

нумерация

90-кв - искображование массива данных в некоторому числу

хем	номер	фамилия
	0000000	
51402	745100000	Иванов

в таблицу можно писать последние 5 цифр (т.к. первые 2 цифры номера уже есть)

сравнением

по номеру телефона получили new → берем пятиразрядную → сравниваем номера

скорость  $O(1)$

данные помещаются в ячейку, номер которой определяется.

0	
1	
2	$v_1$
3	$v_3$
4	$v_2$
5	$v_4$
6	

В 3 ячейках записаны  
значения  $v_1, v_2, v_3$

$v_1 \rightarrow \text{hash}(v_1) = 2$

$v_2 \rightarrow \text{hash}(v_2) = 4$

$v_3 \rightarrow \text{hash}(v_3) = 2$

$v_4 \rightarrow \text{hash}(v_4) = 3$

$v_5 \rightarrow \text{hash}(v_5) = 5$  - но в таблице его нет

Приобщение к хеш-ки:

- 1) Давнообразность : если все цифры поменять местами исходные данные ( $=10007$ ), то хеши будут изначально разные
- 2) Равнодуность : распределение одинаково
- 3) Принциповедность
- 4) Сложность вычисления

Как суммировать все данные в python :

можно через функцию  $\Phi$ -то  $\text{hash}()$

лишь (list) между пересчетами

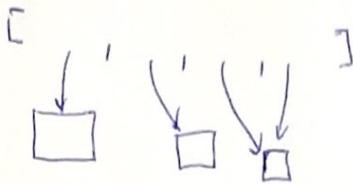
в python операции с видом хеш таблицы:

- 1)  $\text{set}()$
- 2)  $\text{dict}$  - словари

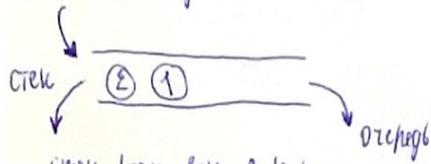
в таблицу добавляются only хешируемые объекты

Структуры данных - некоторые классы реализуют проекции данных  
которые можно присвоить и обрабатывать под. для данных (однотипные, связанные)

### 1) Список list (массив - способ хранения данных)



### 2) Стек, очередь



стек называют в list

В качестве очереди использование list не нужно (`pop(0)`-изъятие), используя библиотеку `queue`  
`from queue import Queue`

`x = Queue()`

`x.put(f)`

`y = x.get()`

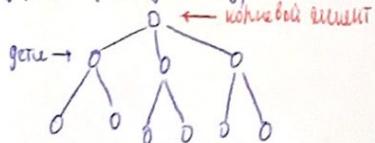
### 3) Ассоциативный массив (dict)

ключ = key	слово = значение
Moscow	Russia

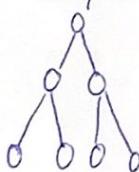
### 4) Связанные структуры

#### 5) деревья, графы, куки

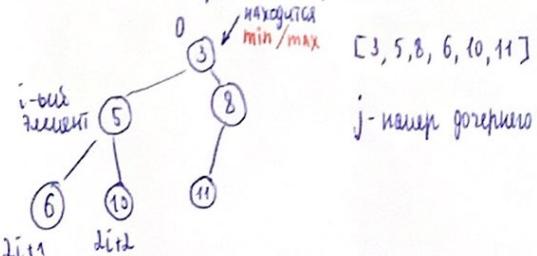
- деревья (дерево + ветвь)



Бинарные деревья (только 2 ветви)

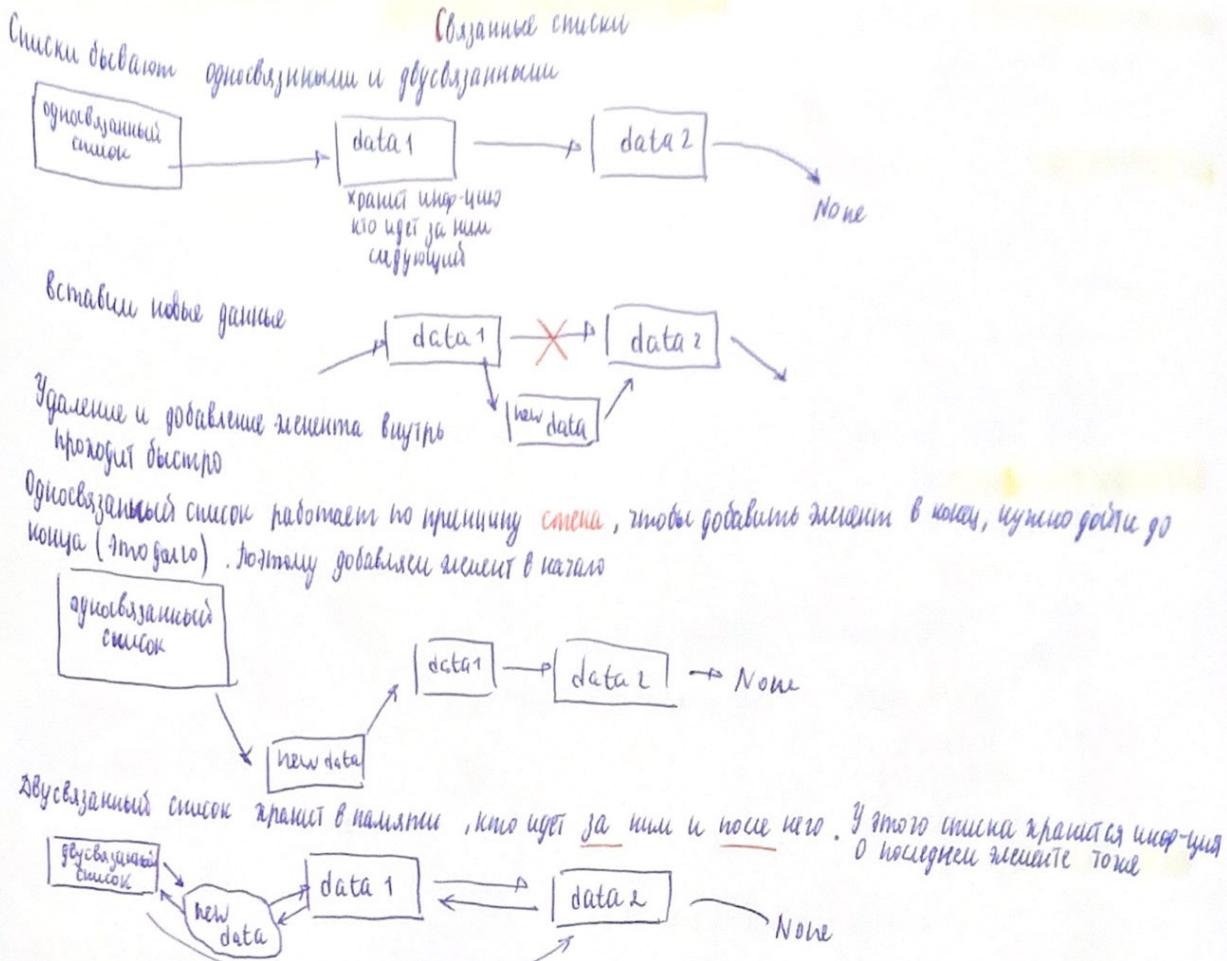


- куки - генерическое дерево, которое удаляется сверху-вниз и снизу-вверх, где удаляемый элемент имеет один



Если в куки  $N$  элементов, то время куки  $\sim \log_2(N)$

$j$ -номер первого элемента  $\Rightarrow (j-1)/2$  - наимен. индекса



Алгоритмы для односвязанных списков:

```

def LinkedList():
    # создание односвязанного списка
    return {'first': None}

def Node(value):
    # оп-ка, содержит value, value-значение
    return {'value': value, 'next': None}

def add_to_list(value, l-list):
    new_node = Node(value)
    new_node['next'] = l-list['first']
    l-list['first'] = new_node

def pop_from_list(l-list):
    res = l-list['first']['value'] # сохранение первого
    l-list['first'] = l-list['first']['next']
    return res

def print_list(l-list):
    current_node = l-list['first'] # первый узел
    while current_node is not None; # пока не дойдем до конца, т.е. пока текущий узел не None
        print(current_node['value'], end=' ')
        current_node = current_node['next'] # переходим к следующему узлу

    print() # отбрасываем кончик

def insert_value(node, value):
    new_node = Node(value) # новый узел
    new_node['next'] = node['next']
    node['next'] = new_node

```

автоматическое форматирование + heapify

↑ ? не беря подсказки

```

def add_to_heap(h, element):
    h.append(element)
    shift_up(h, len(h)-1)  # меняем местами на месте элемента

def shift_up(h, i):
    if i == 0:  # если голова, то ничего не делать
        return
    parent_i = (i-1)//2
    if h[i] < h[parent_i]:
        shift_up(h, parent_i)

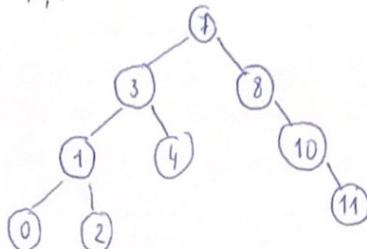
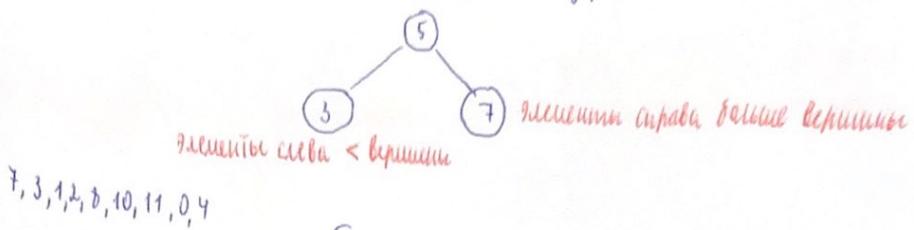
def pop_from_heap(h):
    if len(h) == 0:
        ошибка
    result = h[0]
    if len(h) == 1:
        return result
    h[0] = h.pop()
    shift_down(h, 0)
    return result

def shift_down(h, i):
    m = min([h[i] + h[2*i+1 : 2*i+3]])
    if h[i] == m:
        return
    if h[2*i+1] == m:
        меняем местами h[i] с h[2*i+1]
        shift_down(h, 2*i+1)
    else:
        меняем местами h[i] с h[2*i+2]

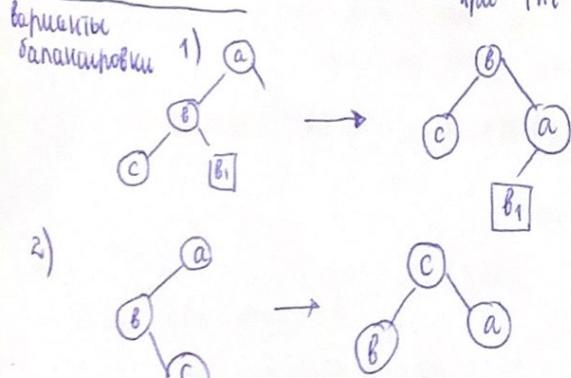
def heapify(h):
    for i in range(len(h)-1, -1, -1):
        shift_up(h, i)

```

## Двоичные деревья

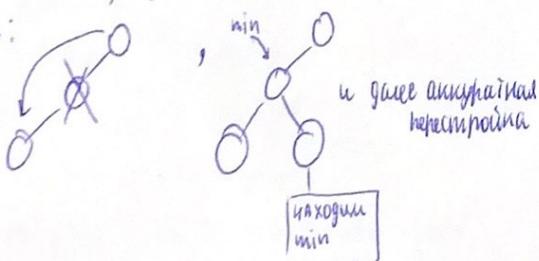


$h_L$        $h_R$ , если  $|h_L - h_R| \leq 1$ , то дерево сбалансировано  
при  $|h_L - h_R| = 2$  - нужно провести балансировку



Аналогично для случая, когда из левого сына удаляется

или удалили:

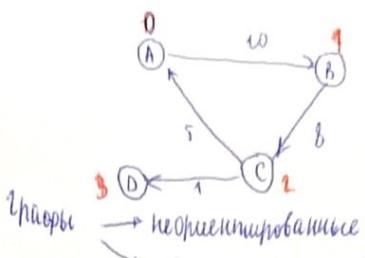


Алгоритм бинарного дерева

```

def BinTree():
    return {'root': None}  # инициализация корневого элемента
def Node(value):
    # создание узла, который содержит значение value и имеет родителя, которого
    # нет
    return {'left': None,
            'right': None,
            'value': value}
def insert_into_tree(t, value): # вставка в дерево
    if t['root'] is None: # проверка на создание корневого элемента
        t['root'] = Node(value)
    else:
        insert_into_node(t['root'], value) # вставка в узел
def insert_into_node(node, value):
    if value == node['value']: # проверка на равенство значений
        return
    if value > node['value']: # если значение больше, то добавлять элемент вправо
        if node['right'] is None: # проверка создания правого элемента
            node['right'] = Node(value) # создание правого элемента
        else:
            insert_into_node(node['right'], value) # вставка в правый элемент
    else:
        insert_into_node(node['left'], value) # вставка в левый элемент
def print_tree(t):
    print_node(t['root'])
    print()
def print_node(node): # вывод узла
    if node is None:
        return
    print('(', end=' ')
    print_node(node['left'])
    print(node['value'], end=' - ')
    print_node(node['right'])
    print(')', end=' ')

```



Графы  
неориентированные  
ориентированные (ходячие в одну сторону)  
Если граф неориентированный, то граф симметричный

$$\text{вес-число } (x, y) = G[x][y]$$

Еще один способ представления графов:

	A	D
A	0	
B	1	2
C	2	4
D	0	7
	8	

	0	1	2	3	4	5	6	7
0	1	40						
1	2	5						
2	0	60						
3	2	8						
4	0	5						
5	1	3						
6	3	1						
7	2	1						

### Теория графов

Граф  $G$  - упорядоченная пара множеств  $(V, E)$ ,

где  $V$  - множество вершин,

а  $E$  - множество неупорядоченных пар  $v \in V$

$$v \in V \quad e \in E$$

vertices    edges

$|V|$  - количество вершин,  $|E|$  - количество (11-мощность)



крайевые  
ребра

граф без петель и кратных  
ребер называется 简单图

( степень = 1 )

Степень вершины ( $\deg v$ ) - количество смежных ребер

Вершины смежны, если у них есть общее ребро

Смежности - отношение между вершинами и ребрами

Изолированная вершина - вершина, имеющая степень 0, называемая 悬挂结点 (лип)

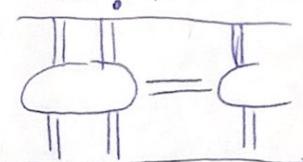
Вершина, имеющая степень, равную 1, называется 单边结点 (лип)

Решение задачи про мосты: если однократно по графу, даете у каждого

$$G = \{ 'A': \{ 'B': 10, 'C': 5 \}, 'B': \{ 'A': 10, 'C': 8 \}, 'C': \{ 'A': 5, 'B': 8 \}, 'D': \{ 'C': 1 \} \}$$

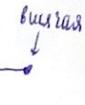
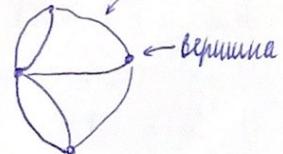
график  
для классификации  
графов

### Компоненты связности



Удаление вершины

ребро

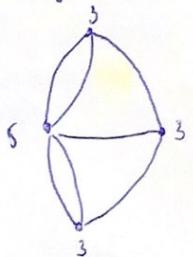


бисектрисы

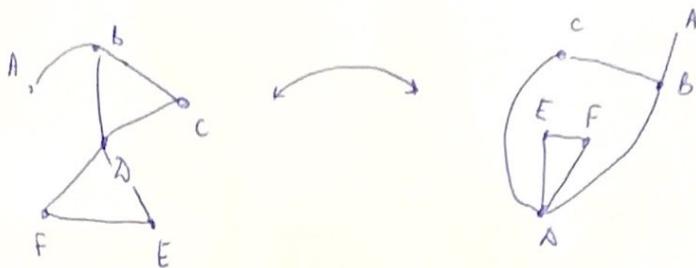
бисектрисы: сумма всех степеней графа четна

Полиграф - граф, в котором  $V$ -ноды-ва  $V$ ,  $e$ -ноды-ва  $E$

Эйлеровы графы



Частионные графы.  
 А и В - изоморфны, если  $\exists$  <sup>бijeктивное</sup> отображение  $V_A \leftrightarrow V_B$  и  $E_A \leftrightarrow E_B$  (отношения между вершинами не должны остат)



### Пути и циклы

Маршрут - путь вершин, где каждая соединена со следующей ребром (ни-бо ребер или греч. start u finish)

Путь - ориентированный маршрут

Цикл маршура - путь ребер

Член - маршрут без повтора ребер

Простой член - член без повтора вершин

Простой путь - путь без повтора ребер

Цикл - путь, где start = finish

Простой цикл - цикл, где нет повтор. ребер

Вершины связны, если есть член, их соединяющий

Компонента связности - подграф исходного графа, содержащий все вершины одного из классов эквивалентности по связности и все инцидентные им ребра

(максимальная связная подгруппа)

Связный граф - граф с одной компонентой связности

Мост - такое ребро, после удаления которого 1 из-бо компонент связности

Точка哥енсона - аналогично мосту только с вершиной

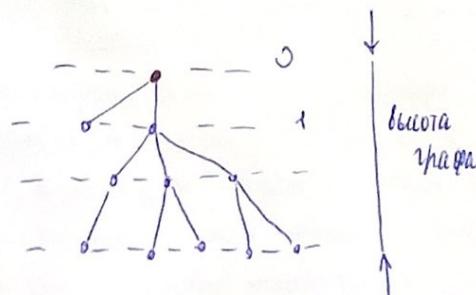
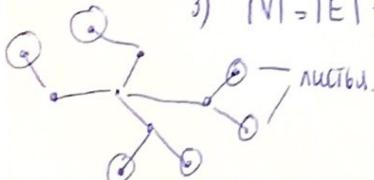
### Деревья

Дерево - граф (связный), в котором:

1) между  $V_1, V_2 \in V$  есть максимум одного члена

2) нет циклического члена

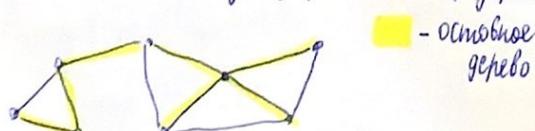
3)  $|V| = |E| + 1$  ( $\text{ни-бо } \leftrightarrow 1$ )



Компактное дерево - дерево, в котором одна из вершин - корень

Дерево - пустой граф и др-са.

Основное дерево - подграф связного графа, вышагивающий все из вершин из  $V_{\text{ог}} = V_0$ , являющийся деревом



Число вершин  
Число ребер  
 $V = \{ 'A', 'B', 'C', 'D' \}$   
 $E = \{ ('A', 'B'), ('B', 'C'), ('C', 'D') \}$

Хранение графа в виде матрицы  
Матрица смежности  
 $V = [ 'A' 'B' 'C' 'D' ]$   
 $\text{index} = \{ V[i]: i \text{ for } i \text{ in range}(len(V)) \}$   
 $A = [ [0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0] ]$

	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	0

столбец вершины,  
строка смежность

Словарь смежности  
 $G = \{ 'A': \{ 'B' \}, 'B': \{ 'A', 'C' \}, 'C': \{ 'B', 'D' \}, 'D': \{ 'C' \} \}$

Напечатать соседей (с помощью цикла for):  
 $\text{for neighbour in } G[V]:$   
 $\quad \text{print(neighbour)}$

### Алгоритм чтения графов

$N, M$   
 $\begin{matrix} A & b \\ B & C \\ C & D \end{matrix}$  в строку

$M, N = \text{map(int, input().split())}$  N - количество вершин, M - количество ребер  
 $G = [ [0] * N for _ in range(N) ]$  создание изображения матрицы смежности  
 $V\_names = [ ]$  список вершин  
 $\text{for } v \text{ in range(M)}:$  добавление ребра  
 $\quad V1, V2 = \text{input().split()}$

$\text{for } v \text{ in } (V1, V2):$   
 $\quad \text{if } v \text{ not in } V\_names:$

$\quad V\_names.append(v)$   
 $i1 = V\_names.index(V1)$   
 $i2 = V\_names.index(V2)$

$G[i1][i2] = 1$   
 $G[i2][i1] = 1$

$\text{print(''.join(V\_names))}$  join возвращает строку из всех элементов

$\text{for line in } G:$  берём строку смежности

$\text{print(''.join(map(str, line)))}$

### Функции сортировки вершин

$\text{for } V\_ind, V\_name \text{ in enumerate(V\_names):}$

$\text{print(V\_name, end=' ')} \quad \text{здесь}$

$G[V\_ind].count(1) \quad \text{количество единиц в строке}$

$* [V\_names[i] for i in range(N) if G[V\_ind][i] == 1]$  уменьшить количество

### Алгоритм обхода шириной на python

$N, M = \text{map(int, input().split())}$

$G = \{ \}$   
 $\text{for } v \text{ in range(M):}$   
 $\quad V1, V2 = \text{input().split()}$   
 $\quad \text{for } v \text{ in } (V1, V2):$   
 $\quad \quad \text{if } v \text{ not in } G:$   
 $\quad \quad G[v] = [ ]$   
 $\quad \quad G[V1].append(V2)$   
 $\quad \quad G[V2].append(V1)$   
 $\text{print}(G)$

76

```

A
N, M = map(int, input().split())
G = [{}]
for _ in range(N):
    for _ in range(M):
        v1, v2 = map(str, input().split())
        G[v1].append(v2)
        G[v2].append(v1)
for i, j in enumerate(G):
    print(i, len(j), *j)

```

### Алгоритм наименшего остекла с помощью цепей

```

edges = []
offset = 0
for i in range(N):
    for e in i:
        edges.append(e)
    offset.append(len(edges))
print(edges)
print(offset)
for v_i in range(len(offset)-1):
    print(v_i, begin[i])
    offset[v_i+1] = offset[v_i] + edges[offset[v_i]:offset[v_i+1]] + 1

```

### Обход цепи в ширину

```

def dfs(G, vertex, used=None):
    if used is None:
        used = set()
    used.add(vertex)
    for neighbour in G[vertex]:
        if neighbour not in used:
            dfs(G, neighbour, used)

```

to runaca kai-bo kaunokrem obyektom b'nefument.  
G=read\_graph(...)

used = set()  $\leftarrow N=0$

for vertex in G.keys():
 if vertex not in used:
 dfs(G, vertex, used)
 N+=1

### Алгоритм (аналог обхода) поискать

найти компоненты из спарка используя def read\_graph с обратным типом G.inv - узлы изолированные цепи  
нашли используя оп-усо dfs: (с обратным в конец) dfs(G, vertex, used, stack)

Найдем все остальные компоненты stack.append(vertex) добавить хоста на конец списка иначе не пойдет

G, G\_inv = read\_graph(...)

used = set()

stack = []

for vertex in G.keys():
 if vertex not in used:

dfs(G, vertex, used, stack)
 print(stack)

N=0

used = set()

while len(stack) > 0:

v = stack.pop()

if v not in used:

component = []

dfs(...)

print(component)
 N+=1

### Извлечь цепи из спарка

def read\_graph(filename):

```

N=M=None
G={}
for line in open(filename, 'r'):
    if N is None:
        N, M = map(int, line.split())
        continue
    v1, v2 = line.split()

```

for v in v1, v2:

if v not in G:

G[v] = []

G[v1].append(v2)
 G[v2].append(v1)

### Автоматическое BFS

```
from queue import Queue
N=0
def bfs(G, vertex, used=None):
    N=0
    if used is None:
        used = set()
    q = Queue() создание очереди
    q.put(vertex) наименование вершины, с которой будет начинаться поиск
    while not q.empty():  пока очередь не пуста
        v = q.get() берем вершину с очереди
        if v not in used: если вершина еще не посещена
            N += 1
            used.add(v) запоминаем вершину и список посещенных
            for neighbour in G[v]:
                if neighbour not in used:
                    q.put(neighbour)
```

Также есть задача, в которой используется bfs

- 1) *наименование начальной вершины start* go *найти все вершины*

```
def read_graph(...):
    from queue import Queue
    def bfs(G, start):
        distance = {v: None for v in G} создаем список начальных
        q = Queue() создание очереди
        q.put(start)
        distance[start] = 0 номера от начальной вершины равно 0
        while not q.empty():
            v = q.get()
            for neighbour in G[v]:
                if distance[neighbour] is None:
                    q.put(neighbour)
                    distance[neighbour] = distance[v] + 1
    return distance
```

$\min(\text{start}, \text{end})$   $\rightarrow$  start  $\longrightarrow$  end

$\min(\text{start}, v) + \min(\text{end}, v)$  *требует, чтобы вершина v возвращала номер*

$\min(\text{start}, \text{end}) = \min(\text{start}, v_1) + \min(\text{end}, v_2) + 1$  *требует, чтобы вершина вернула сумму v1+v2*

- 3) *оу-из библиотеки* *найти от start до end*

```
def bfs(G, start):
    path = {v: None for v in G}
    q = Queue()
    q.put(start)
    path[start] = ['A'] изменение с + jogaev
    while not q.empty():
        v = q.get()
        for neighbour in G[v]:
            if path[neighbour] is None:
                q.put(neighbour)
                path[neighbour] = path[v] + [neighbour]
    return path
```

### Обход графа в ширину

```
def read_files(...):
    f = open('task1.txt')
```

def read\_graph(...):  
 f = open('task1.txt')  
 G = {}  
 for line in f:  
 v, \*neighbours = line.strip().split()  
 G[v] = set(neighbours)  
 f.close()  
 return G

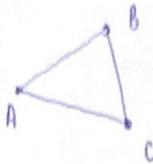
2) *наименование вершины start* go *найти все вершины*

```
def read_graph(...):
    def path(G, start, end):
        distance = bfs(G, start) оу-из bfs уж + задачи
        res = [end]
        while distance[end] != 0:
            for v in G[end]:
                if distance[end] - distance[v] == 1:
                    end = v
                    res.append(end)
                    break
        return res[::-1] -развернутый список
```

4) *оу-из библиотеки* *найти вершины от start до end*

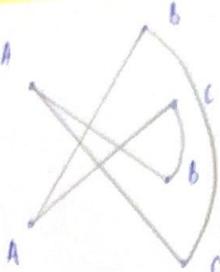
```
def bfs(G, start):
    parent = {v: None for v in G}
    q = Queue()
    q.put(start)
    parent[start] = None
    while not q.empty():
        v = q.get()
        for neighbour in G[v]:
            if parent[neighbour] is None:
                q.put(neighbour)
                parent[neighbour] = v
    return parent
```

5) Докажите математическим методом, что он является правильной



Уголом в

уголье



Можно, значит путь есть из A в B и.

Нулю нынѣ от A (4) и B (4), как известно

алгоритм:

```
def read_graph(...):
```

```
{    ...  
    for v in v1, v2:  
        if v + '0' not in G:  
            G[v + '0'] = set()    # пустой  
            G[v + '1'] = set()    # пустой  
        G[v1 + '0'].add(v2 + '1')  
        G[v2 + '0'].add(v1 + '1')  
        G[v1 + '1'].add(v2 + '0')  
        G[v2 + '1'].add(v1 + '0')  
    return G
```

```
from queue import Queue
```

```
def bfs(G, start):
```

Start = start + '0' # начинать с 0 заголовка

path = {v: None for v in G}

q = Queue()

q.put(start)

path[start] = [start]

while not q.empty():

v = q.get()

for neighbour in G[v]:

if path[neighbour] is None:

q.put(neighbour)

path[neighbour] = path[v] + [neighbour]

6) Задача о конусе (математика)

Составить задачу о конусе

G = dict()

```
for i in 'abcdefghijklmnopqrstuvwxyz':
```

```
    for j in '12345678':
```

G[i+j] = set()

```
for x in G:
```

```
    for y in G:
```

if abs((ord(x[0]) - ord(y[0])) + (ord(x[1]) - ord(y[1]))) = 2:

G[x].add(y)

G[y].add(x)

```
for i in G:
```

print(i, ':', G[i])

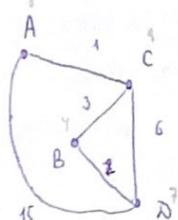
## 6) Дагалык с көмек (итеративное)

- Рассчитывает кратчайшего пути от start до end
- Число из текущей матрицы хранится против с соответствующими

from queue import Queue

```
def bfs(G, start, end):
    start = start
    path = {v: None for v in G}
    q = Queue()
    q.put(start)
    path[start] = start
    while not q.empty():
        v = q.get()
        for neighbour in G[v]:
            if path[neighbour] is None:
                q.put(neighbour)
                path[neighbour] = path[v] + [neighbour]
            if neighbour == end:
                return path[neighbour]
    return None
```

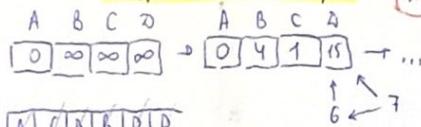
1) Вариант



Нанесение кратчайшего расстояния на графике (бумажный) (но если)

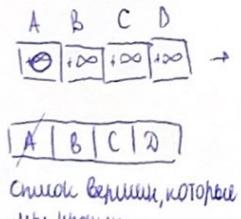
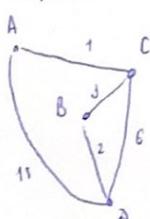
Алгоритм Дейкстры

храним C + весами



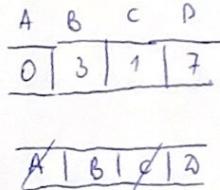
6 ← 7

2) Вариант



Черновик вершины, которую мы храним

где бывают мы вершины, которые до которых came минимальное → C



где алгоритм

def deikstra\_1(G, start):

```
distance = {v: float('inf') for v in G}
q = Queue()
q.put(start)
distance[start] = 0
while not q.empty():
    v = q.get()
    for neighbour in G[v]:
        tmp = distance[v] + G[v][neighbour]
        if distance[neighbour] > tmp:
            q.put(neighbour)
            distance[neighbour] = tmp
return distance
```

def deikstra\_2(G, start):

```
heap = []
distance = {v: float('inf') for v in G}
distance[start] = 0
heapq.heappush(heap, (0, start))
while len(heap) > 0:
    v = heapq.heappop(heap)[1] запись вершины из
    for neighbour in G[v]:
        tmp = distance[v] + G[v][neighbour]
        if distance[neighbour] > tmp:
            distance[neighbour] = tmp
            heapq.heappush(heap, (tmp, neighbour))
return distance
```

## алгоритм Форда-Форсмана

start

	A	B	C	D	E	F
0	+∞	+∞	0	+∞	+∞	+∞
1						
2						

У стартовой вершине начальное расстояние 0, а  
в остальных  $+∞$

$(u, v)$

$$\underline{A} \quad \underline{B} \quad q \rightarrow +∞ + q = +∞$$

$$d = [+∞] * N \Rightarrow d[0][start] = 0$$

for i in range :

for x,y,w in Edges :

$$d[i][y] = \min(d[i-1][y], d[i-1][x] + w)$$

importано знать пары :  
x - начало ребра  
y - конец ребра  
w - вес ребра

$O(N \cdot M)$

## алгоритм Форда-Форсмана (ищем кратчайшее расстояние от всех вершин до всех)

	A	B	C	D	E	F	G
A	0	+∞	+∞	+∞	40	12	
B		0					
C			0				
D	.			0	+∞		
E					0		
F						0	
G							0

for k in range(N) :

for i in range(N) :

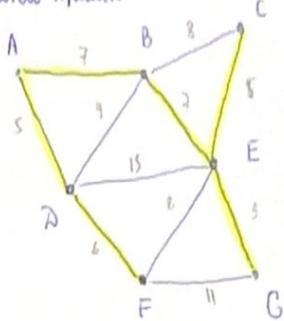
for j in range(N) :

$$d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

## Основные деревья

Задача: найти минимальное дерево дерева:

1) Автоматика



Выполнили процедуру вершины

1)  $AD < AB \Rightarrow AD$

2) далее из всех ребер, которые приближались к вершине, находила, т.е., который имеет минимальный вес  $\rightarrow AF$

3) ...  $\rightarrow AB$

4) исключили из  $BA \rightarrow BE$

5)  $EC$

$$5 + 7 + 7 + 6 + 8 = 39$$

Время  $O(M \cdot N)$

Есть 2 варианта реализации: 1) наивный, простой; если [ ]-вершина, откуда начинается некое ребро, которое будет хранить используемые вершины, будут в списке - блок. Вершины в неё нет. Проходится по всем ребрам из used-[ ], которые будут в вершину и в списке - находят минимальную

2) использует связанный список матриц

$O(M \cdot \log N)$

A	B	C	D	E	F	G
100	120	120	120	100	100	100
↓	↓	↓	↓	↓	↓	↓
5	7	2	15	6	8	3
A	B	C	D	EF	FG	G

согласно строкам с минимальными расстояниями от конечных объектов

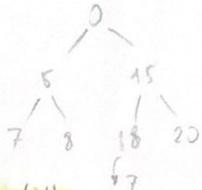
до вершин

$AD \rightarrow B$  chosen

$DF$

$AB$

...



def shift-down(id):

hxt = head[id][0]  
if id\*2+1 < len(heap): проверка на наличие ребенка,  
nxt = min(hxt, heap[2\*id+1][0]) находят минимальный из 3-х

if id\*2+2 < len(heap):

nxt = min(hxt, heap[2\*id+2][0])

if heap[id][0] == nxt: если текущий блк = min, то меняем его значение  
return

if heap[2\*id+1][0] == nxt: если второй ребенок - min, меняем значение

next\_id = 2\*id+1

heap[next\_id], heap[id] = heap[id], heap[next\_id]

heap[next\_id][1] = next\_id

heap[id][1] = id

shift-down(next\_id)

return

[также cause, но для  $2*id+2$ ]

if id == 0: (если это корень не сравниваем с

ребенком, то не меняем)

parent = (id-1) // 2

if heap[parent][0] > heap[id][0]: меняем элемент - блок parent

heap[parent], heap[id] = heap[id], heap[parent]

heap[parent][1] = parent

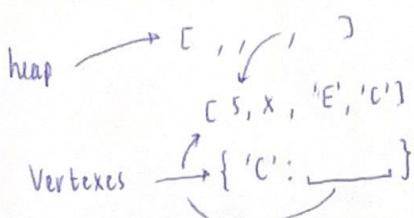
heap[id][1] = id место хранения блока x

shift-up(parent)

```

# программа для поиска минимального остовного дерева
def read_graph():
    N, M = map(int, input().split())
    G = {}
    for _ in range(M):
        v1, v2, w = input().split()  # w - вес
        if v1 not in G:
            G[v1] = {}
        if v2 not in G:
            G[v2] = {}
        G[v1][v2] = float(w)
        G[v2][v1] = float(w)
    return G

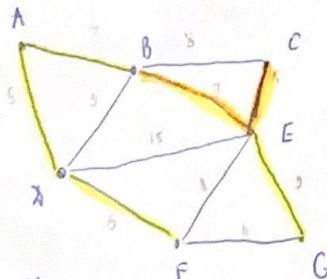
```



```

vertexes_sets = []
def make_set(x):
    return {
        'parent': x,
        'count': 1
    }
def find_set(x):
    if vertexes_sets[x]['parent'] == x:
        return x
    else:
        y = find_set(vertexes_sets[x]['parent'])
        vertexes_sets[x]['parent'] = y
    return y
def union_set(x, y):
    x, y = map(find_set, (x, y))
    if vertexes_sets[x]['count'] < vertexes_sets[y]['count']:
        min_weight = vertexes_sets[x]['count']
        min_set = x
        max_set = y
    else:
        min_weight = vertexes_sets[y]['count']
        min_set = y
        max_set = x
    vertexes_sets[min_set]['parent'] = max_set
    vertexes_sets[min_set]['count'] += vertexes_sets[max_set]['count']

```



**def prim(G):**

start = next(iter(G.keys())) # стартовая вершина

Vertices = {x: E[0] if ('inf', i, None, x) for i, x in enumerate(G.keys())}

union\_set, неизвестно, где находятся вершины, которые входят в минимальное дерево из N-ти

min\_weight = float('inf') # для первого шага

Vertices[start][0] = float('inf') # для первого шага

for v in G[start]: # инициализация

Vertices[v][0] = start # вершина откуда мы пришли от start, путь

Vertices[v][2] = start # предыдущая вершина - откуда

for k, w in Vertices.items(): # обработка всех вершин с пути, к ним

push(w)

pop() # вытащили вершину

res = [] # список ребер, которые будут в минимальном дереве

weight = 0 # суммарный вес

used = {start} # множество ребер

for \_ in range(len(G)-1): # Алгоритм Прима работает только для связного

v = pop() # выбираем вершину, которая минимальна

res.append((v[0], v[1])) # добавляем ребро в дерево, это будет из V[0] в V[1]

weight += v[0]

used.add(v[0]) # новая вершина

for n in G[v[0]]: # проходим по всем соседям вершины V[0]

if n not in used:

if G[v[0]][n] < Vertices[n][0]: # ребра не есть в множестве ребер

Vertices[n][0] = G[v[0]][n]

Vertices[n][2] = v[0]

shift\_up(Vertices[n][1])

return res, weight

**Автоматическое тестирование**  $O(m \log m + m \cdot \Theta(n))$

когда нужно проверить правильность работы алгоритма

- 1) ее помощь отображает то мин., а какуюто из вершин не вернет ближайшего ребра (если все-ба)
- 2) если она вернет какуюто вершину из набора мин-б, то работает

реализуется через auxiliary переменную. min-B ( $\text{make-set}(x)$ ,  $\text{union-set}(y, z)$ ,  $\text{find-set}(x)$ )

```

def key_el(x):
    for x,y,w in E:
        if find_set(x) != find_set(y):
            min_weight += w
            min-tree.append((x,y,w))
            union_set(x,y)
for _ in range(M):
    x,y,w = input().split()
    w = float(w)
    vertexes_sets[x] = make_set(x) # находит вершину
    vertexes_sets[y] = make_set(y) # находит вершину
    E.append((x,y,w)) # добавляем ребро (x,y,w)
E.sort(key=key_el) # сортируем с учетом
min-tree = [] # минимальное дерево
min_weight = 0

```