# T(n) и точная оценка

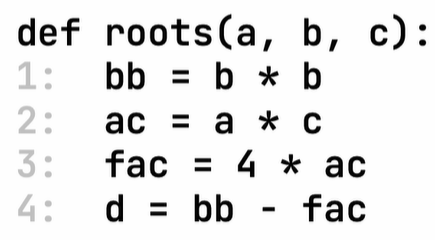
Время вычисления - временная сложность

память для вычисления - пространственная сложность

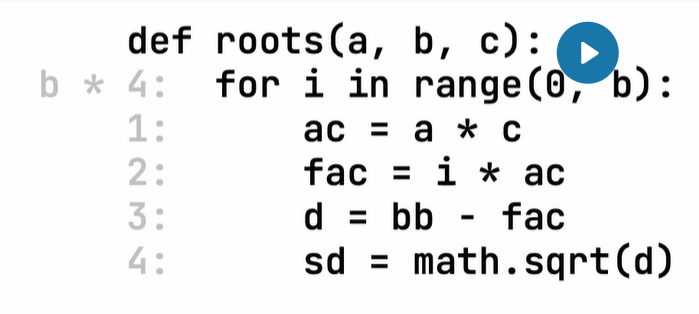
Оценка алгоритма:

количество инструкций - **точная оценка алгоритма**

(4 инструкции на слайде)



При использовании циклов количество инструкций умножается на число итераций



При этом если колво итераций цикла зависит от входного значения b, то вводится понятие функции времени T(n).

в случае с циклом for i in range(a): T(n) = na - линейная зависимость (алгоритм линейно зависит от параметра b)

Соответсвенно при вложенном цикле:

for i in range(a):

for j in range(a):

\*n инструкций\*

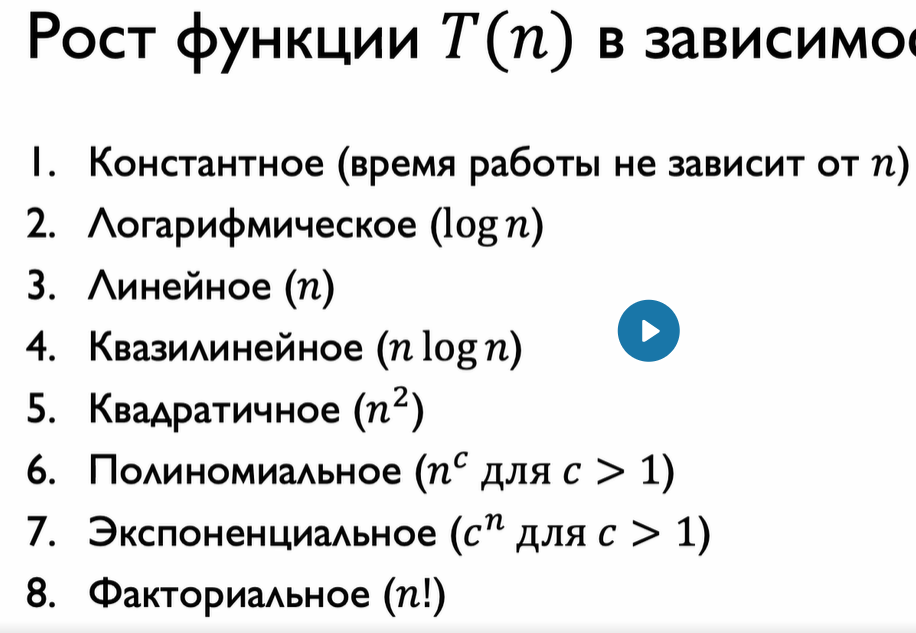
T(n) = n\*а\*а - квадратичная зависимость

**T(n) = n\*a\*a - это и есть вычисление точной оценки временной сложности алгоритма (кол-ва инструкций).**

Замечние:

Эффективность алгоритмов (Т(n)) (кол во инструкций) влияет на время выполнения задачи (секунды) гораздо более сильно, что повышение производительности комьютера (инструкций/секунду).

разные типы T(n) подразделяют



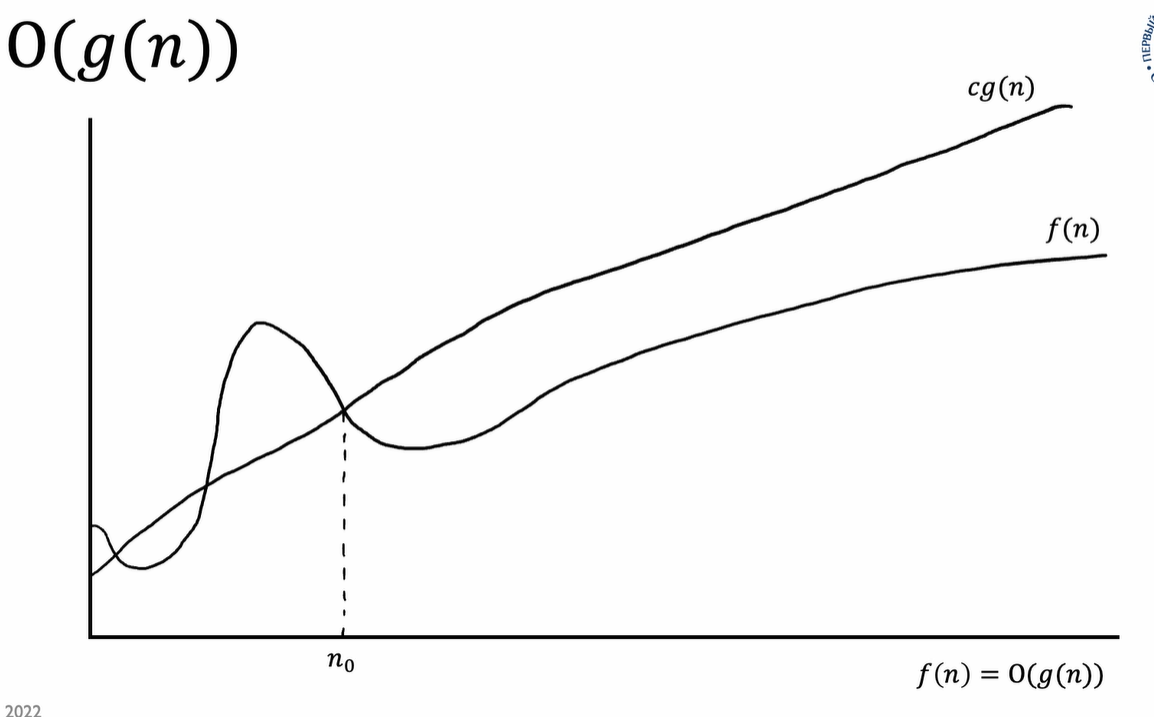
# Асимптотическая оценка

Ключевые концепции:

* Большое O - асимптотически верхняя граница T(n).

O(f(x)) - это совокупность функций, лежащих в пределах

0<= < O( f(x) ) < c1\*f(x) , x > x0



* + Эта нотация используется для описания верхней границы роста времени работы алгоритма.
  + Она показывает, как время выполнения алгоритма *максимально* растет с увеличением размера входных данных.
  + Обычно используется для описания худшего случая.
* Ω (Омега) - асимптотически нижняя граница T(n)

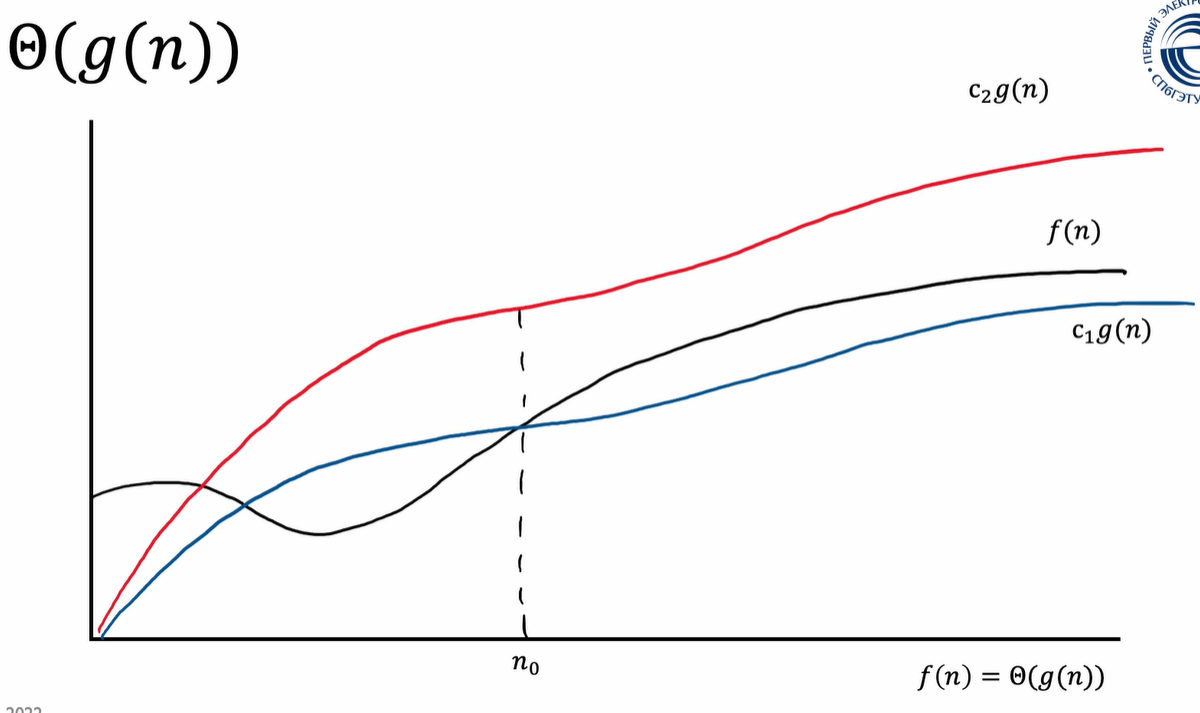
Ω(f(x)) - это совокупность функций, лежащих в пределах

0<= c1\*f(x) < Ω( f(x) ) , x > x0

* + Эта нотация используется для описания нижней границы роста времени работы алгоритма.
  + Она показывает, как время выполнения алгоритма *минимум* растет с увеличением размера входных данных.
  + Обычно используется для описания лучшего случая.
* Θ (Тэта):

Θ(f(x)) - это совокупность функций, лежащих в пределах

0<= c1\*f(x) < Θ( f(x) ) < c2\*f(x) , x > x0



* + Эта нотация используется для описания точного роста времени работы алгоритма.
  + Она показывает, как время выполнения алгоритма *точно* растет с увеличением размера входных данных.
  + Обычно используется для описания среднего случая.

# Вероятностный анализ T(n)

Требуется узнать сколько времени займёт выполнение алгоритма и понятно, что для разных входных данных время будет отличатся:

давайте разберем понятие “лучшего” и “худшего” времени работы алгоритма на примере сортировки пузырьком.

Сортировка пузырьком - это простой алгоритм сортировки, который сравнивает соседние элементы и меняет их местами, если они стоят не в правильном порядке.

Пример:

Представьте, что у нас есть список: [5, 2, 4, 6, 1, 3].

* Сортировка пузырьком будет проходить так:
  + Проход 1:
    - Сравниваются 5 и 2, меняются местами. Список становится: [2, 5, 4, 6, 1, 3]
    - Сравниваются 5 и 4, меняются местами. Список становится: [2, 4, 5, 6, 1, 3]
    - … и так далее до конца списка.
  + Проход 2:
    - Проводится аналогично первому проходу, но уже для отсортированного частично списка.
  + И так далее, до тех пор, пока весь список не будет отсортирован.

Лучший случай:

* Лучший случай для сортировки пузырьком - это когда список уже отсортирован.
* Алгоритму понадобится один проход, чтобы убедиться, что все элементы уже стоят на своих местах.
* Время работы алгоритма в этом случае будет линейным (O(n)), так как он просто проходит по всем элементам один раз.

Худший случай:

* Худший случай - это когда список отсортирован в обратном порядке.
* Алгоритму придется выполнить n - 1 проходов, чтобы отсортировать список.
* На каждом проходе алгоритм будет сравнивать и менять местами пары элементов, что приведет к квадратичному времени работы (O(n^2)).

Когда нельзя определить какой набор данных даст лучший или худший случай, то применяют вероятностный анализ (время в среднем).

Среднее время T(n) определяется как

₸(n) = CУММ( Са(x) \* P(x) ), где (формула математического ожидания

случайной величины)

Ca(х) - время затрачиваемое алгоритмом а при входных данных х.

P(x) - вероятность наступления таких входных данных

# Оценка рекурсивных алгоритмов

Рекурсивные алгоритмы - это алгоритмы, которые вызывают сами себя внутри своего определения.

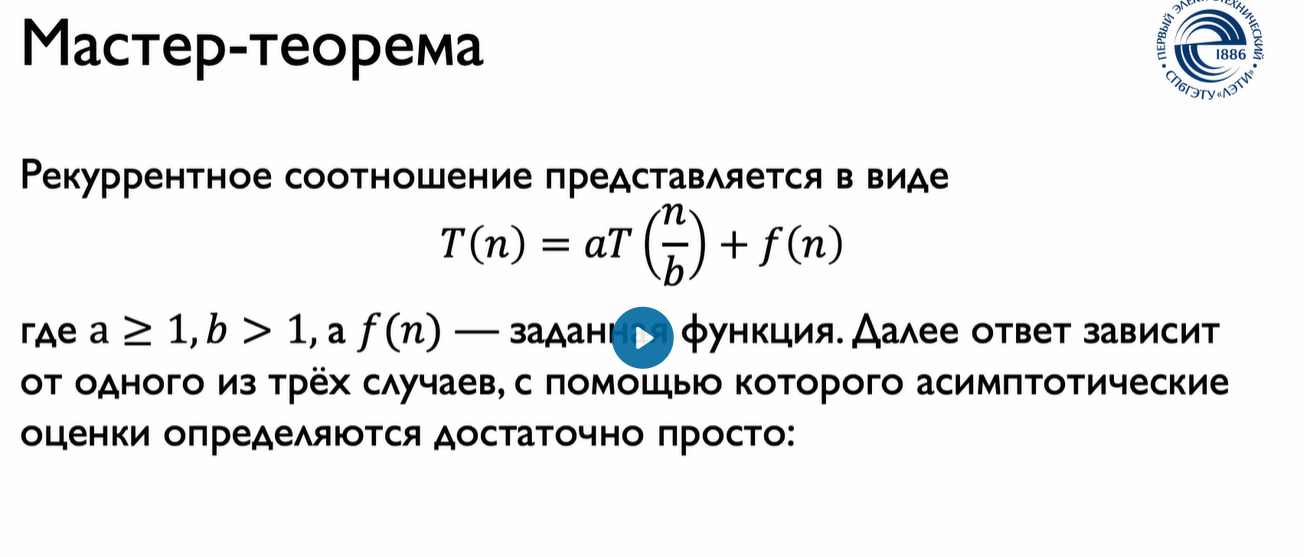
Рекуррентные соотношения - это уравнения, которые описывают время работы рекурсивного алгоритма в зависимости от размера входных данных.

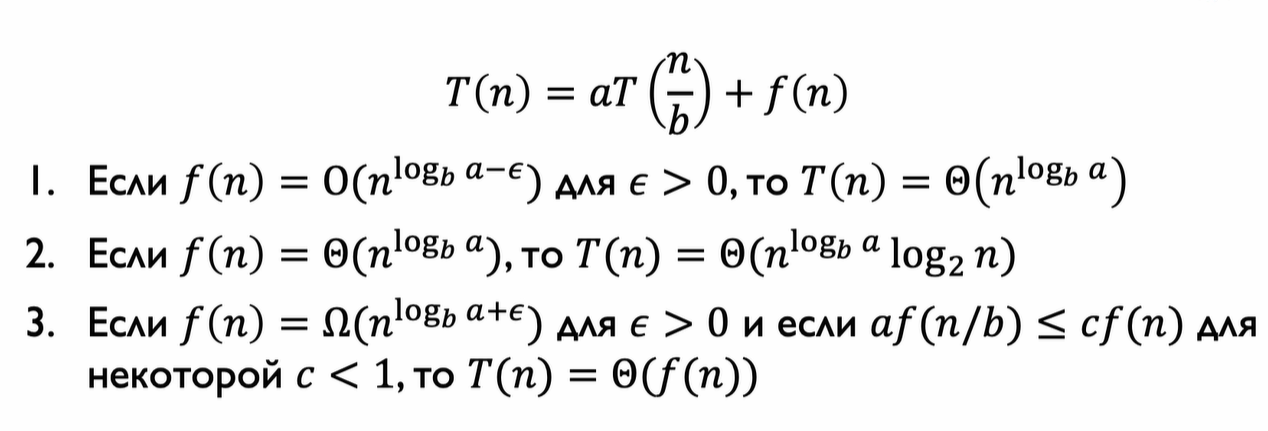
Задача: найти общую формулу, которая описывает время работы рекурсивного алгоритма при любом размере входных данных.

Слайде перечислены три основных метода решения рекуррентных соотношений:

* Метод подстановки:
  + Этот метод предполагает, что мы можем “угадать” решение и затем доказать его по индукции.
  + Этот метод часто используется для простых рекуррентных соотношений.
* Дерево рекурсии:
  + Этот метод визуализирует рекурсивные вызовы алгоритма в виде дерева.
  + Позволяет разбить время работы алгоритма на части и проанализировать их.
  + Хорошо подходит для визуализации и понимания алгоритма, но может быть сложным для больших деревьев.
* Мастер-теорема:
  + Это мощный инструмент для решения широкого класса рекуррентных соотношений.
  + Она предоставляет формулу для вычисления асимптотической сложности алгоритма.
  + Требует выполнения определенных условий для применения.

Принцип мастер-теоремы, любая временная сложность рекурентного алгоритма (T(n)) может быть вычислена как





Пример:

Представим рекурсивный алгоритм, который вычисляет факториал:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n-1)

Рекуррентное соотношение для этого алгоритма:

* T(0) = 1
* T(n) = T(n-1) + 1

# Структуры данных

## Линейные структуры данных

Структура данных - способ хранить данные, чтобы облегчить чтение и модификацию этих данных

Абстрактный тип данных - это тип данных, который характеризуется только операциями доступными над объектом этого типа данных. Информация о хранении этого типа данных в памяти и операции доступные над типом - все что требуется знать пользователю для работы с абстрактным типом данных

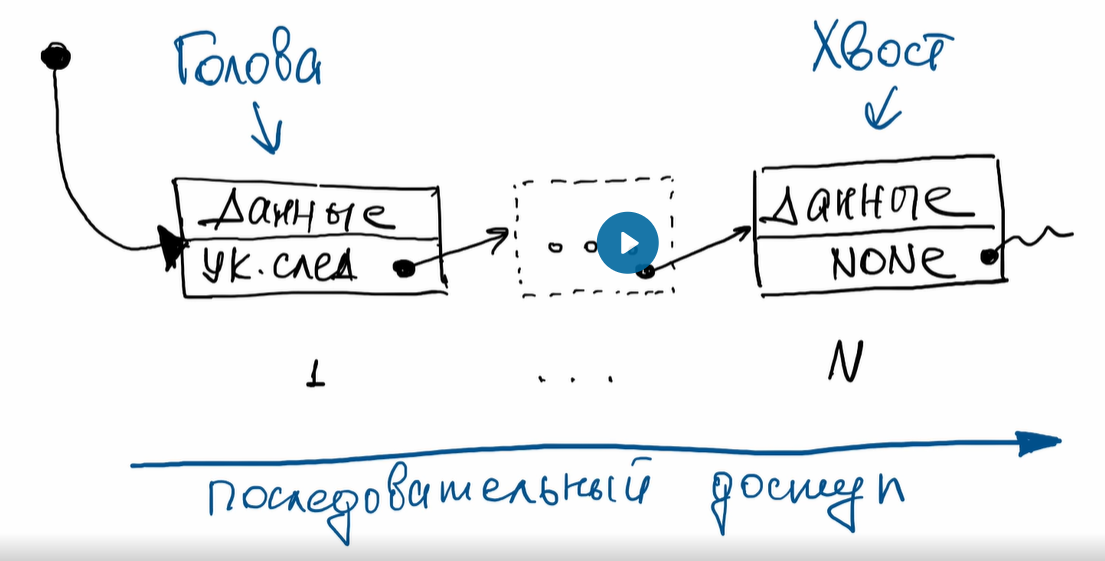
например:

значения: массив целых чисел

операции: объединение, пересечение, разность

### Связанный список

как работает связный список



### 

#### Поиск и длина

На основе этого механизма списка можно написать функцию поиска элемента по индексу и функцию длины списка:

для класса

class MyList(list):

def \_\_init\_\_(self):

self.head = None

def find(self, index) -> Data:  
 current = self.head

while index > 0:

index -= 1

current = self.next

return current

def size(self):

current = self.head

cnt = 0

while current is not None:

cnt += 1

current = self.next

return cnt

#### Вставка элемента

def add(self, element: str, index: int):

# создание элемента

value = Data()

value.value = element

if index == 0:

value.next = self.head # сказали, что следующий элемент это head

списка, таким образом пополнили список

новым значением, вставшим перед

нулевым элементом

self.head = value # объявили, что head теперь является наш

добавленный элемент

else:

value.next = self.find(index +1) # наш элемент указывает на следующий

index +1 элемент в списке

prev = sel.find(index-1) # запоминаем index -1 элемент в списке

prev.next = value # предыдущий элемент указывает на наш

добавленный элемент

Важно: сначала делаем так чтобы наш элемент указывал на index +1 элемент, и только потом, чтобы предыдущий index -1 элемент указывал на наш добавленный

#### Удаление

def delete(self, index):

if index == 0:

self.head = self.head.next

else:

prev = self.find(index -1)

prev.next = self.find(index +1) # prev.next = prev.next.next

### Динамический список

ДС - это список, который реализует создание дополнительных ячеек, когда место заканчивается. Таким образом, что создает массив большей длины и копирует все элементы в него.

#### Создание нового массива

def allocate\_array(size: int) -> list:

return [None] \* size

#### Копирование эл-тов в новый массив

def new\_capacity(self, size: int):

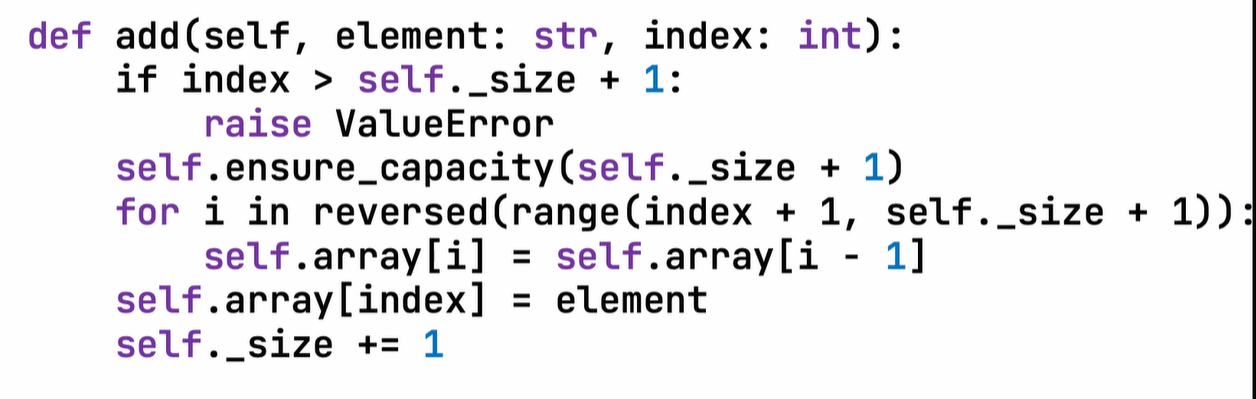
if self.\_size < size:

new\_arr = allocate\_array(len(self.array) \*2)

self.copy(self.array, new\_arr)

self.array = new\_arr

#### Добавление элемента в динамический массив с аллокацией

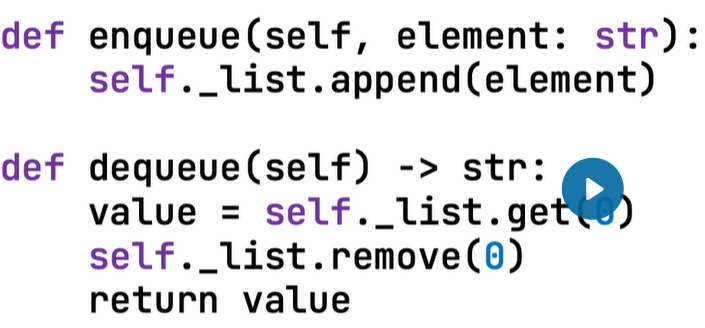


### Очередь (FIFO)

добавление в начало

удаление с конца

реализация этого принципа на базе списка

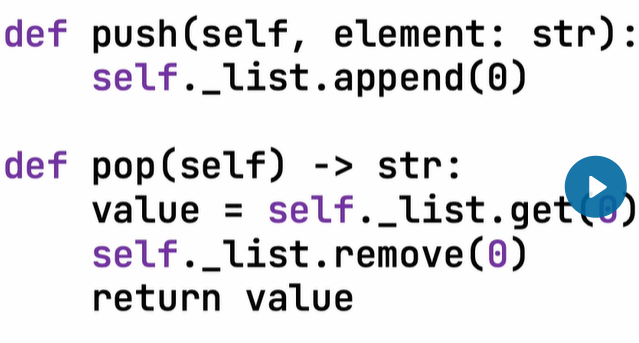


### Стек (LIFO) / ( FILO)

добавление в конец

удаление с конца

LIFO:



Используется в текстовых редакторах, когда мы отменяем последнее действие

## Ассоциативные массивы

### Словарь в пайтон

people = ['Лёня', 'Бенджамин']

numbers = [89876160164, 89875835803]

numbers\_book = dict(zip(people, numbers))

print(numbers\_book)

print(type(numbers\_book))

хранить ассоциативный массив можно:

1. массив (пример выше)
2. сбалансированные бинарные деревья
3. хэш - таблицы

### Хэш таблицы

ХТ - это структура данных подобная словарю

хэш функция - это функция которая возвращает какое то целое 8битное int число. Она может быть определена по-разному, например

hash(a) = a mod b

Хэшфункция должна быть быстро вычислимой и иметь минимальное количество коллизий

Плохая хэш функция часто возвращает одно и тоже значение при разных входных данных - это называется коллизия

для разрешения коллизий

1. Метод цепочек: создают массив из элементов с одинаковым хэшем
2. Метод открытой адресации: используют правило по которому распологаются одинаковые хэши

### Двоичные деревья

ДД - структура данных, в которой объекты структурируются как семейное древо, у каждого элемента есть 1 родитель, кроме корня.

Бывают деревья разной степени r = n (сколько может быть потомков у родителя). r = 2 - двоичное дерево.

У каждого нода (узла) древа есть свой ключ и ссылки на потомков.

#### Реализация двоичного дерева в пайтон

class Node:

def \_\_init\_\_(self, key=None):

self.key = key

self.left = None

self.right = None

root = Node("root")

child = Node("left")

root.left = child

#### Определения

Двоичное дерево поиска - дерево у узлов которого максимум 2 наследника и есть право ключей: ключ левого наследника < ключа родителя, а ключ правого наследника >= ключа родителя.

Путь дерева - порядок нодов от корня до нужного элементов

Длина дерева - самый длинный путь

Лист дерева - нижние потомки без собственных потомков

Глубина узла - расстояние от узла до корня

Высота узла (H(узел)) - H(null) = -1; H(один нод без наследников) = 0;

H(нод с наследниками) = max(H(левый наследник), H(правый наследник)) +1

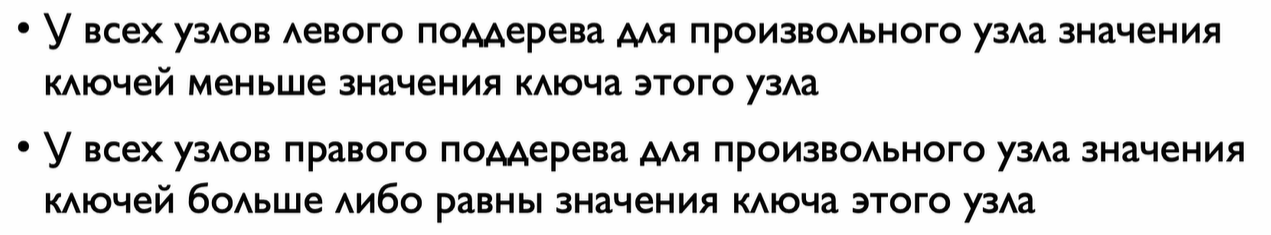
Лоза - все элементы дерева левые/ правые

Сбалансированное дерево - дерево в случае лозы модифицируется с помощью вставки и удаления узлов так, чтобы левое и правое поддерево были примерно равны

Форма записи дерева через скобки

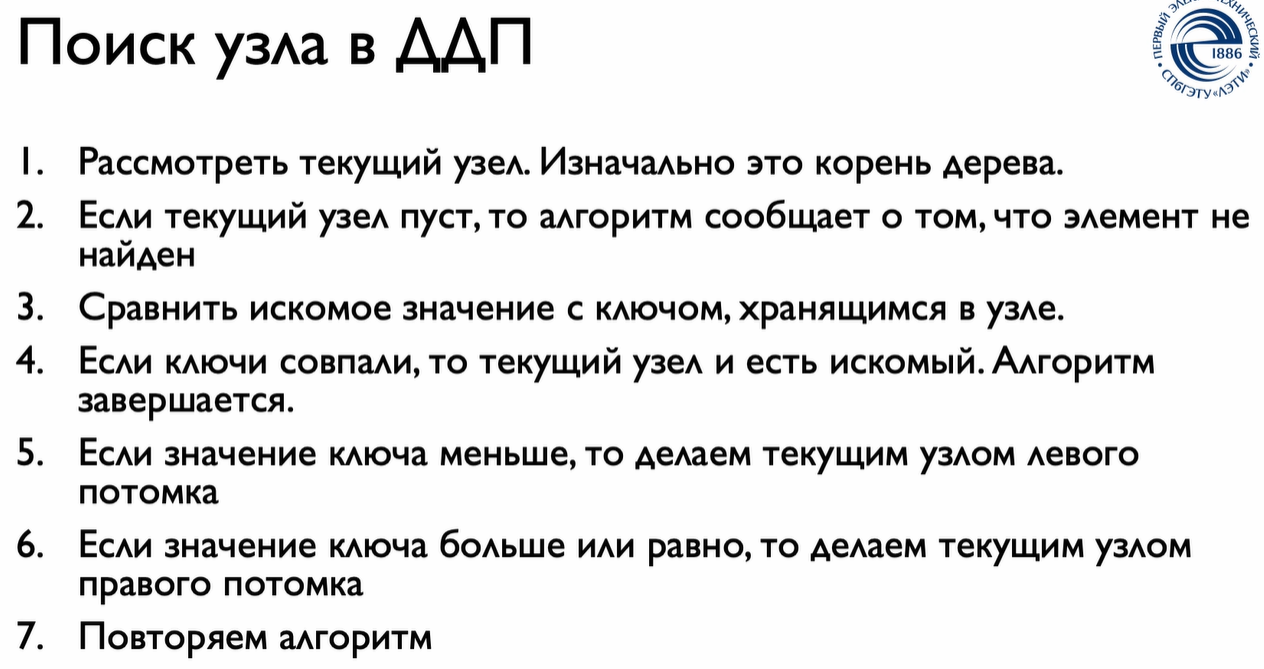


Организация значений дерева



#### Алгоритмы:

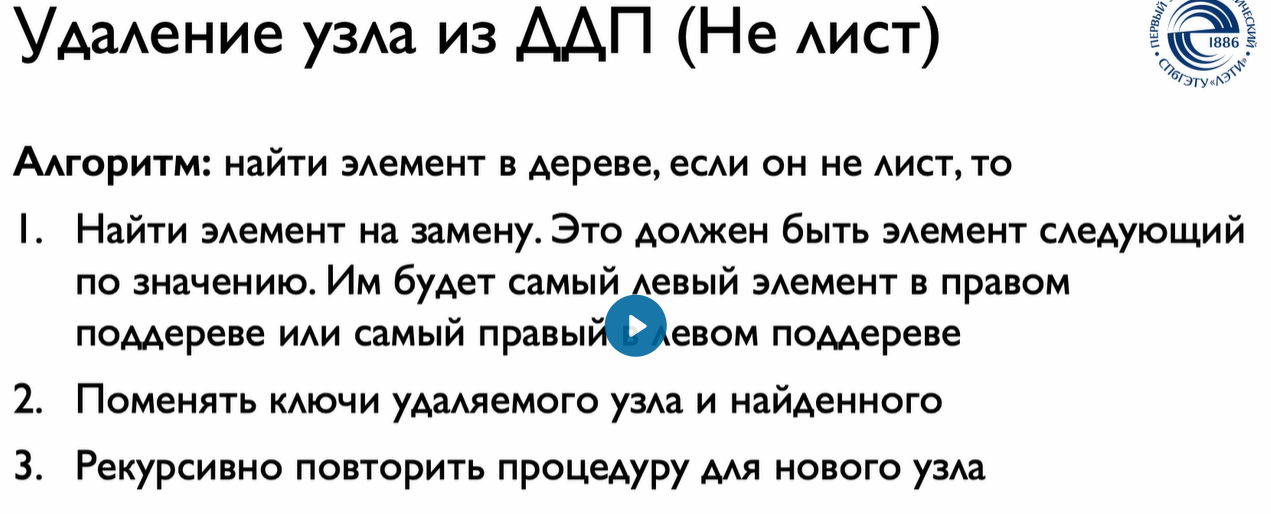
##### Поиск



##### Удаление узла (лист)

##### Удаление узла (не лист)

<https://cloud.etu.ru/index.php/s/f4seTWfnq88ak8j>



##### Обходы дерева

ОД - это когда с каждый элементом дерева выполняют какое то действие (например выводят на экран).

3 обхода дерева:

прямой обход (в глубину)

схема: корень - левое поддерево - правое поддерево



def traverse(node: Node):

if node is None:

return

print(node.key)

traverse (node. left)

traverse(node.right)

Центрированный обход (в глубину)

схема: левое поддерево- корень - правое поддерево

выводит значения по возрастанию



def traverse(node: Node):

if node is None:

return

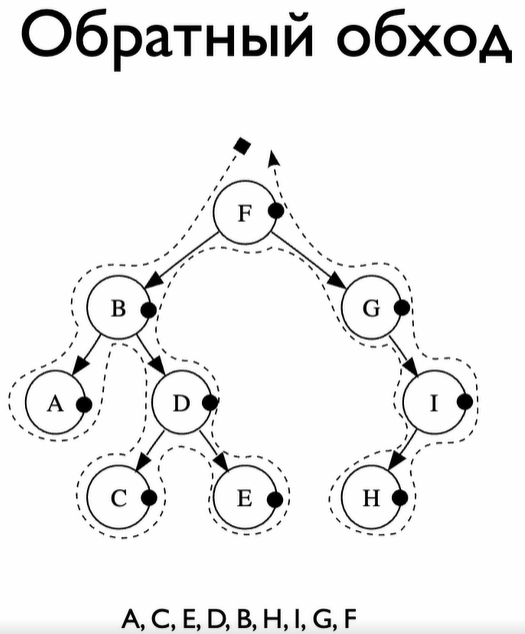
traverse (node. left)

print(node.key)

traverse(node.right)

обратный обход (в глубину)

схема: левое поддерево - правое поддерево - корень



def traverse(node: Node):

if node is None:

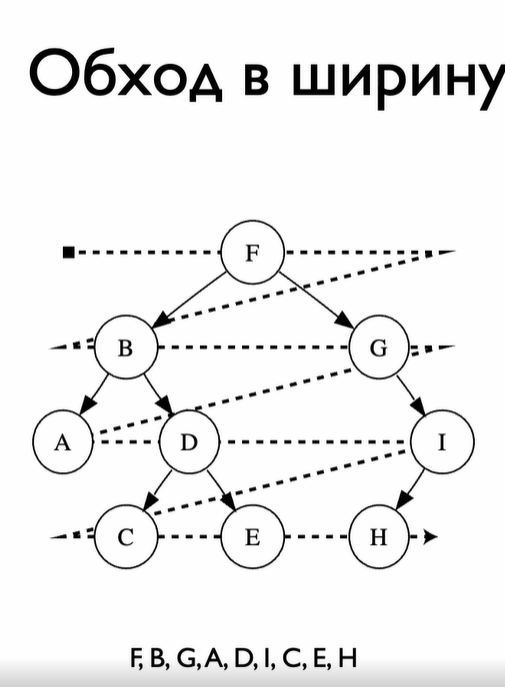
return

traverse (node. left)

traverse(node.right)

print(node.key)

Обход в (в ширину)



def list(node: Node):

queue = [node]

while len(queue) > 0:

elem = queue.pop(0)

print(elem.key)

if elem.left is not None:

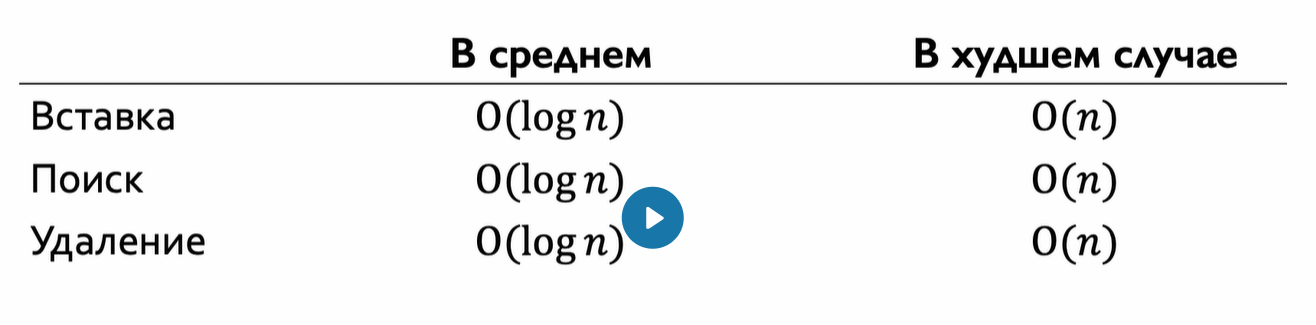
queue.append(elem.left)

if elem.right is not None:

queue.append(elem.right)

#### Временная сложность поиска элемента

T(n) = log2(n+1)



Худший случай - лоза (все элементы левые/ правые)

#### АВЛ дерево

Важно соблюдать баланс дерева, чтобы поиск занимал T(n) = log(n)

Чтобы исключить вырождения дерева в лозу прибегают к балансировке - перенос некоторых элементов в другую часть дерева.

Сбалансированное дерево - это дерево, которое было плохим и подверглось балансировке

АВЛ дерево - сбалансированное дерево, так что высота правого и левого поддерева КАЖДОГО НОДА отличается не более чем на 1.

Или же модуль баланса каждого нода дерева <= 1

##### Баланс

Балансируют деревья исходя из значения баланса каждого нода.

Для AВЛ дерева допустимый баланс B(узел) <= 1

как вычисляется баланс узла

высота H(node):

H(null) = -1

H(один нод без наследников) = 0

H(нод с наследниками) = max(H(левый наследник), H(правый наследник)) +1

Тогда баланс нода B(node):

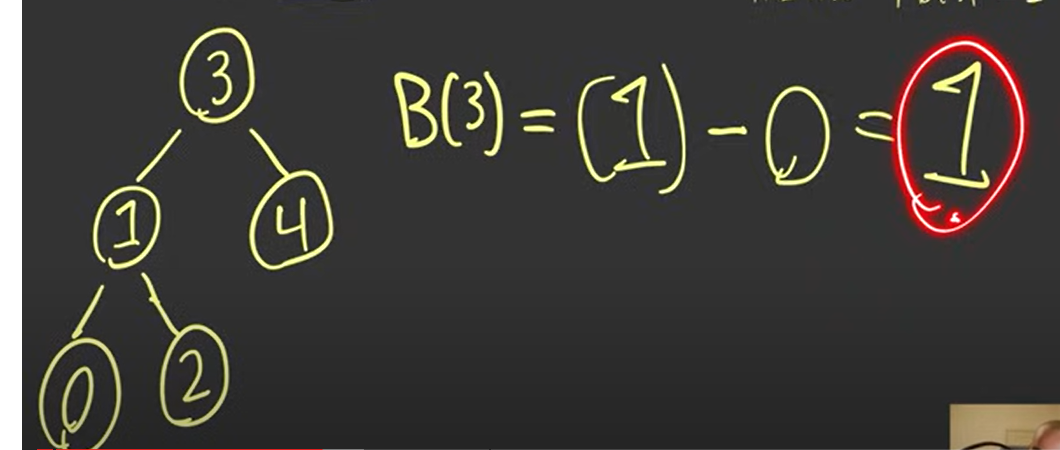
B(node) = H(левый наследник) - H(правый наследник)

И для АВЛ дерева: abs(B(node)) <= 1

Тогда если B(node) = H(l) - H(r) = 1 → левое поддерево перевешивает

= -1 → правое крыло перевешивает

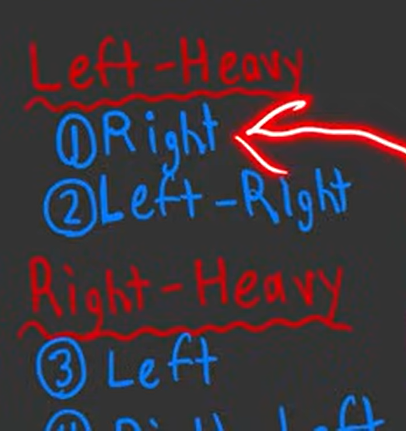
Пример



B(3) = H(1) - H(4) = 1 - 0 = 1 → является АВЛ деревом

##### Алгоритмы балансировки

правила применения балансировки



###### Правый поворот

def right\_rotate(y):

x = y.left

T2 = x.right

# Выполняем поворот

x.right = y

y.left = T2

###### Левый поворот

def left\_rotate(y):

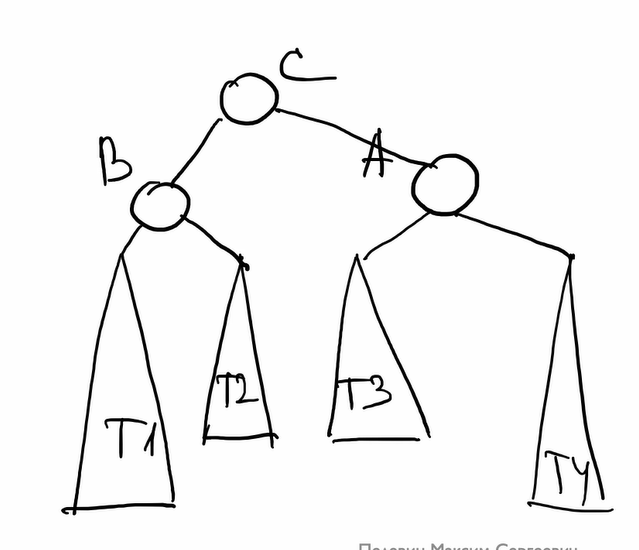
x = y.right

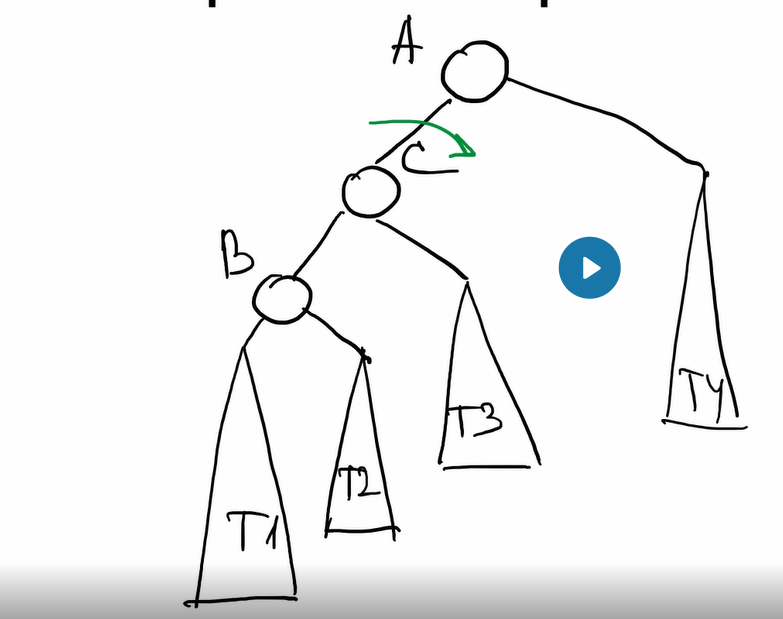
temp\_xleft= x.left

x.left = y

y.right = temp

###### Левый - правый поворот





[1] Правило осуществлять балансировку при добавлении элемента:



Как нужно выполнять баланс

1. посчитать балансы всех узлов
2. начиная более высоких элементов применять повороты по правилу [1]
3. после каждого поворота пересчитывать балансы и поворачивать менее высокие элементы

#### ДЗ

проверка на наличие наследников:

def in\_order\_traversal(t):

if 'left' in t:

in\_order\_traversal(t['left'])

print(t['key'])

if 'right' in t:

in\_order\_traversal(t['right'])

### B-дерево

## Графы

Граф это структура данных, такая, что

[ V,E,Ψ ] , где V - это множество вершин

E - множество рёбер

Ψ - функция, связываюшая некоторые вершины некоторыми ребрами

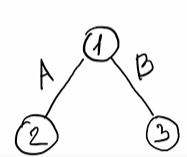
пример:

V = [1,2,3]

E = [A,B]

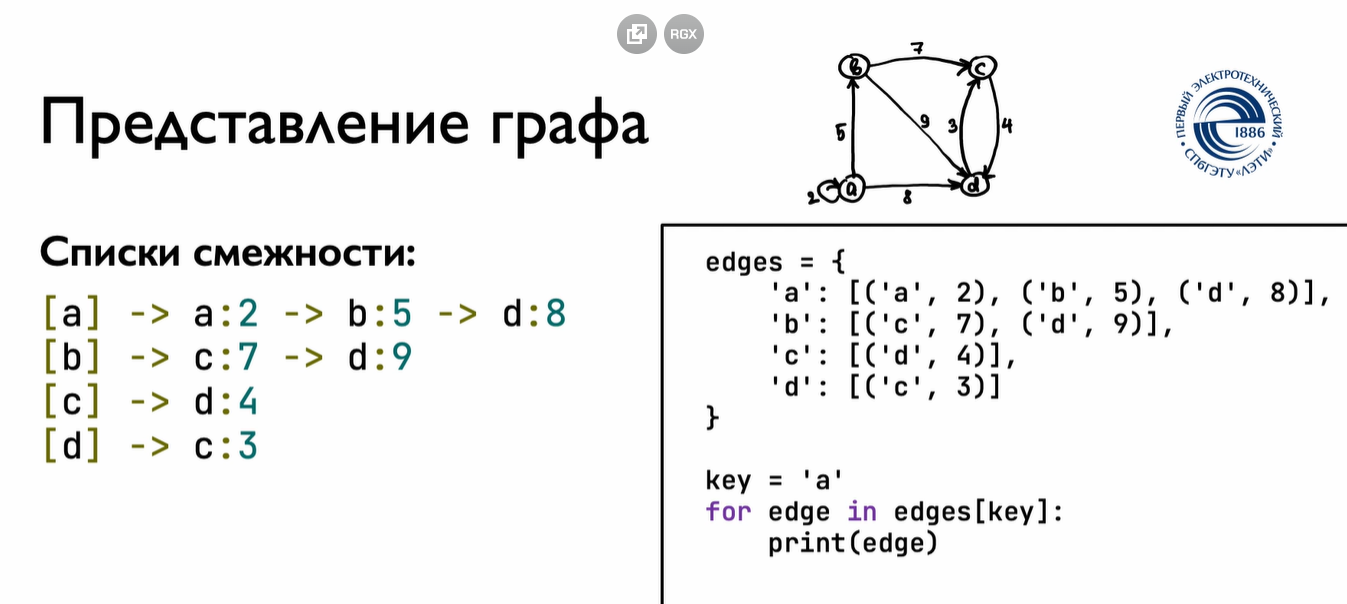
Ψ(A) = [ 1,2 ] - это значит, что вершины 1 и 2 связаны ребром A

Ψ(B) = [ 1,3 ] - вершины 1 и 3 связаны ребром B



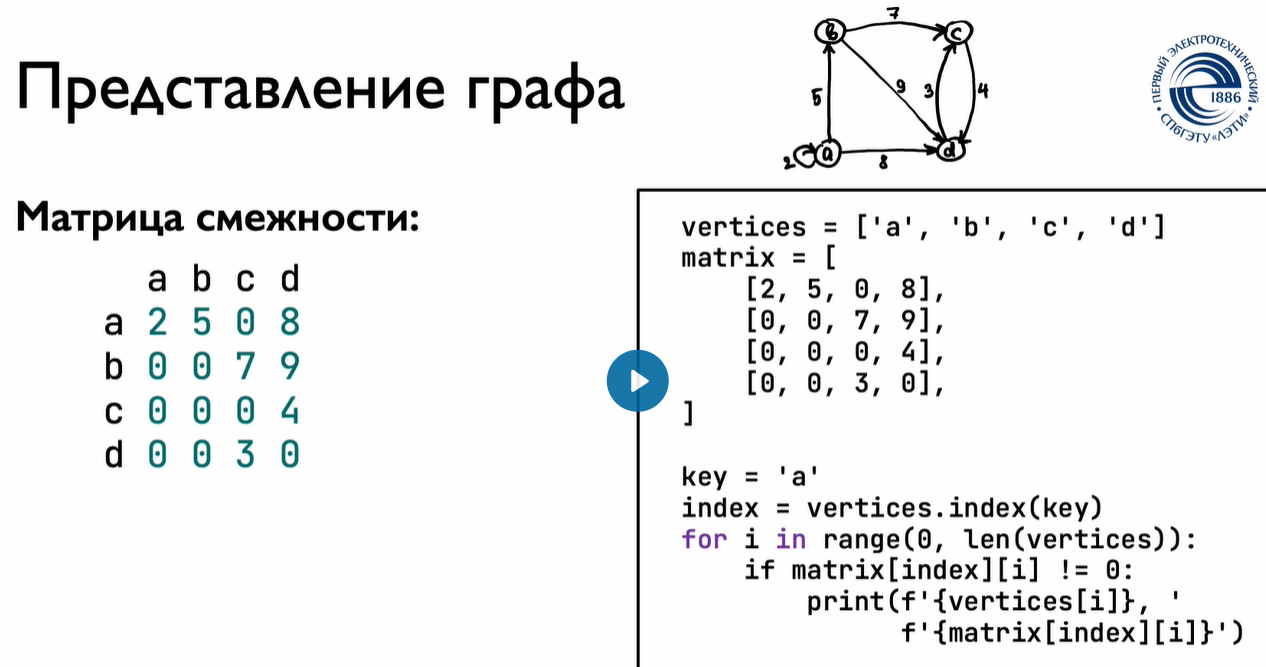
Так [ V,E,Ψ ] полностью изображает весь граф и является полным описанием графа

Реализация на пайтон



В каждом ключе хранится список из пар: следующая вершина к которой можно отправится и стоимость пути.

Представление в виде матриц



0 - нет ребра

1 / x - есть ребро / стоимость перехода по ребру

алгоритм выводит

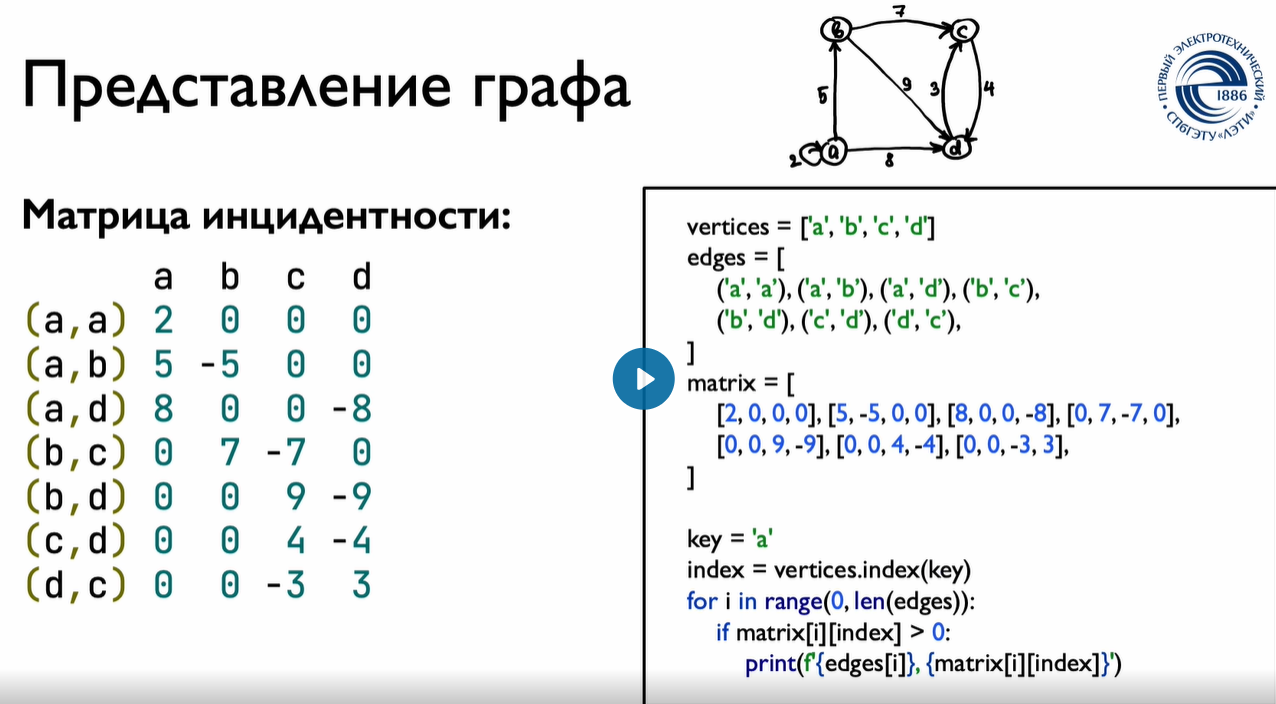
(‘a’, 2)

(‘b’, 5)

(‘d’, 8)

количество вершин к которым можно перейти из вершины key за стоимость

Представление в виде матрицы инцидентности



строки - список всех РЁБЕР

столбцы - все вершины

(а,а) - а = 2 - означает ребро а-а относится к вершине а и ребро направлено как а -> a, а стоимость 2

(а,b) - b = -5 - означает ребро а-b относится к вершине b и ребро направлено как а -> b, а стоимость 2

Для неориентированного графа знак не имеет значения

### Обход в ширину

позволяет пройти по кратчайшему пути (я хз как это)

def bfs(graph, s, process):

queue = [s] # очередь вершин

visited = {s: True} # массив посещений, чтобы граф не зацикливался

while len(queue) != 0:

next = queue.pop(0)

process(next) # обработка вершины

neighbours = graph[next]

for neighbour in neighbours:

if not visited.get(neighbour):

queue.append(neighbour)

visited[neighbour] = True

### Поиск кратчайшего пути

модификация обхода в ширину, чтобы сохранить кратчайший путь

def bfs(graph, s):

queue = [s]

visited = {s: True}

predecessors = {}

while len(queue) != 0:

next = queue.pop(0)

yield next, predecessors

neighbours = graph[next]

for neighbour in neighbours:

if not visited.get(neighbour):

queue.append(neighbour)

visited[neighbour] = True

predecessors[neighbour] = next

### Обход в глубину

позволяет выполнить топологическую сортировку:

То есть определить какие узлы являются начальными (к которым нет приходящих рёбер), а какие последними (у которых нет уходящих рёбер, а только приходящие)

и так по порядку

def dfs(graphs, s, process):

process(s)

neighbours = graphs[s]

visited[s] = True

for neighbour in neighbours:

if not visited.get(neighbour):

dfs(graphs, neighbour, process)

рабочий алгоритм выстраивания узлов в топологическом порядке

visited = {}

def dfs(graphs, process):

for key in graphs.keys():

dfsElement(graphs, key, process)

def dfsElement(graphs, s, process):

if visited.get(s) is not None:

return

neighbours = graphs.get(s)

visited[s] = True

if neighbours is not None:

for neighbour in neighbours:

if not visited.get(neighbour):

dfsElement(graphs, neighbour, process)

process(s)

sorted = []

dfs({

'shirt': ['tie', 'belt'],

'socks': ['shoes'],

'undershorts': ['pants', 'shoes'],

'pants': ['belt', 'shoes'],

'belt': ['jacket'],

'tie': ['jacket']

}, lambda x: sorted.insert(0, x))

print(sorted)

### Алгоритм Крускала

Алгоритм Крускала — это алгоритм поиска минимального остовного дерева, построенного на графе. Остовным деревом называют ацикличный связный граф минимального веса, другими словами, это алгоритм, который позволяет соединить все вершины графа так, чтобы в новом полученном графе не существовало циклов, а сумма всех взвешенных рёбер была бы минимальной.

Например, для случая, когда в графе существуют 3 ребра: AB, BC и AC с весами: w(AB)=1, w(BC)=2 и w(AC)=3, то этот граф может иметь 3 остовных (но пока не минимальных) деревьев:

1. AB и BC
2. BC и AC
3. AB и AC

Сумма весов рёбер для каждого такого дерева будет равна соотвественно 3, 5 и 4. Таким образом минимальным остовным деревом может быть только один вариант, а именно 1: AB и BC.

Алгоритм Крускала позволяет находить минимальное остовное дерево. Для этого в самом начале работы алгоритма полагают, что результат содержит пустое множество рёбер. После чего в исходном графе находят ребро наименьшего веса и добавляют его к результату, после чего это ребро исключают из рассмотрения. При добавлении ребра обязательно проверяется, возникнет ли после добавления ребра в результат цикл или нет. Если цикл возникает, то такое ребро не добавляется в результат.

Рассмотрим на следующем примере. Пусть граф задан через использование списков рёбер, где каждое ребро описывается кортежем (вершина 1, вершина 2, вес ребра):

[('A', 'B', 1), ('B', 'C', 3), ('A', 'C', 2), ('D', 'C', 3)].

Построим для него минимальное остовное дерево. Для этого отсортируем рёбра в порядке возрастания их весов:

[('A', 'B', 1), ('A', 'C', 2), ('B', 'C', 3), ('D', 'C', 3)].

И создадим пустой список для хранения результатов:

[].

Шаг 1: Добавляем первое ребро из отсортированного списка в результат. Так как он добавляется первым и вершины разные, цикл не может образоваться, таким образом результат = [('A', 'B', 1)]

Шаг 2: Проверяем следующее ребро ('A', 'C', 2). При добавлении в результат циклов не образуется, т.к. из 'A' нельзя попасть в 'C' другим путём. Таким образом результат = [('A', 'B', 1), ('A', 'C', 2)]

Шаг 3: Проверяем следующее ребро ('B', 'C', 3). При его добавлении образуется цикл, т.к. из B можно попасть в C через вершину A, то есть по рёбра: ('A', 'B'), ('A', 'C'), поэтому это ребро не попадает в результат.

Шаг 4: Проверяем следующее ребро ('D', 'C', 3). При его добавлении циклов не образуется, т.к. нельзя попасть из вершины 'D' в вершину 'C' по тем рёбрам, что уже добавлены в результат. Поскольку это последняя пара рёбер, то окончательный ответ состоит из рёбер: [('A', 'B', 1), ('A', 'C', 2), ('D', 'C', 3)].

Для проверки того, что добавляемое ребро может создать цикл можно использовать любой из обходов графа (результата) и проверить, что при посещении всех верши в этом графе не будет встречена соседняя вершина. Этот вариант корректный, но достаточно долгий. Вместо него можно использовать структуру данных под названием *система непересекающихся множеств*.

Эта структура позволяет хранить информацию о том, к какому множеству относится вершина графа. Изначально полагается, что каждая вершина есть отдельное множество. Для этого можно использовать ассоциативный массив, который назовём ufs (сокращение от union-find set). Если для произвольной вершины "a" список возвращает None, то этот элемент является идентификтором множества. К примеру, пусть существуют 3 вершины A, B и C. Вначале запрос для любой из них будет возвращать None, например:

ufs[A] == None

Это значит, что A является идентификатором множества A, B — идентификатором множества B и C — идентификатором множества C. Чтобы объединить 2 вершины в одно множество, выберем любую из них и укажем ссылку на неё в ufs:

ufs[A] = B.

Теперь B является идентификтором множества их вершин (A и B), т.к. запрос ufs[B] вернёт None. Вершина A ссылается же на B. Вершина C составляет отдельное множество. Теперь добавим в множество c идентификтором B вершину С. Допустим, это сделано через связку с вершиной A:

ufs[C] = A.

Теперь C ссылается на A, а A ссылаться на B. Все три вершины входят в одно множество с идентификатором B. Таким образом, чтобы определить в какое множество входит вершина достаточно переходить по ufs до тех пор, пока не встретится None.

Таким образом, чтобы определить, входят ли 2 различные вершины в одно множество или нет, достаточно найти идентификаторы множеств к которым эти множества относятся. Если это один и тот же идентификатор, то это означает, вершины входят в одно множество, а ребро, которое содержит эти вершины создаст цикл при добавлении.

Таким образом, весь алгоритм Крускала описывается так:

1. Отсортировать все рёбра в порядке возрастания веса
2. Создать пустой список для хранения результата
3. Создать ассоциативный массив ufs для хранения системы непересекающихся множеств
4. Для каждого ребра из отсортированного списка:
5. Проверить, что идентификторы множества текущих вершин ребра различны. Если это так, то добавляем ребро в результат и объединяем вершины ребра в одно множество
6. иначе пропускаем ребро

Когда рёбра закончатся, результат будет содержать только те рёбра, которые составляют минимальное остовное дерево.

def merge(a, b):

c = []

i = j = 0

while i < len(a) and j < len(b):

if a[i][2] < b[j][2]:

c.append(a[i])

i += 1

elif a[i][2] == b[j][2]:

c.append(a[i])

c.append(b[j])

i += 1

j += 1

else:

c.append(b[j])

j += 1

while i < len(a):

c.append(a[i])

i += 1

while j < len(b):

c.append(b[j])

j += 1

return c

def sort(s):

if s == []:

return []

if len(s) == 1:

return s

mdl = len(s) // 2

lf = sort(s[:mdl])

rg = sort(s[mdl:])

return merge(lf, rg)

#a = [4,67,2,8,6,5,4,7,9]

#print(splt(a))

def find(ufs, key):

if key not in ufs:

return key # Возвращаем None, если нет идентификатора

if ufs[key] is None:

return key

key = ufs[key][0]

return find(ufs, key)

def mst(a):

c = []

ufs = dict()

a = sort(a)

for i in range(len(a)):

root1 = find(ufs, a[i][0])

root2 = find(ufs, a[i][1])

if root1 != root2:

c.append(a[i])

#if a[i][0] not in ufs:

# ufs[a[i][0]] = [] неправильная логика

# ufs[a[i][0]].append(a[i][1])

#else:

# ufs[a[i][0]].append(a[i][1])

if root1 not in ufs:

ufs[root1] = []

ufs[root1].append(root2)

return c

a = [

('A', 'B', 1),

('B', 'C', 3),

('A', 'C', 2),

('D', 'C', 3)

]

print(mst(a))

## Ufs (union-find set)

Cтруктуру данных под названием *система непересекающихся множеств (ufs)*.

Эта структура позволяет хранить информацию о том, к какому множеству относится вершина графа. Изначально полагается, что каждая вершина есть отдельное множество. Для этого можно использовать ассоциативный массив, который назовём ufs (сокращение от union-find set)

Реализация в пайтон в виде словаря

ufs = {

1: 2,

2: 4,

3: 4,

4: None

}

# Алгоритмы сортировки

## Простые сортировки

Обычно O(n²)

### Сортировка выборкой

def selectionsort(a):

for i in range(len(a)):

for j in range(i, len(a)):

if a[j] < a[i]:

a[j], a[i] = a[i], a[j]

return a

a = [4,67,2,8,6,5,4,7,9]

print(selectionsort(a))

временная сложность алгоритма:

T(n) = O(n2)

### Сортировка пузырьком

Смысл: начиная с конца массива проверяем два элемента, если левый больше правого, то меняем элементы местами - таким образом в начало массива вернётся самый малый элемент.

def swap(arr, i, j):

arr[i], arr[j] = arr[j], arr[i]

def bubble\_sort(elements):

n = len(elements)

for i in range(1, n):

is\_swapped = False

for j in reversed(range(i, n)):

if elements[j - 1] > elements[j]:

swap(elements, j - 1, j)

is\_swapped = True

if not is\_swapped:

break

### Сортировка вставками

Смысл:

1. начиная с начала массива (слева).
2. сравниваем два элемента правый и левый.
3. если слева больше, то передвигаем элемент влево, пока не найдётся элемент слева меньший, чем текущий.

def swap(arr, i, j):

arr[i], arr[j] = arr[j], arr[i]

def insertion\_sort(elements):

n = len(elements)

for i in range(1, n):

j = i

while elements[j - 1] > elements[j] and j > 0:

swap(elements, j - 1, j)

j -= 1

### ДЗ Бинарной вставка

def binary\_search\_smaller(array, target):

"""""

Returns:

Индекс элемента target в массиве, если он найден,

иначе -1.

"""

left = 0

right = len(array) - 1

while left <= right:

mid = (left + right) // 2 # Находим средний индекс

if array[mid] > target:

right = mid - 1 # Ищем в правой половине

elif array[mid] < target:

if array[mid+1] >=target:

return mid

else:

left = left+1

return -1 # Элемент не найден

def binary\_insertion\_sort(a):

for i in range(1, len(a)):

if a[i-1] > a[i]:

idx0 = 0

idx = binary\_search\_smaller(a[0:i], a[i])

if idx == -1:

idx = 0

idx0 = 1

temp = a[i]

for j in reversed(range(idx, i)):

a[j+1] = a[j]

if idx0 == 1:

a[0] = temp

else:

a[idx+1] = temp

return a

a = [4,67,2,8,6,5,4,7,9]

print(binary\_insertion\_sort(a))

посчитаем временную сложность T(n) сортировки вставками и бинарной вставкой

### ДЗ Бинарная вставка best

def binary\_search(array, target, start, end):

left = start

right = end

while left <= right:

mid = (left + right) // 2 # Находим средний индекс

if array[mid] == target:

return mid # Элемент найден

elif array[mid] < target:

left = mid + 1 # Ищем в правой половине

else:

right = mid - 1 # Ищем в левой половине

return (-left) # Элемент не найден - возвращение индекса , где бы этот элемент должен был бы находится

def swap(x, old, new):

temp = x[old]

for i in reversed(range(new, old)):

x[i+1] = x[i]

x[new] = temp

def sort(x):

for i in range(len(x)-1):

if x[i+1] < x[i]:

idx = binary\_search(x,x[i+1],0, i)

if idx < 0:

idx = -idx

swap(x,(i+1), idx)

return x

## Эффективные сортировки

### Сортировка слиянием

сложная хрень

<https://www.youtube.com/watch?v=LCfwxi2RPK4>

Смысл:

1. список рекурсивно делится на мелкие списки до тех пор пока не достигнет списков размером 1
2. парам разделенных мелких списков применяют метод слияния.
3. Слияние происходит так - проверяются i и j элемент, если i меньше то он встает первым в список и i = i +1, потом сравниваются новый i и старый j
4. так до тех пор пока не вернёмся к исходному списку

def merge(a, b):

c = []

i = j = 0

while i < len(a) and j < len(b):

if a[i] < b[j]:

c.append(a[i])

i += 1

else:

c.append(b[j])

j += 1

while i < len(a):

c.append(a[i])

i += 1

while j < len(b):

c.append(b[j])

j += 1

return c

def splt(s):

if len(s) == 1:

return s

mdl = len(s) // 2

lf = splt(s[:mdl])

rg = splt(s[mdl:])

return merge(lf, rg)

a = [4,67,2,8,6,5,4,7,9]

print(splt(a))

### Быстрая сортировка

def partition(elements, p: int, r: int) -> int:

i = p - 1

for j in range(p, r):

if elements[j] <= elements[r]:

i += 1

swap(elements, i, j)

i += 1

swap(elements, i, r)

return i

def quicksort(elements, start, end):

if start < end:

q = partition(elements, start, end)

quicksort(elements, start, q - 1)

quicksort(elements, q + 1, end)

# Алгоритмы поиска

## Бинарный поиск

Смысл:

0) беретётся ОТСОРТИРОВАННЫЙ массив

1) устанавливаются рамки: левая 0 индекса и правая len(a)-1 индекса

2) проверяется средний элемент

3) если средний равен искомому, то поиск завершён

4) если средний больше искомого, то правая рамка сдвигается на индекс mid -1 и снова проверяется средний элемент

6) если средний меньше искомого, то левая рамка сдвигается вправо на индекс mid +1 и снова проверяется средний элемент

7) итд пока не a[mid] == target

для сортировки бинарной вставкой (неэффективная сортировка):

0) беретёся ОТСОРТИРОВАННЫЙ массив

1) устанавливаются рамки: левая 0 индекса и правая len(a)-1 индекса

2) проверяется средний элемент

3) если средний равен искомому, то поиск завершён

4) если средний больше искомого, то правая рамка сдвигается влево на 1 и снова проверяется средний элемент

6) если средний меньше искомого, то левая рамка сдвигается вправо на 1 и снова проверяется средний элемент

7) итд пока a[mid) < (>) target

def binary\_search(array, target):

"""""

Returns:

Индекс элемента target в массиве, если он найден,

иначе -1.

"""

left = 0

right = len(array) - 1

while left <= right:

mid = (left + right) // 2 # Находим средний индекс

if array[mid] == target:

return mid # Элемент найден

elif array[mid] < target:

left = mid + 1 # Ищем в правой половине

else:

right = mid - 1 # Ищем в левой половине

return -1 # Элемент не найден

# Алгоритмы поиска подстроки

## Наивный поиск

смысл: если совпала первая буква, то проверяют вторую, если не совпала хотя бы одна буква, то поиск сдвигается на следующий символ

def search(s, p):

for i in range(0, len(s) - len(p)):

for j in range(0, len(p)):

if s[i + j] != p[j]:

break

else:

return i

return -1

## Алгоритм Рабина-Карпа:

смысл: буквы переводятся через хэш функцию в числа, на i позиции провер

def search(s, p):

p\_hash = sum(map(lambda c: ord(c), p))

s\_hash = sum(map(lambda c: ord(c), s[:len(p)]))

for i in range(0, len(s) - len(p)):

if p\_hash == s\_hash and p == s[i:len(p)+i]:

return i

s\_hash = s\_hash - ord(s[i]) + ord(s[i + len(p)])

return -1

## Алгоритм Бойера-Мура

Смысл:

1. проверяется последний символ

def search(s, p):

table = {c: i for i, c in enumerate(p[:-1])}

i = 0

while i <= len(s) - len(p):

for j in range(len(p) - 1, 0, -1):

if s[i + j] != p[j]:

break

else:

i += j - table.get(s[i + j], -1)

else:

return i

return -1

## Алгоритм Кнута-Мориса-Пратта

def search(s, p):

s = p + '#' + s

prefix = [0] \* len(s)

for i in range(1, len(s)):

j = prefix[i - 1]

while j > 0 and s[i] != s[j]:

j = prefix[j - 1]

if s[i] == s[j]:

prefix[i] = j + 1

if prefix[i] == len(p):

return i - 2 \* len(p)

return -1