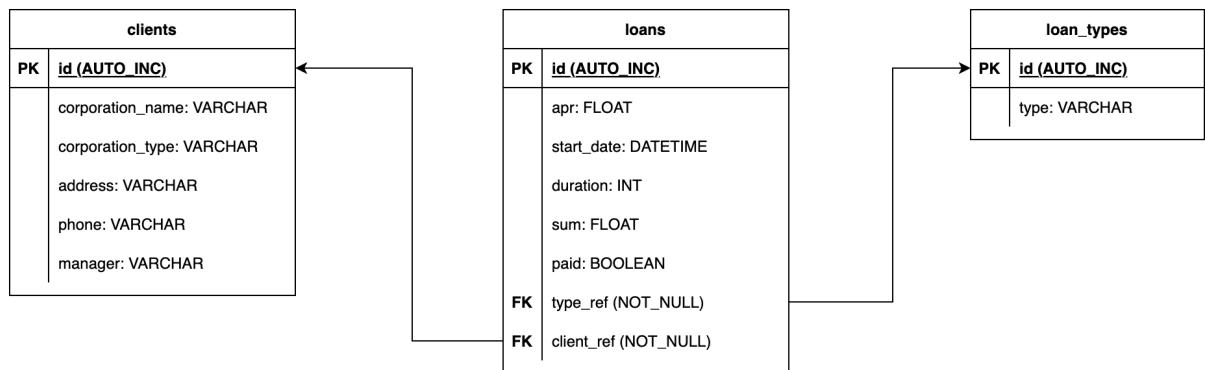
    int calcVolume() override {
        return thickness * widthFirstSide * widthSecondSide;
    }
};
```

# Databases

## 1. Conceptual model



## 2. Actual model



## 3.

```

SELECT
    clients.corporation_name AS "client name",
    loans.sum AS "loan sum",
    loans.start_date AS "Start date",
    loan_types.type AS "Loan type",
    loans.apr as "APR",
    DATE_ADD(loans.start_date, INTERVAL loans.duration DAY) as "End
    date"
FROM loans
INNER JOIN clients ON loans.client_ref = clients.id
INNER JOIN loan_types ON loans.type_ref = loan_types.id
WHERE loans.paid = FALSE;
  
```

4.

```
SELECT DISTINCT
    clients.corporation_name AS "client name",
    clients.phone AS "phone number",
    clients.manager AS "manager"
FROM loans
INNER JOIN clients ON loans.client_ref = clients.id
WHERE loans.duration >= 365;
```

5.

```
SELECT
    loan_types.type as "Loan type",
    sum(loans.sum) as "Annual loan"
FROM loans
INNER JOIN loan_types ON loans.type_ref = loan_types.id
WHERE DATE_ADD(loans.start_date, INTERVAL 365 DAY) > NOW()
GROUP BY type_ref;
```

6.

Ключ отношения - это некоторый уникальный идентификатор определенной строки в некоторой таблице (это может быть как один столбец, так и комбинация из нескольких). Поскольку в таблице может быть достаточно большое количество данных, которые, к тому же, могут повторяться, ключи нужны для уникальной разметки этих данных и быстрого к ним доступа. Собственно ключи не должны повторяться и должны быть уникальными.

7.

Транзакция - это последовательность выполнения некоторых операций над одной БД, которые либо все будут в итоге применены, либо полностью отброшены в случае ошибки. Транзакция - это очень важный элемент безопасности баз данных, ведь в случае ошибки записи в одну таблицу, запись в другие таблицы просто так не останавливается, что может привести к хранению недостоверных данных и разным уязвимостям. Транзакции должны быть атомарными и изолированными.