

# Алгоритмы на графах

## Лекция 2.

### Обходы графов. Поиск в ширину.

Адигеев Михаил Георгиевич

2024

# План лекции

1. Пути и достижимость на графах.
2. Обходы графов.
  - ✓ Обход в ширину
  - ✓ (Обход в глубину)
3. Применение обхода в ширину для решения задач на графах.
  - ✓ Поиск компонент связности
  - ✓ Поиск кратчайших путей на невзвешенных графах
  - ✓ Обнаружение циклов

# Пути и достижимость на графах

# Пути и достижимость

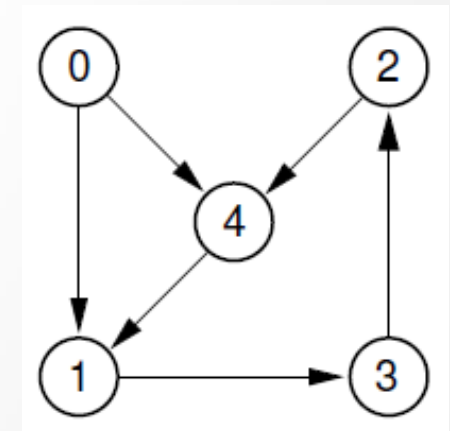
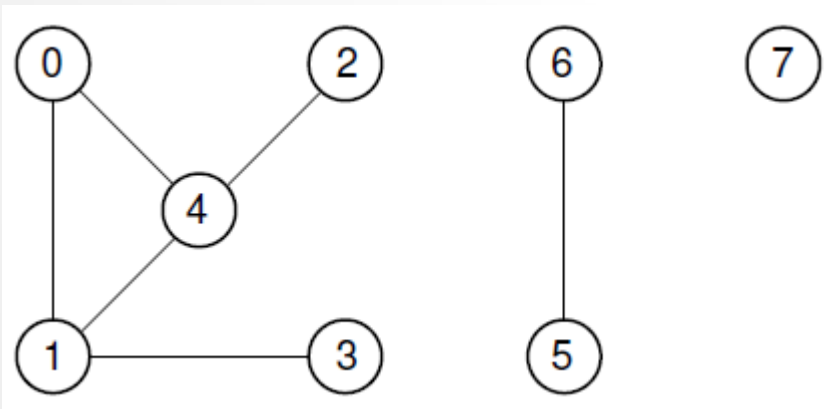
Граф  $G=(V,E)$ .

**Путь** на графе  $G$  – это последовательность дуг  $\{e_1, e_2, \dots, e_l\}$ , в которой для каждого  $i$  конец дуги  $e_i$  является началом дуги  $e_{i+1}$ .

Для неориентированных путей используется название «**цепь**».

Альтернативное представление – в виде последовательности вершин  $\{v_1, v_2, \dots, v_{l+1}\}$ .

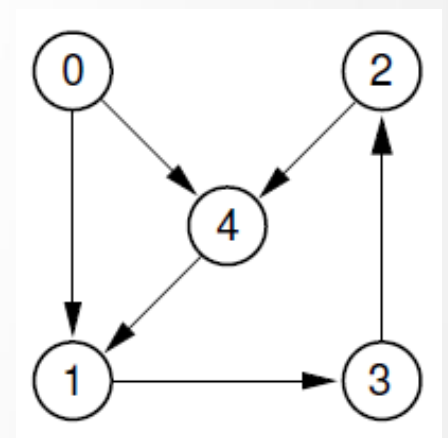
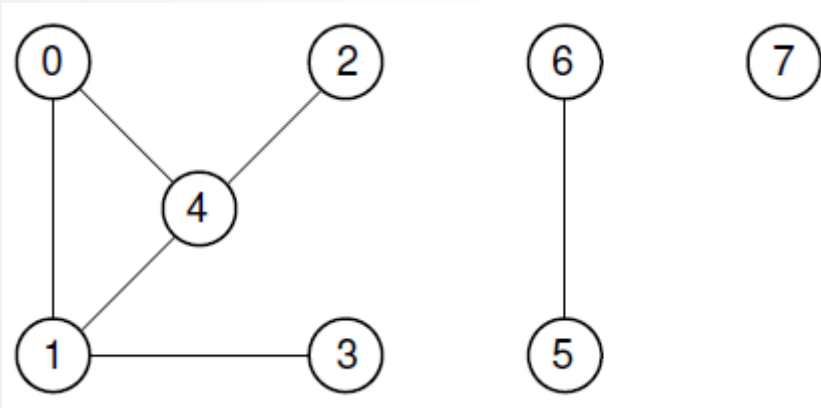
**Длина** пути/цепи = количество дуг/рёбер в пути.



# Пути и достижимость

**Циклом** называется замкнутая цепь, т.е. цепь, в которой конечная и начальная вершины совпадают.

Замкнутый ориентированный путь называется **контуром**.

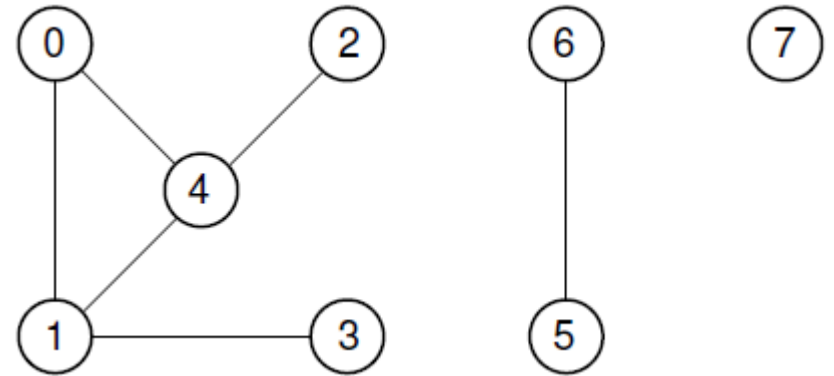


# Пути и достижимость

Говорят, что вершина  $v \in V$  **достижима** из вершины  $u \in V$ , если на графе существует путь/цепь из  $u$  в  $v$ .

Граф называется **связным**, если все пары вершин достижимы друг из друга.

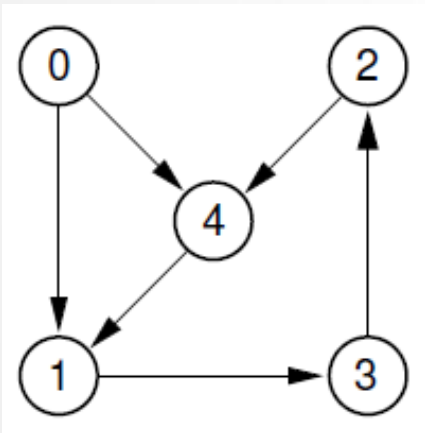
Максимальный связный *подграф* на графе называется **связной компонентой**.



# Пути и достижимость

Для ориентированных графов также вводят понятие *сильной связности*: граф **сильно связан**, если все пары вершин взаимно достижимы друг из друга.

И аналогично: максимальный сильно связный *подграф* на графе называется **сильной (сильно связной) компонентой**.

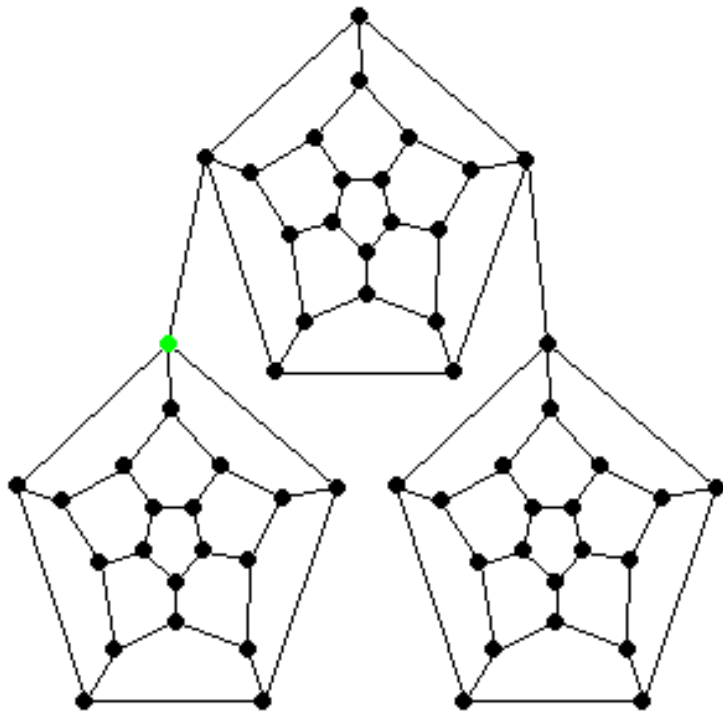


# Обходы графов



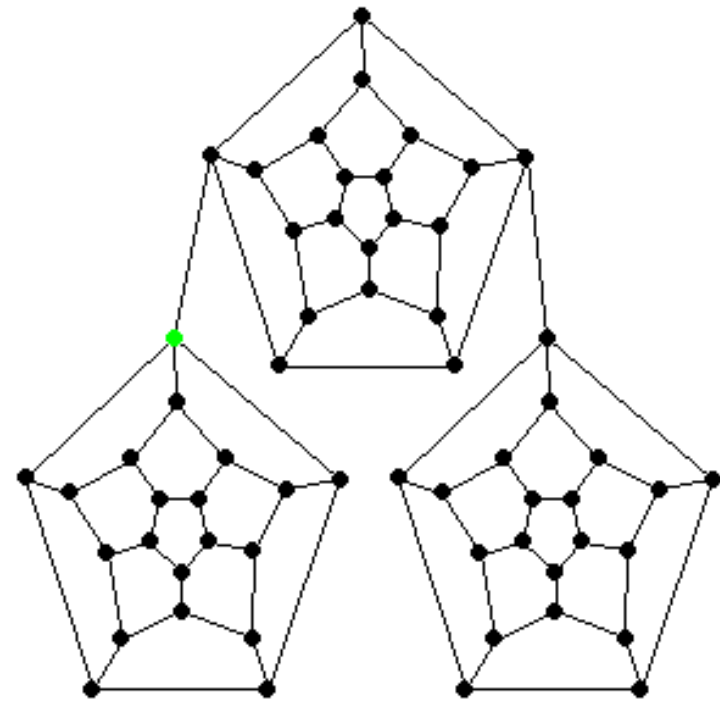
# Обходы графов

Depth-First Search



[www.combinatorica.com](http://www.combinatorica.com)

Breadth-First Search



[www.combinatorica.com](http://www.combinatorica.com)

<https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html>

# Общие принципы обхода

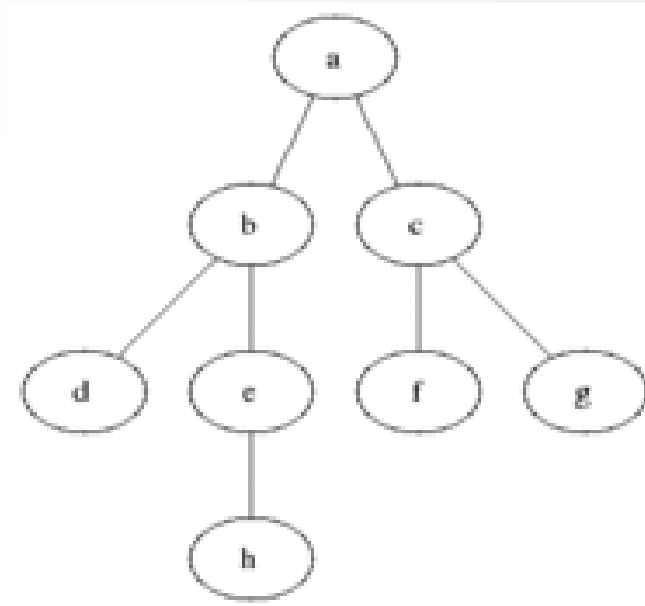
- Мы должны *обойти* все вершины графа в *определённом* порядке и *обработать* их.
- В процессе обхода для каждой вершины мы в какой-то момент посещаем её в первый раз, потом можем несколько раз возвращаться в неё, и в какой-то момент (обычно при первом или при последнем посещении) мы её обрабатываем.
- Для того чтобы корректно обработать вершину (а также инцидентные ей рёбра и смежные вершины), мы должны отслеживать *статус* вершины:
  - 1) Непосещённая
  - 2) Посещённая
  - 3) Обработанная.

# Обход в ширину

Принцип обхода в ширину:  
посетив вершину  $v$ , следует посетить каждую из ещё не посещённых соседних вершин.

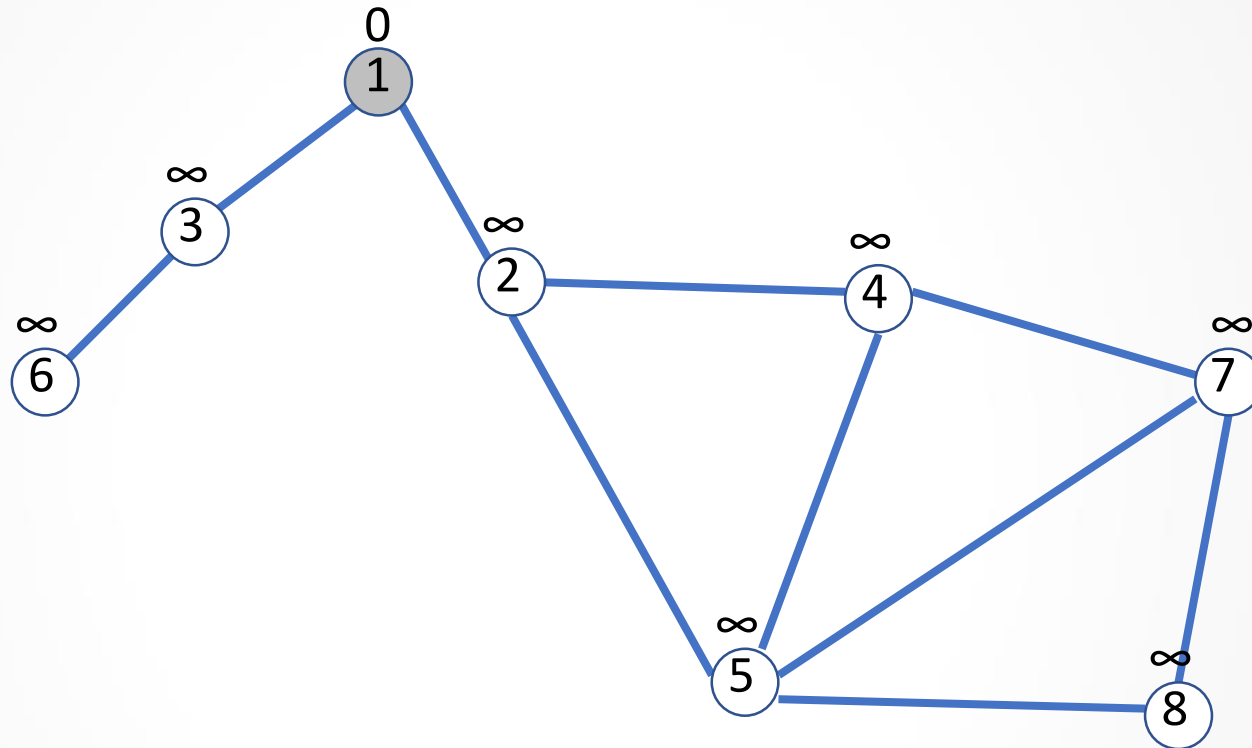
Потом соседей этих соседей, т.д.

Для того чтобы запоминать вершины, которые надо посетить, и порядок посещения, нам понадобится структура данных «Очередь» (queue).



[https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search)

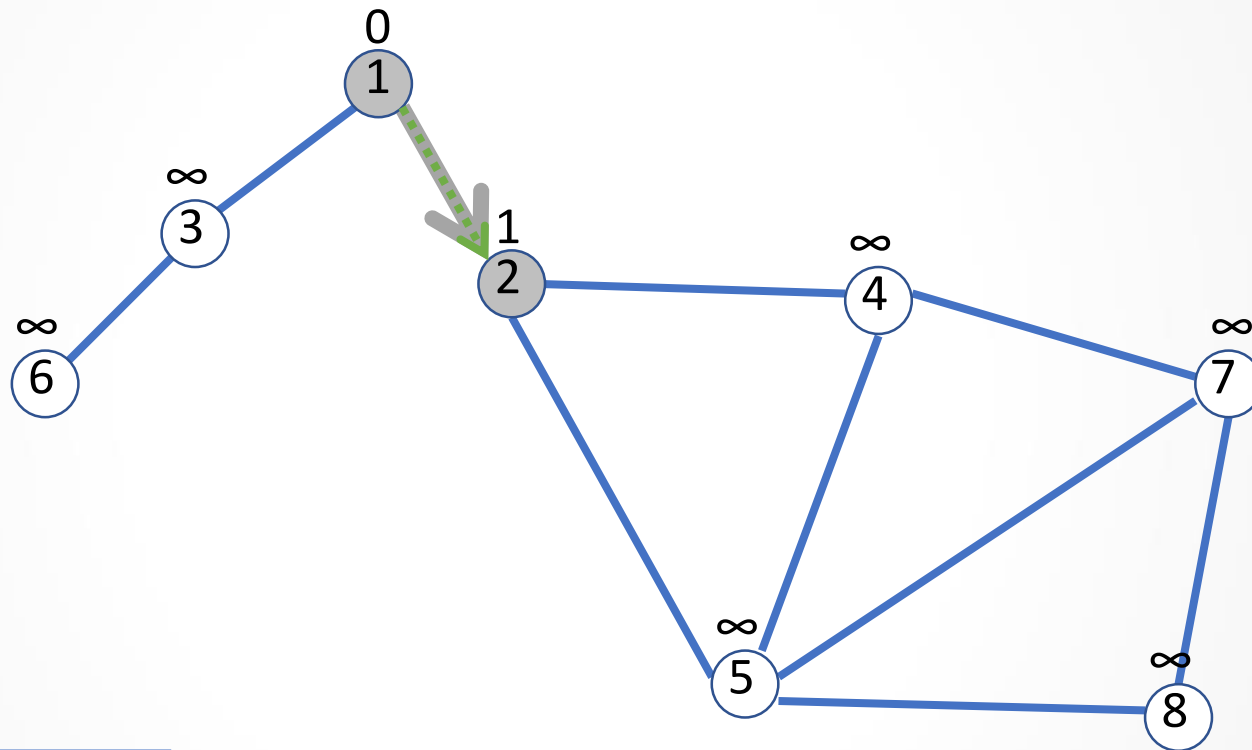
# Обход в ширину: пример



Q: 1

Просматриваемая дуга:  $\emptyset$

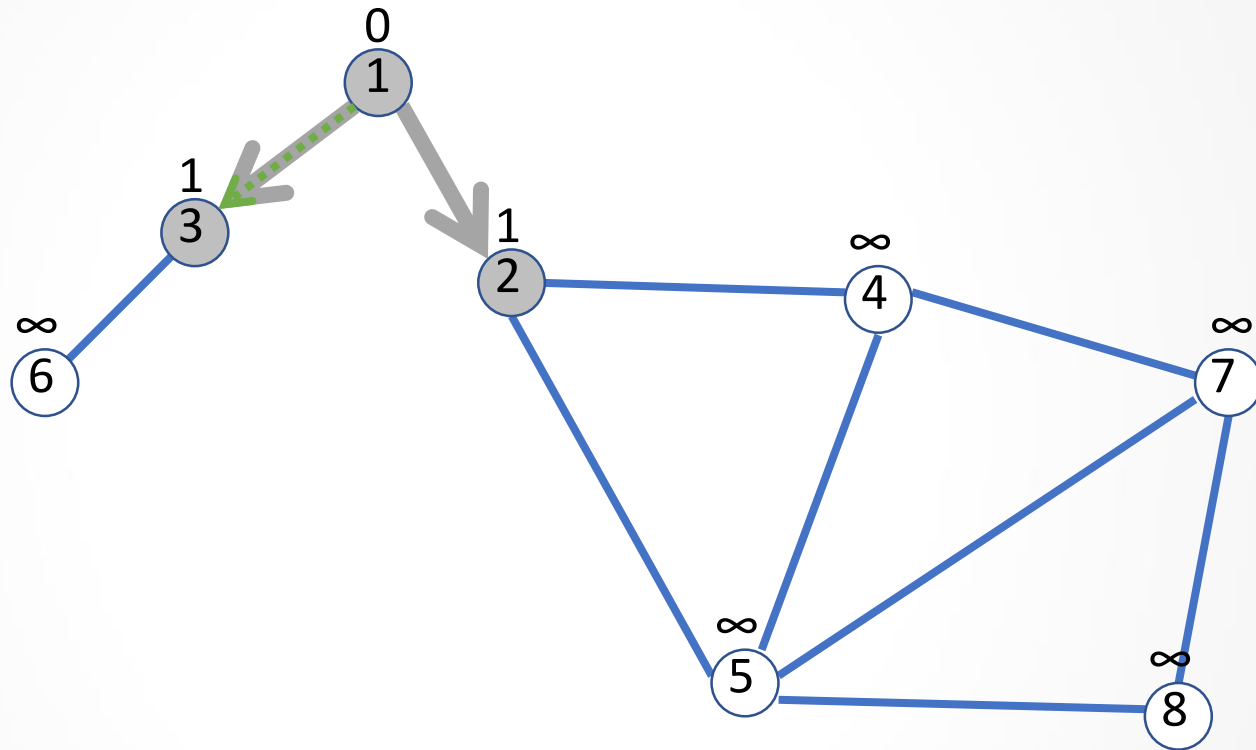
# Обход в ширину: пример



Q: 2

Просматриваемая дуга: (1,2)

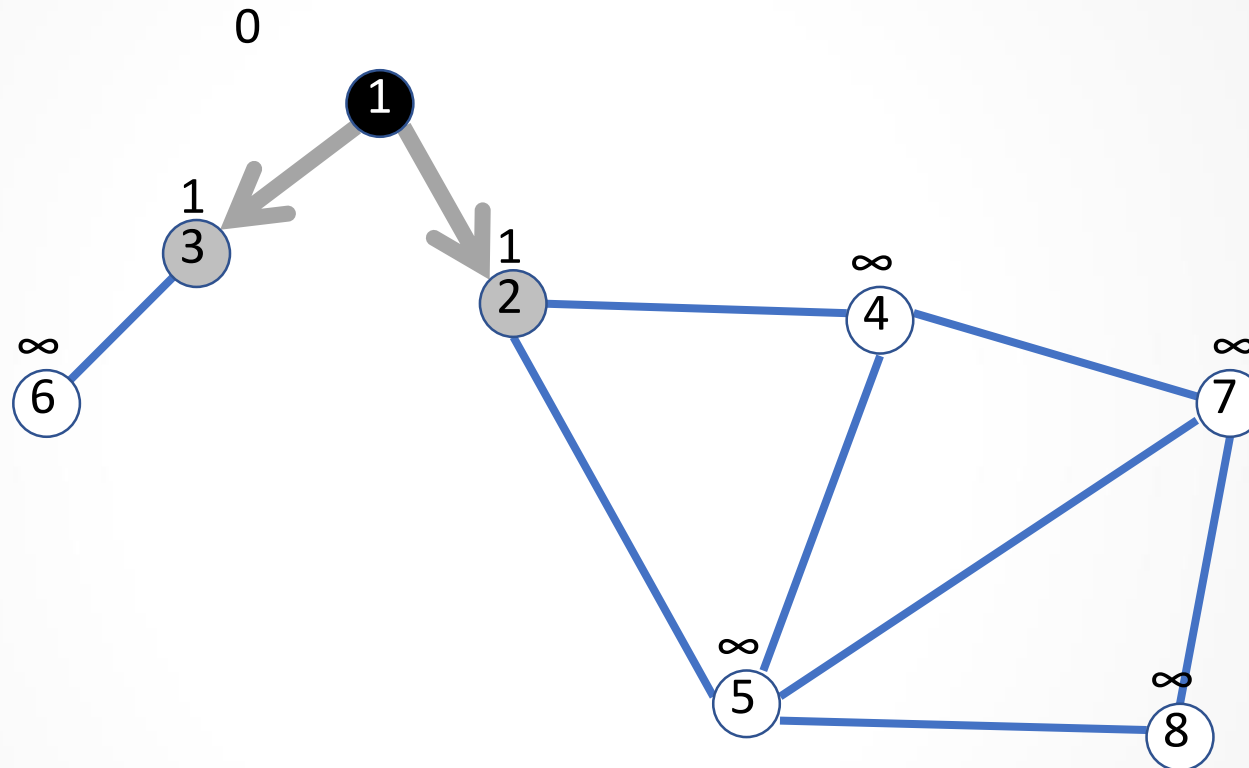
# Обход в ширину: пример



Q: 2,3

Просматриваемая дуга: (1,3)

# Обход в ширину: пример

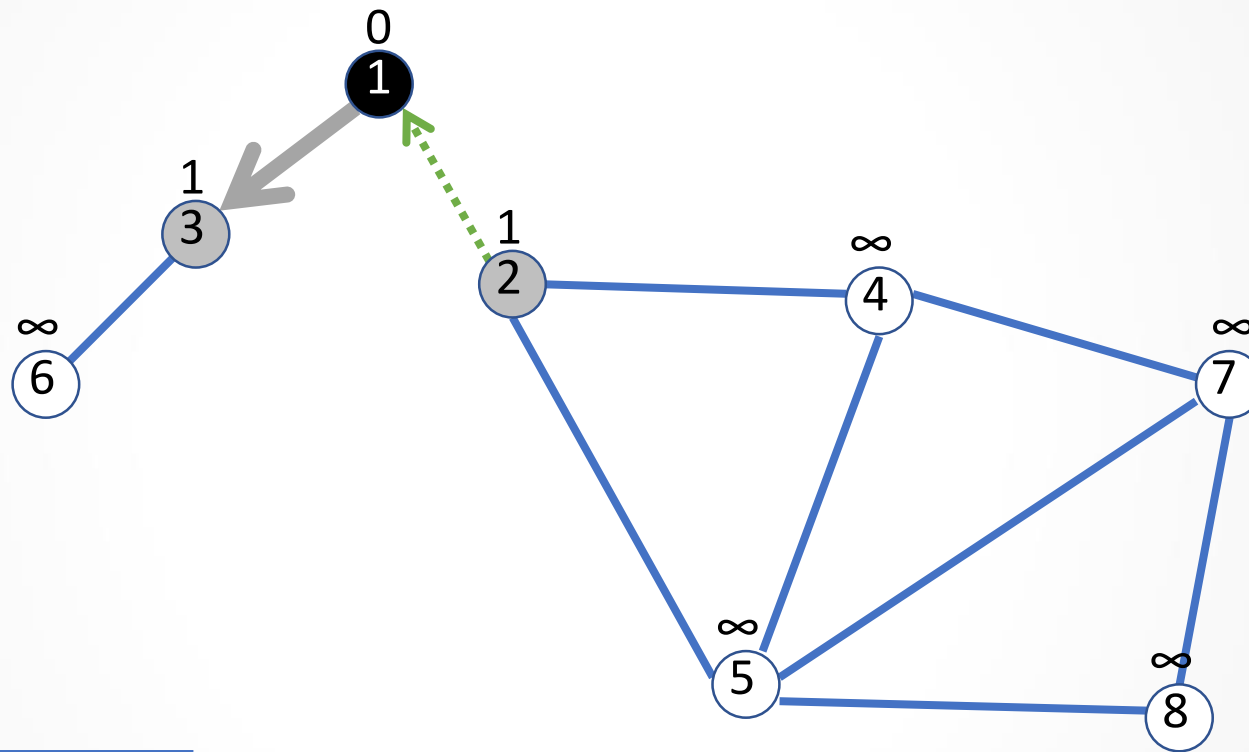


Q: 2,3

Просматриваемая дуга:  $\emptyset$

Вершина 1 становится *обработанной*

# Обход в ширину: пример

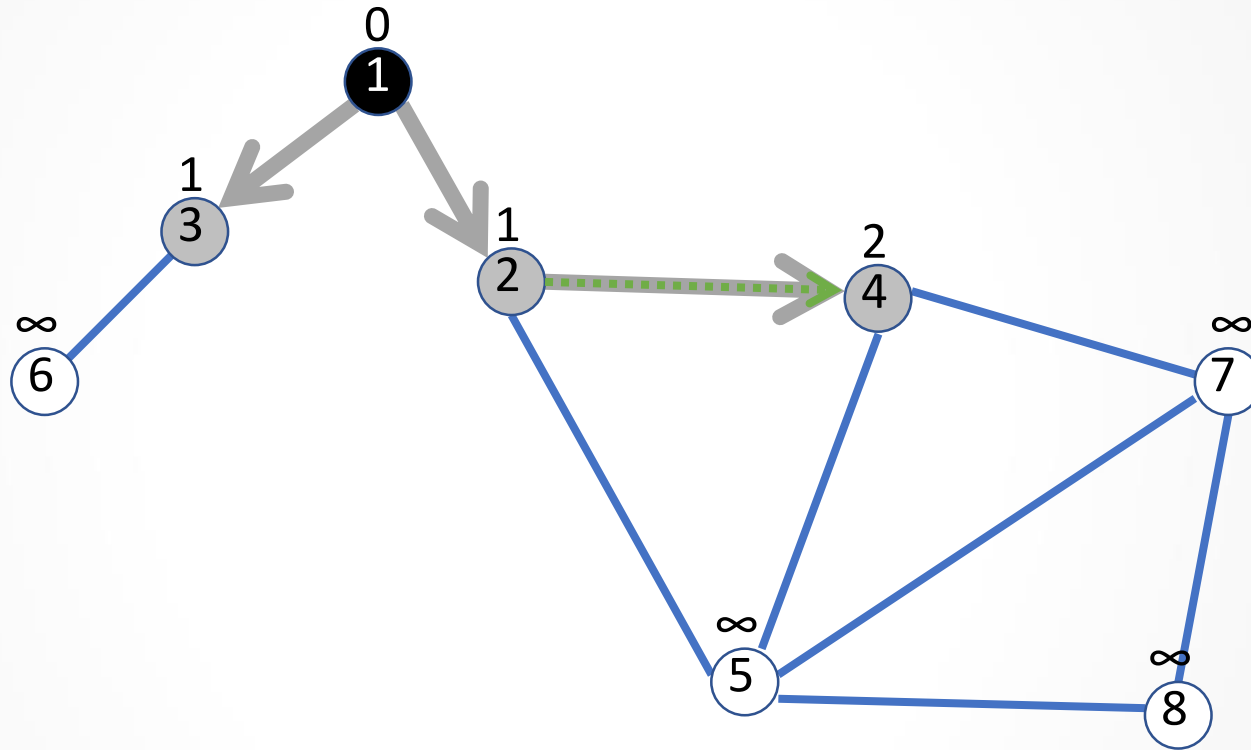


Q: 3

Просматриваемая дуга: (2,1)



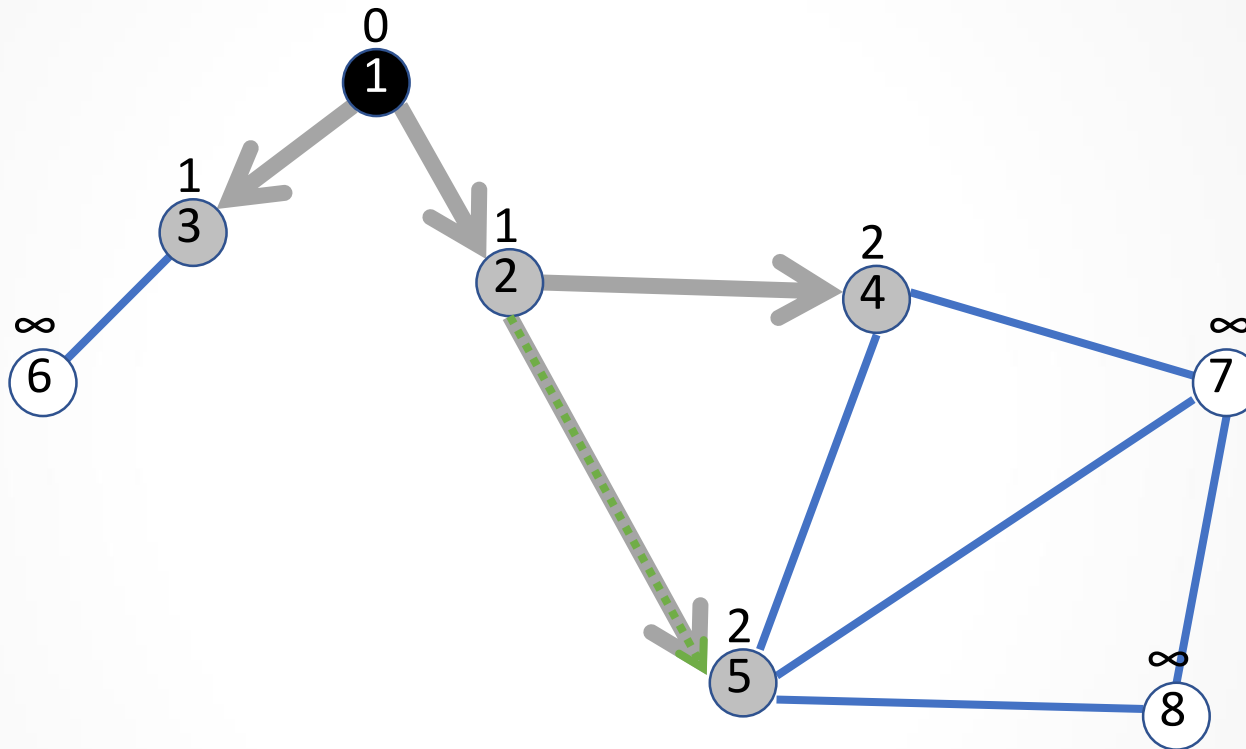
# Обход в ширину: пример



Q: 3,4

Просматриваемая дуга: (2,4)

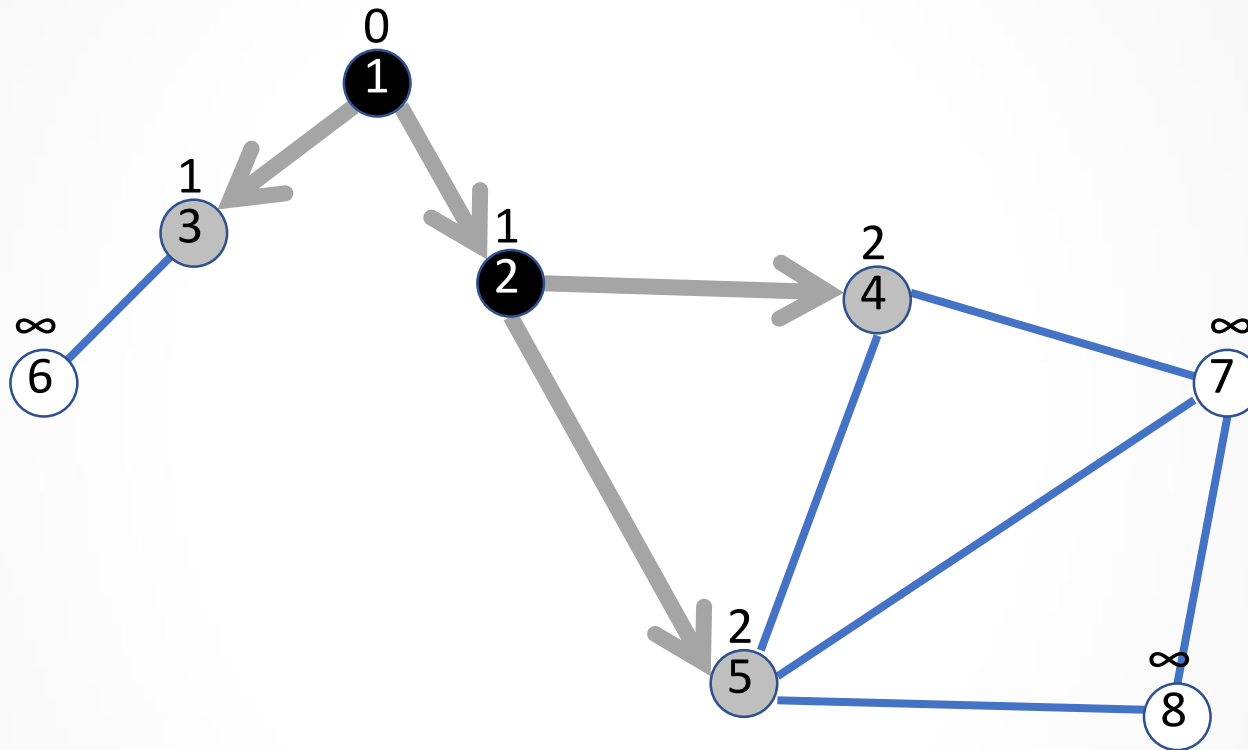
# Обход в ширину: пример



Q: 3,4,5

Просматриваемая дуга: (2,5)

# Обход в ширину: пример

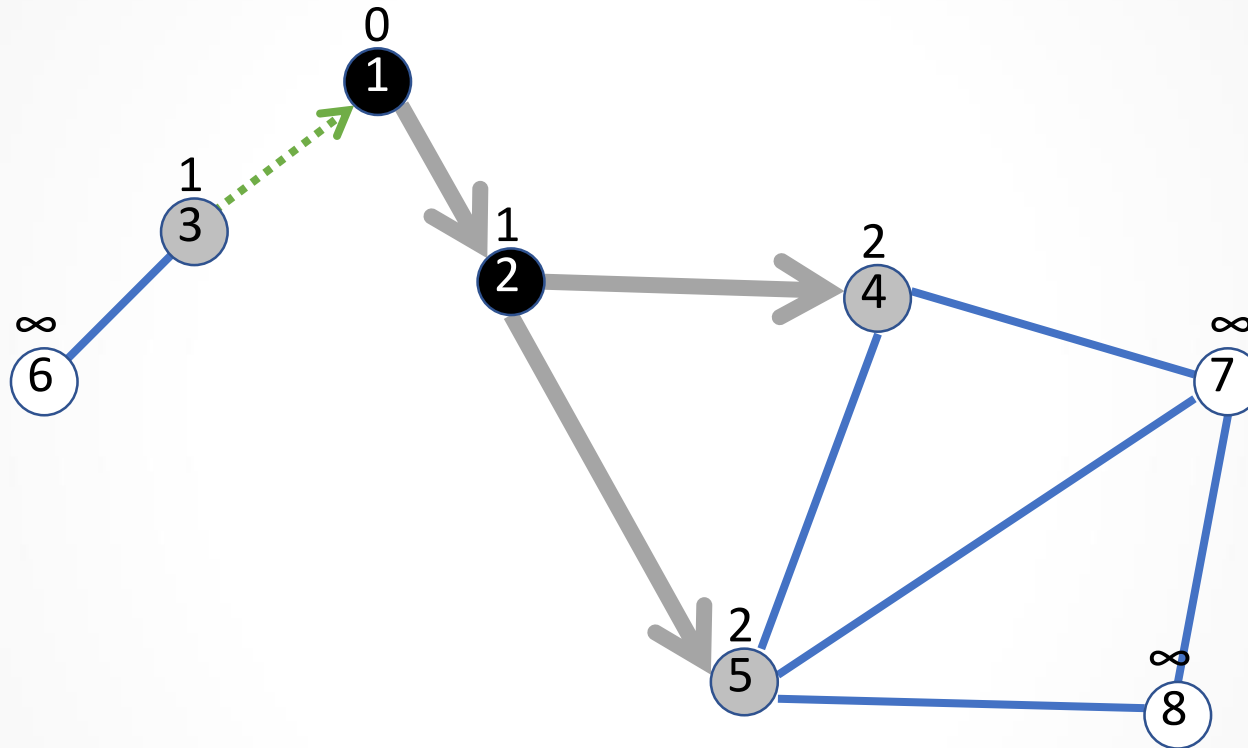


Q: 3,4,5

Просматриваемая дуга:  $\emptyset$

Вершина 2 становится *обработанной*

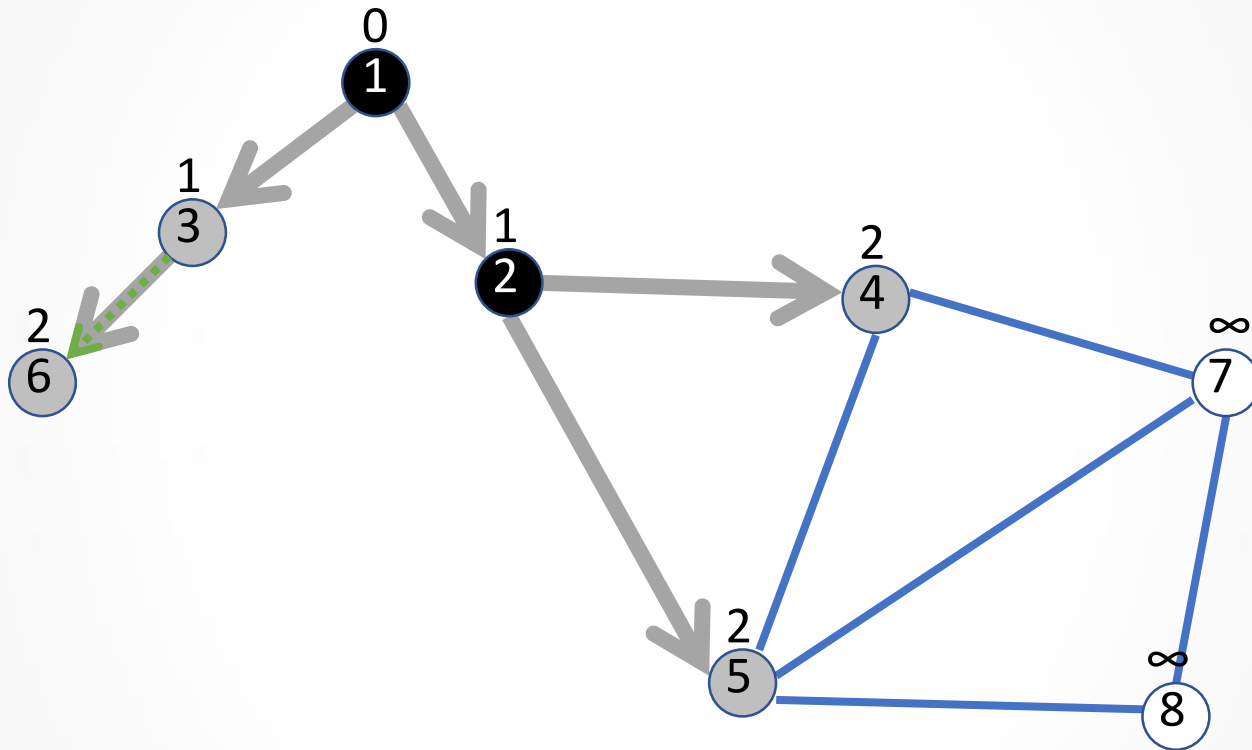
# Обход в ширину: пример



Q: 4,5

Просматриваемая дуга: (3,1)

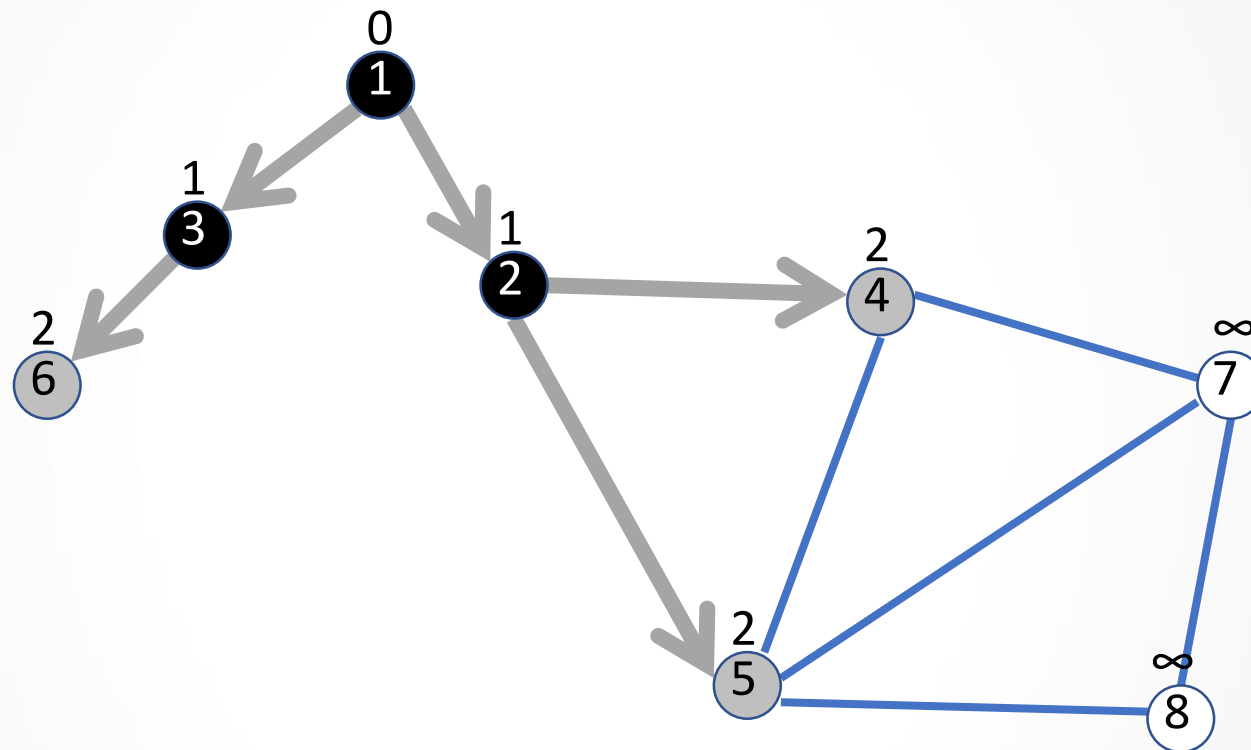
# Обход в ширину: пример



Q: 4,5,6

Просматриваемая дуга: (3,6)

# Обход в ширину: пример

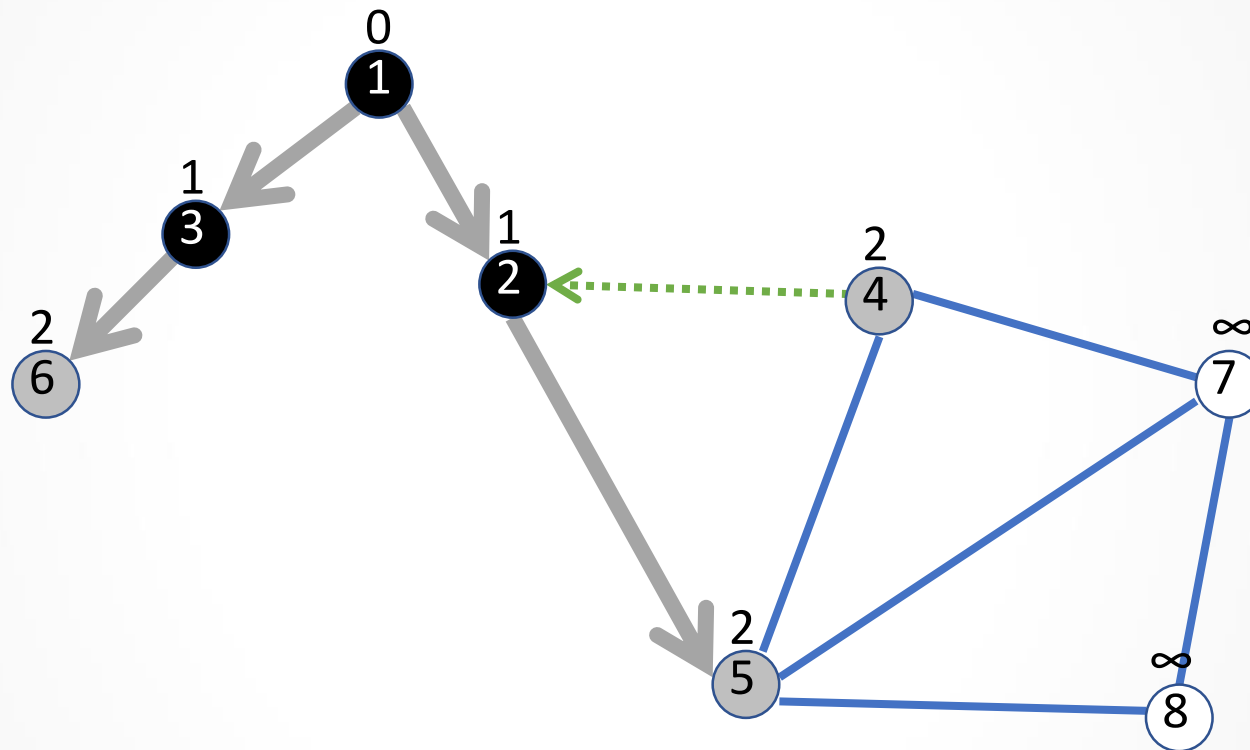


Q: 4,5,6

Просматриваемая дуга:  $\emptyset$

Вершина 3 становится *обработанной*

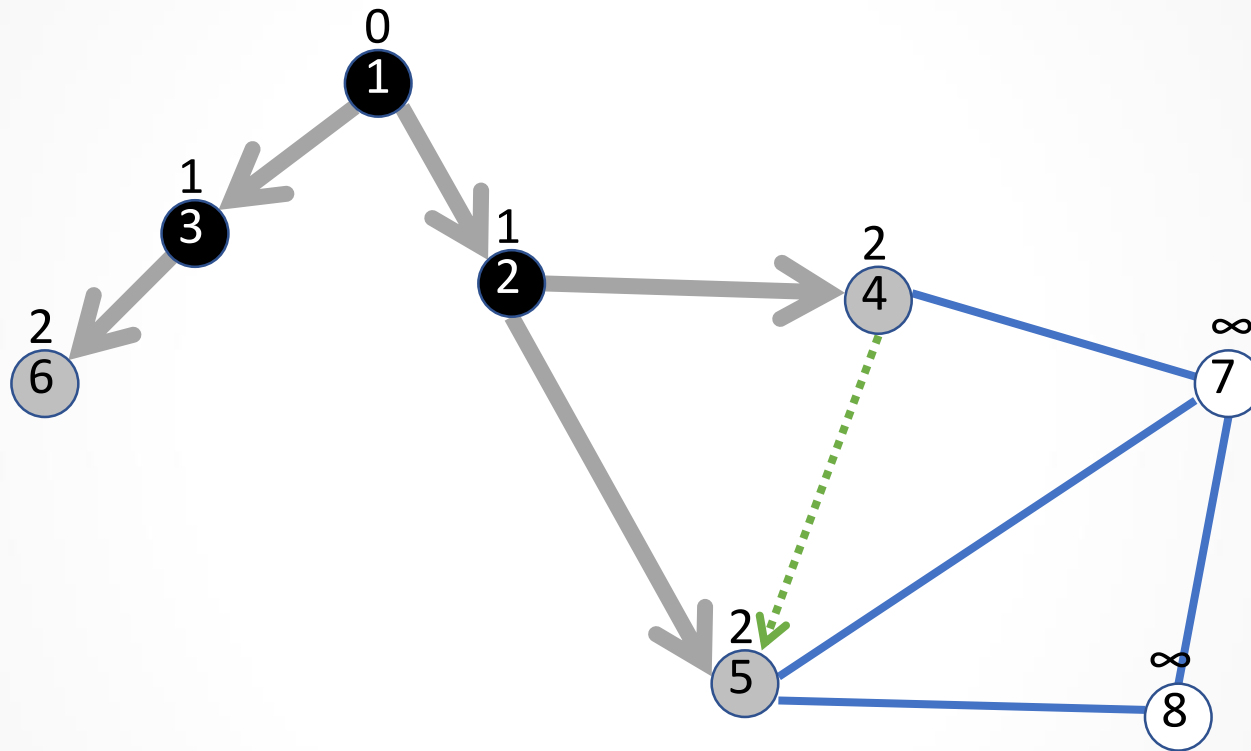
# Обход в ширину: пример



Q: 5,6

Просматриваемая дуга: (4,2)

# Обход в ширину: пример

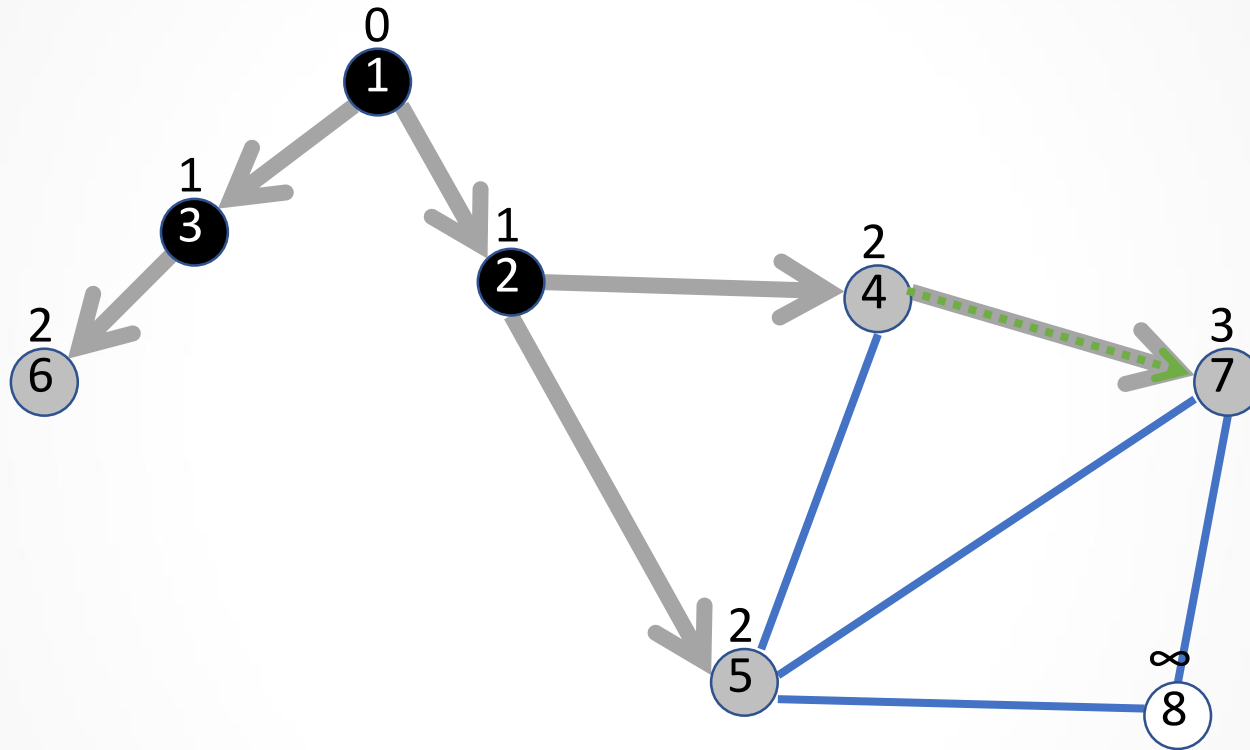


Q: 5,6

Просматриваемая дуга: (4,5)



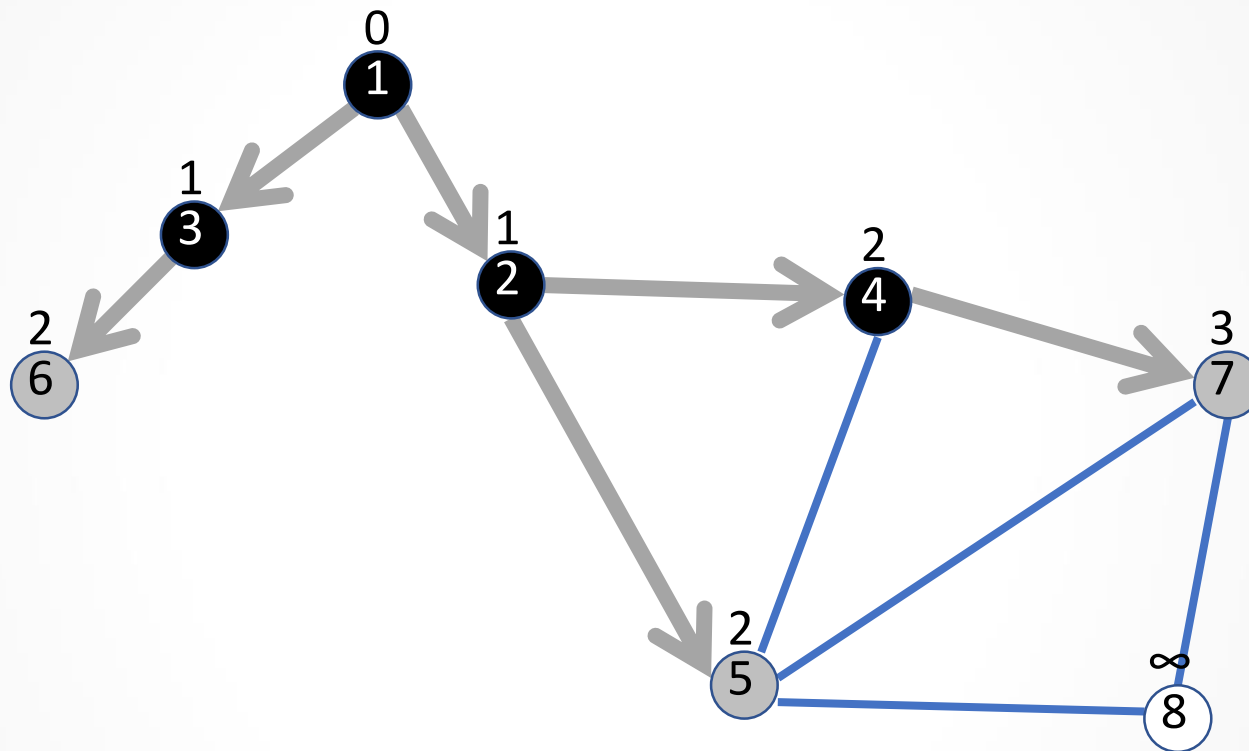
# Обход в ширину: пример



Q: 5,6,7

Просматриваемая дуга: (4,7)

# Обход в ширину: пример

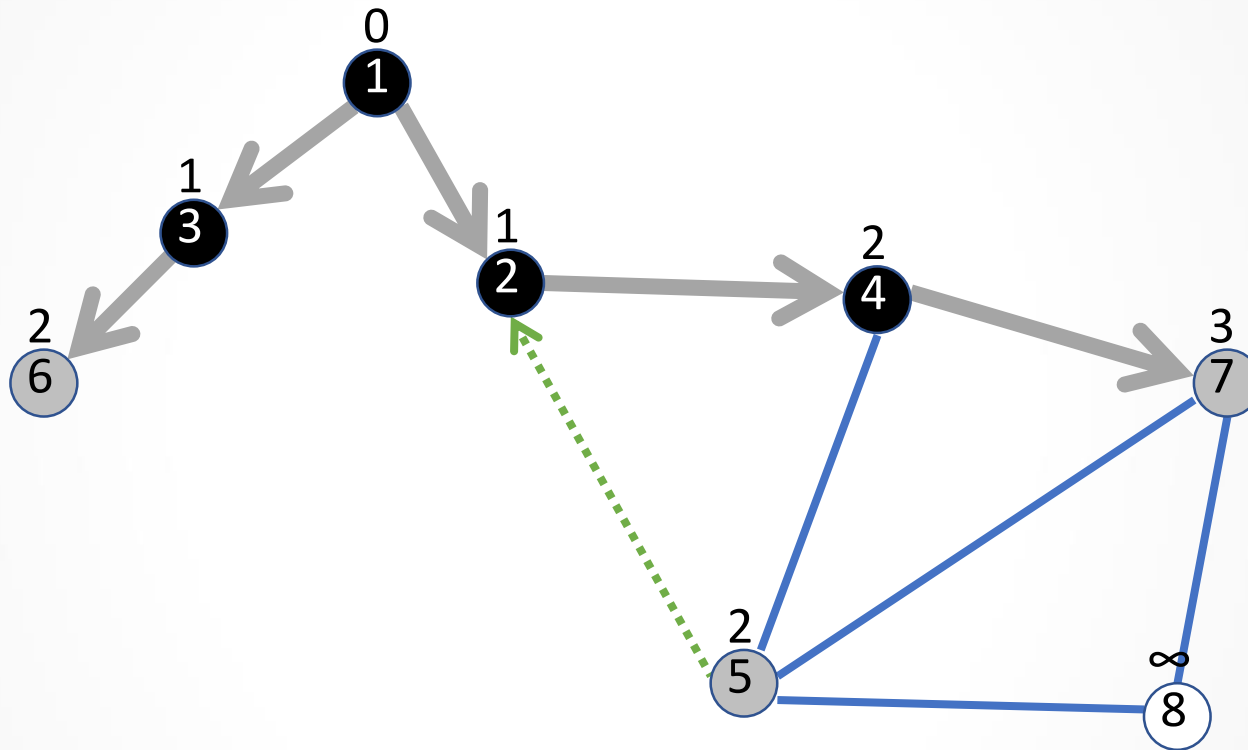


Q: 5,6,7

Просматриваемая дуга:  $\emptyset$

Вершина 4 становится обработанной

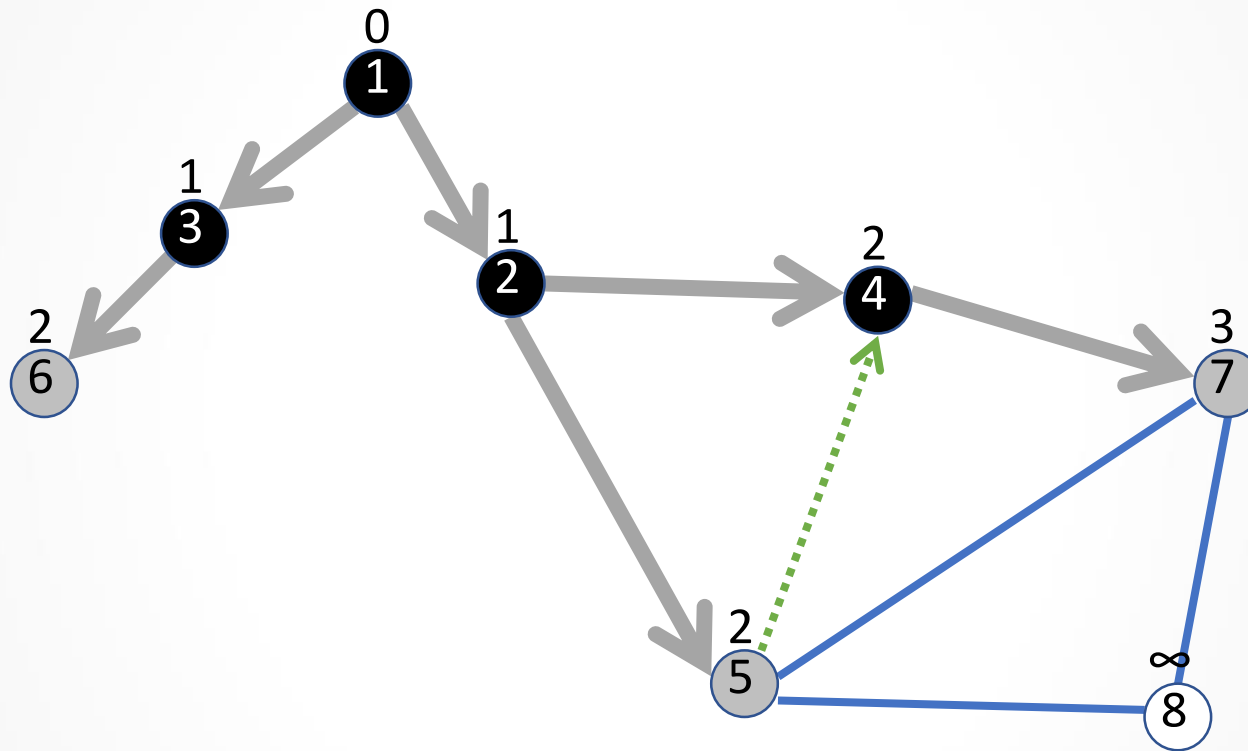
# Обход в ширину: пример



Q: 6,7

Просматриваемая дуга: (5,2)

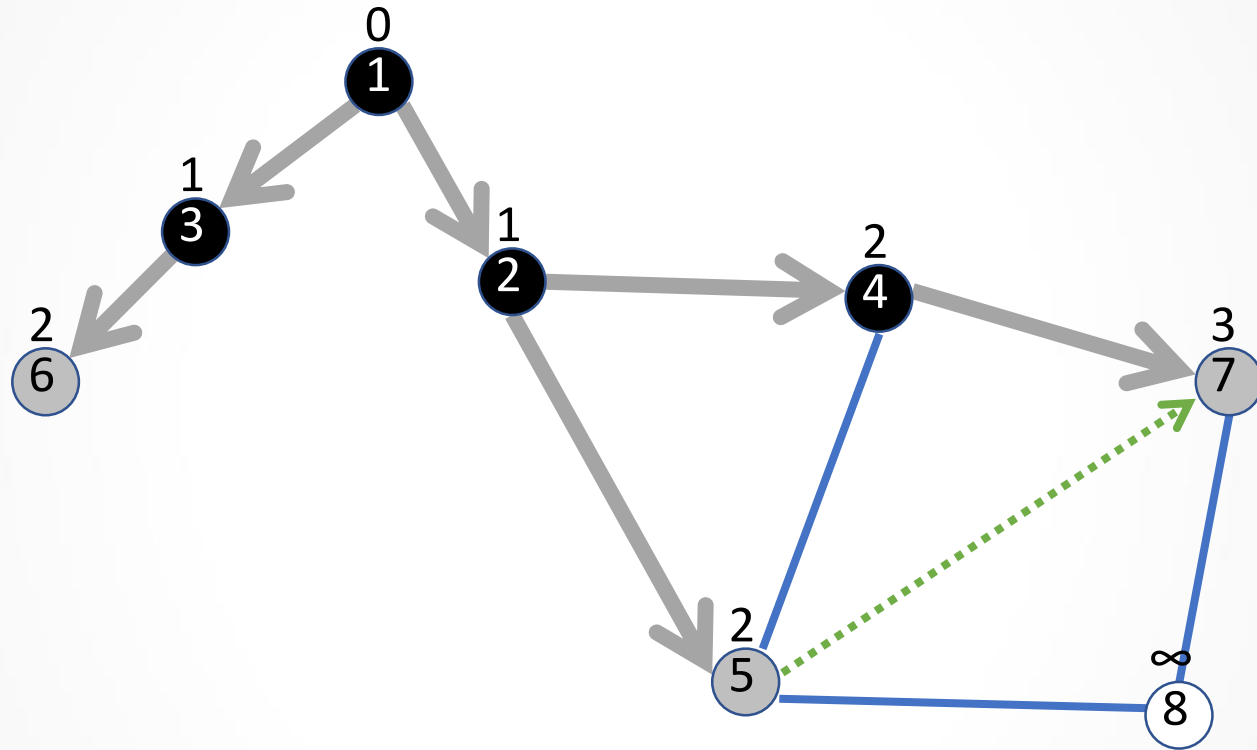
# Обход в ширину: пример



Q: 6,7

Просматриваемая дуга: (5,4)

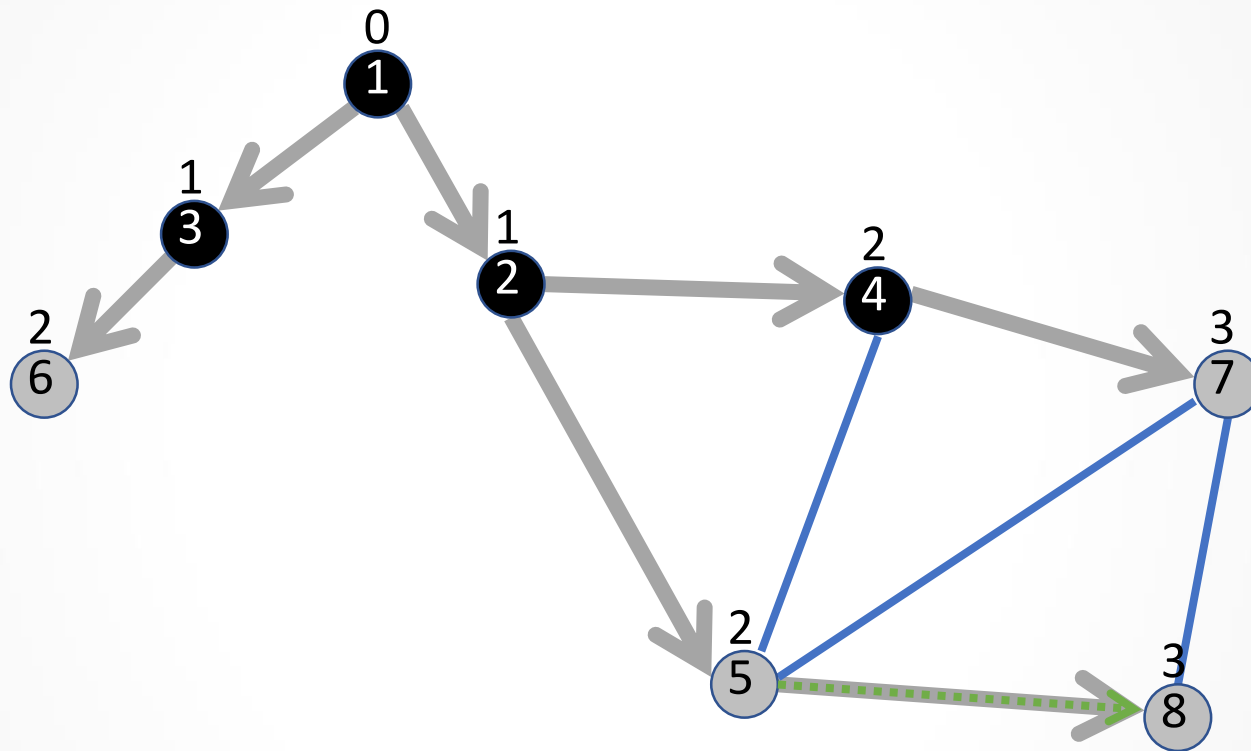
# Обход в ширину: пример



Q: 6,7

Просматриваемая дуга: (5,7)

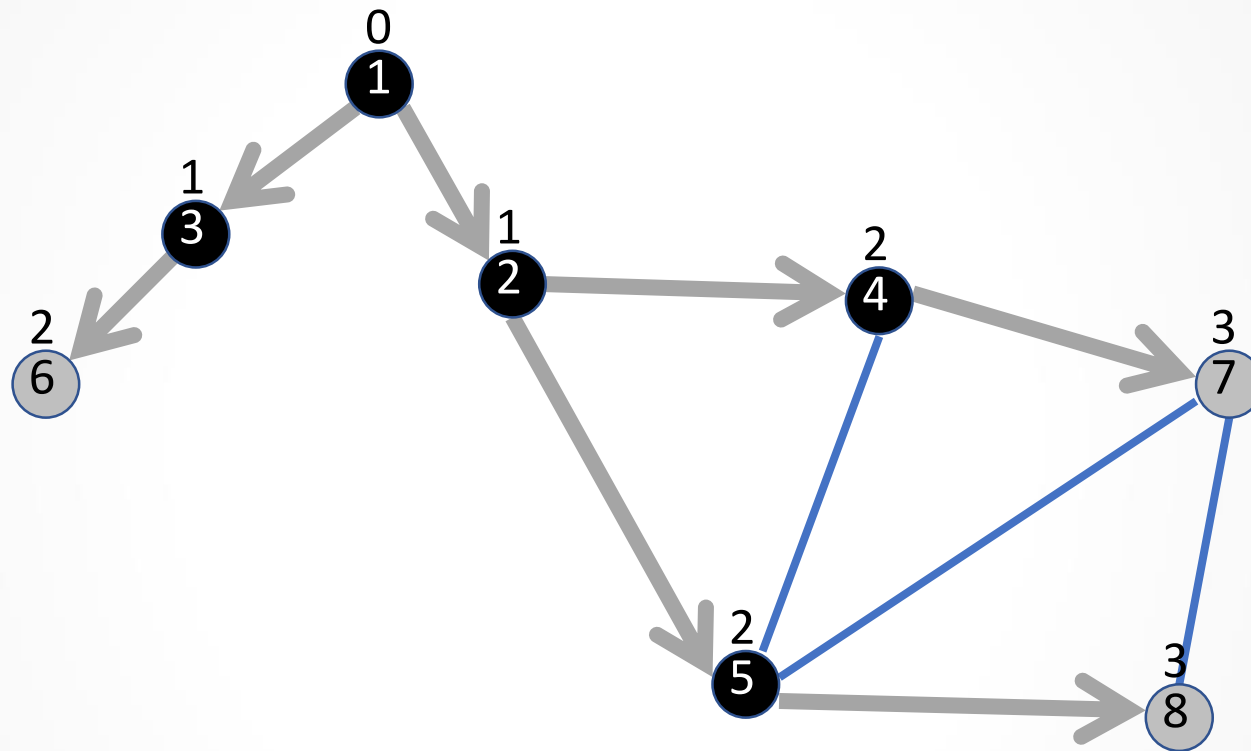
# Обход в ширину: пример



Q: 6,7,8

Просматриваемая дуга: (5,8)

# Обход в ширину: пример

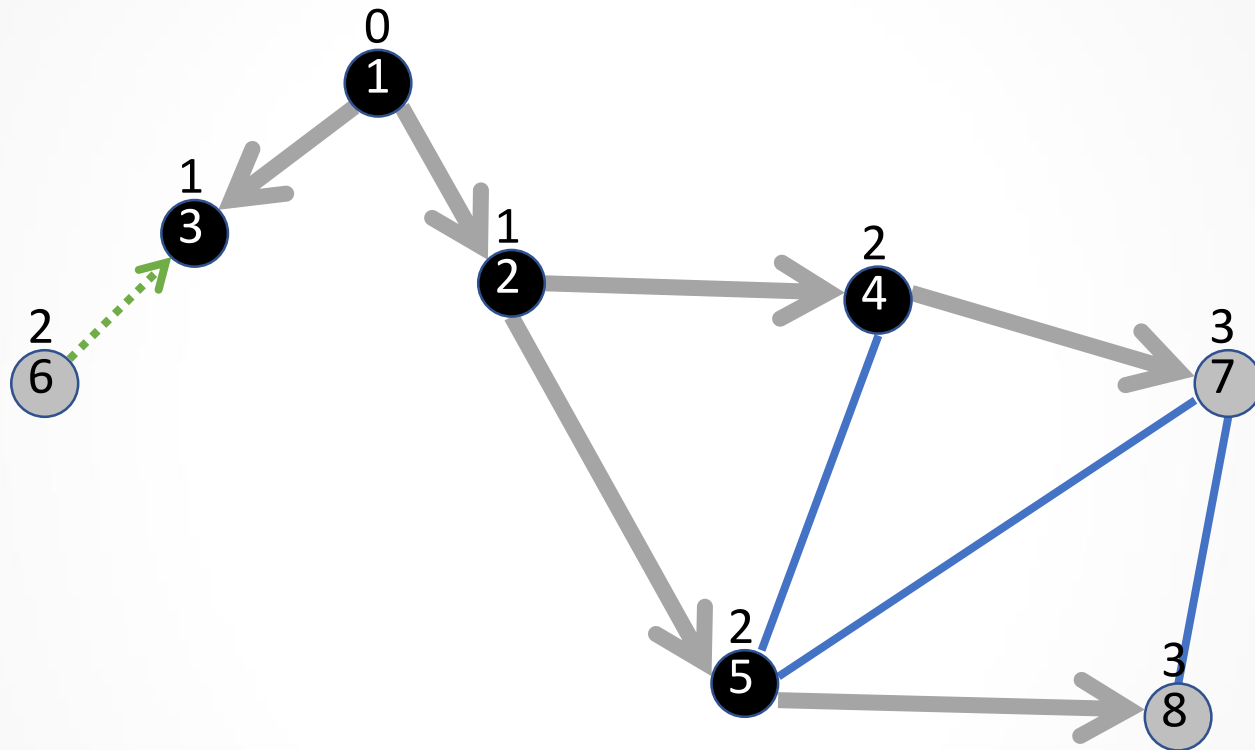


Q: 6,7,8

Просматриваемая дуга:  $\emptyset$

Вершина 5 становится *обработанной*

# Обход в ширину: пример

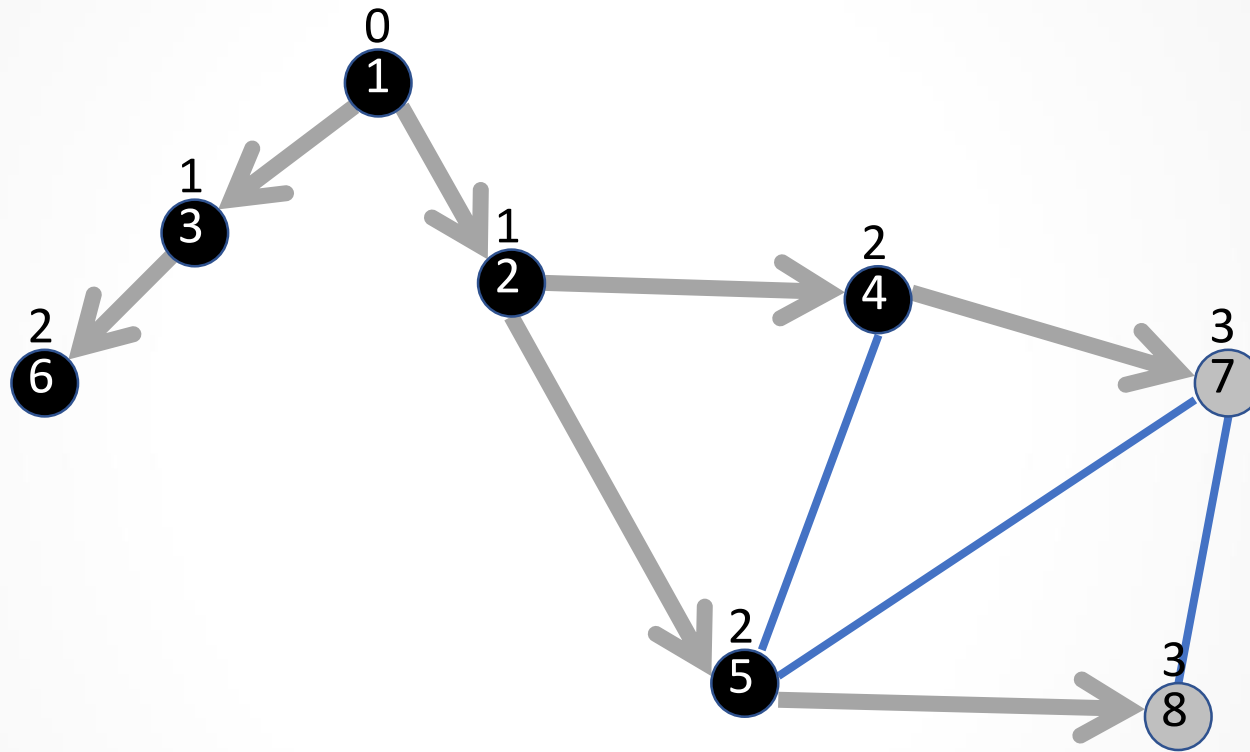


Q: 7,8

Просматриваемая дуга: (6,3)



# Обход в ширину: пример

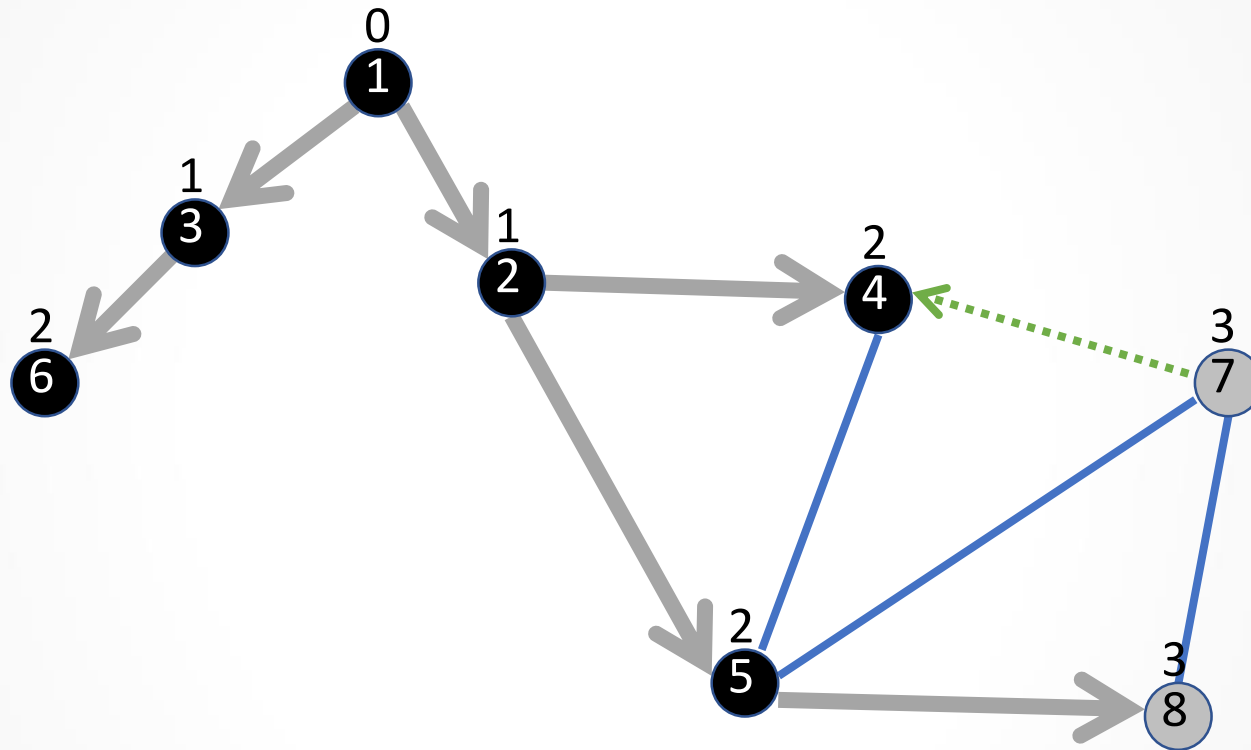


Q: 7,8

Просматриваемая дуга:  $\emptyset$

Вершина 6 становится *обработанной*

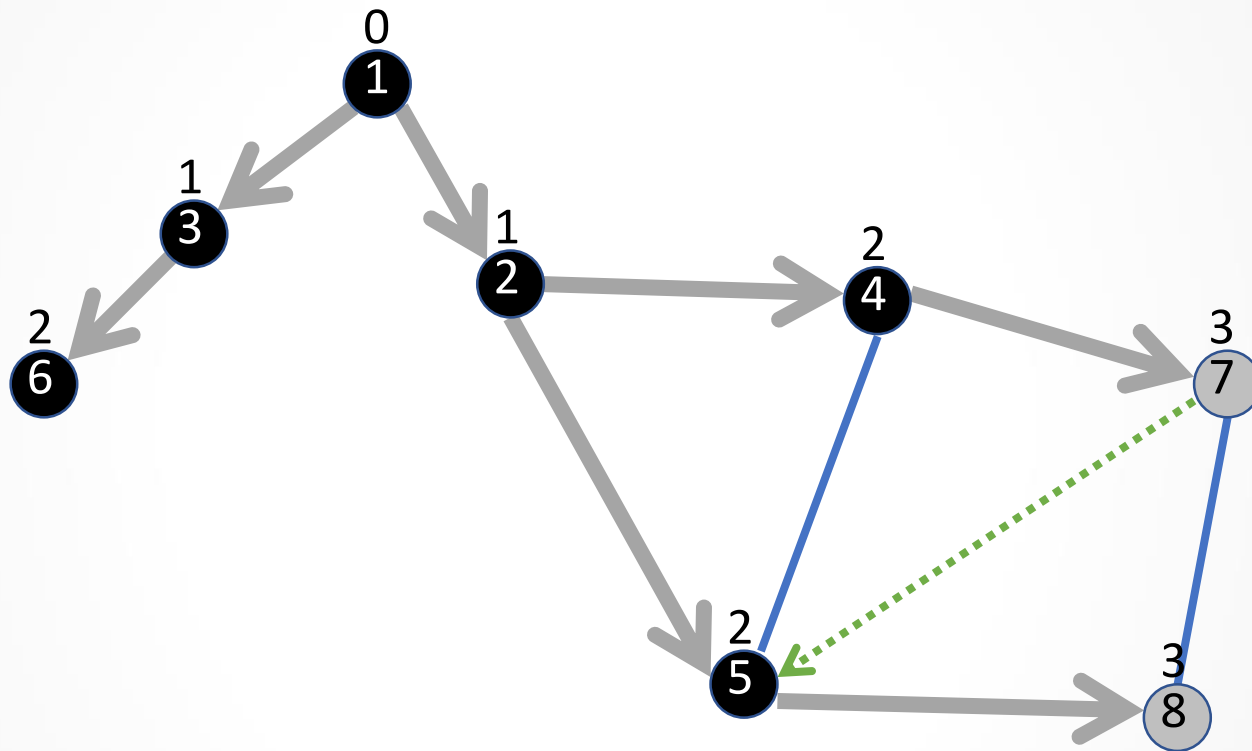
# Обход в ширину: пример



Q: 7,8

Просматриваемая дуга: (7,4)

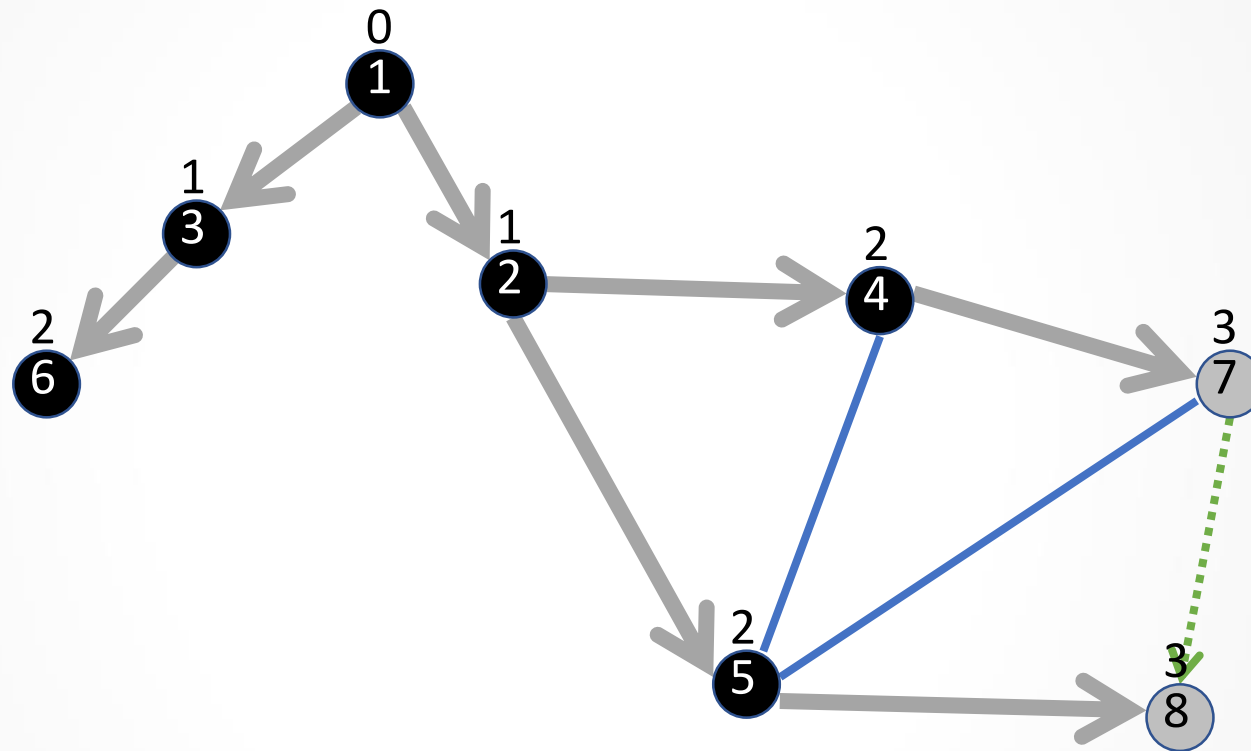
# Обход в ширину: пример



Q: 8

Просматриваемая дуга: (7,5)

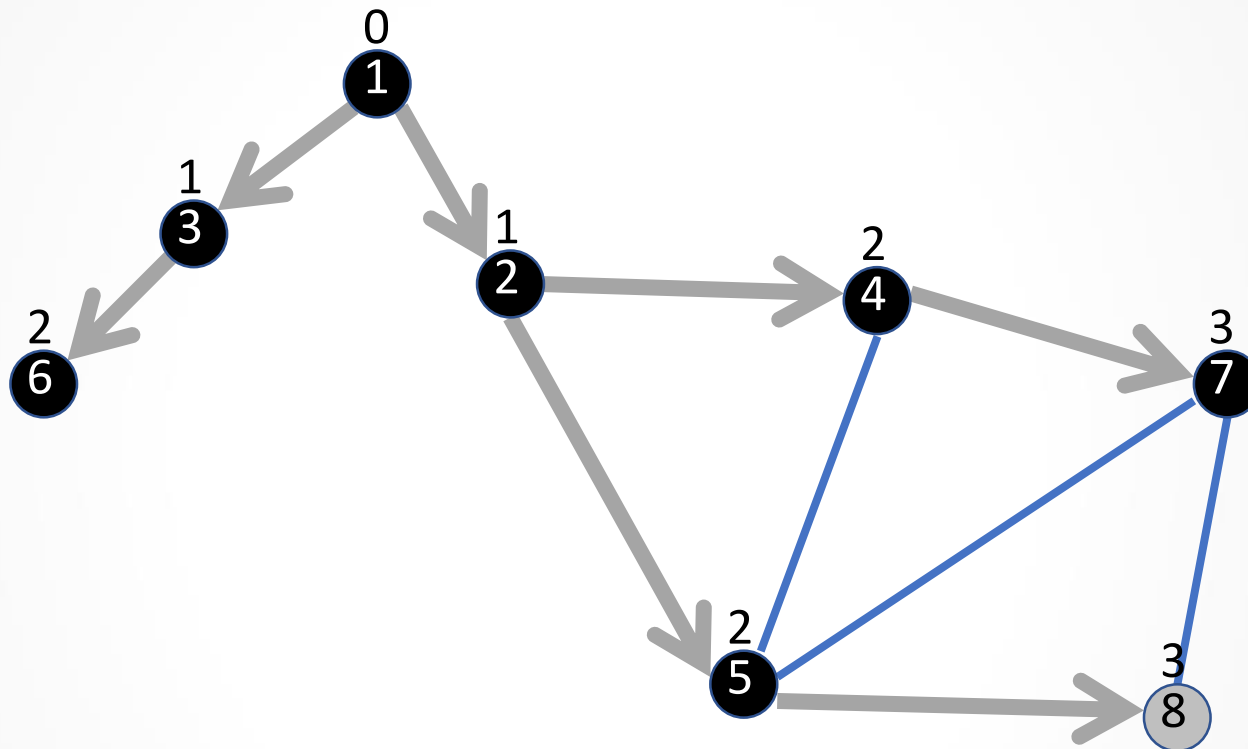
# Обход в ширину: пример



Q: 8

Просматриваемая дуга: (7,8)

# Обход в ширину: пример

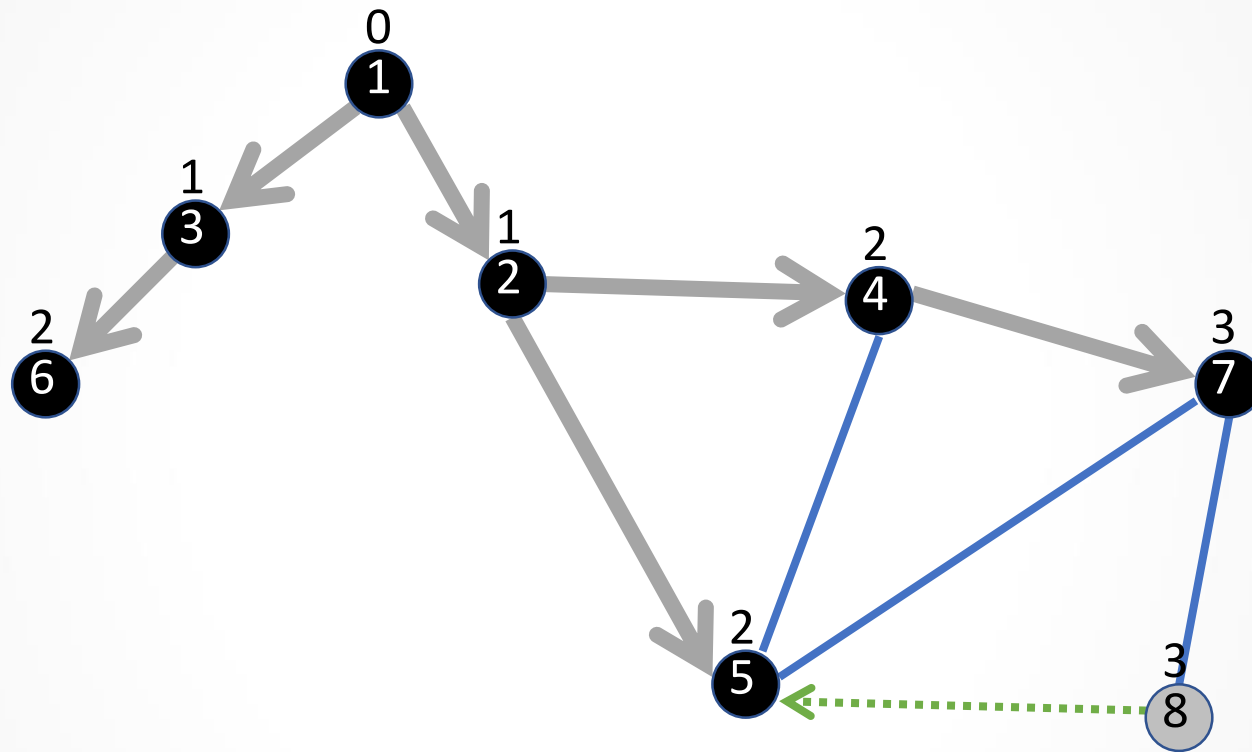


Q: 8

Просматриваемая дуга:  $\emptyset$

Вершина 7 становится *обработанной*

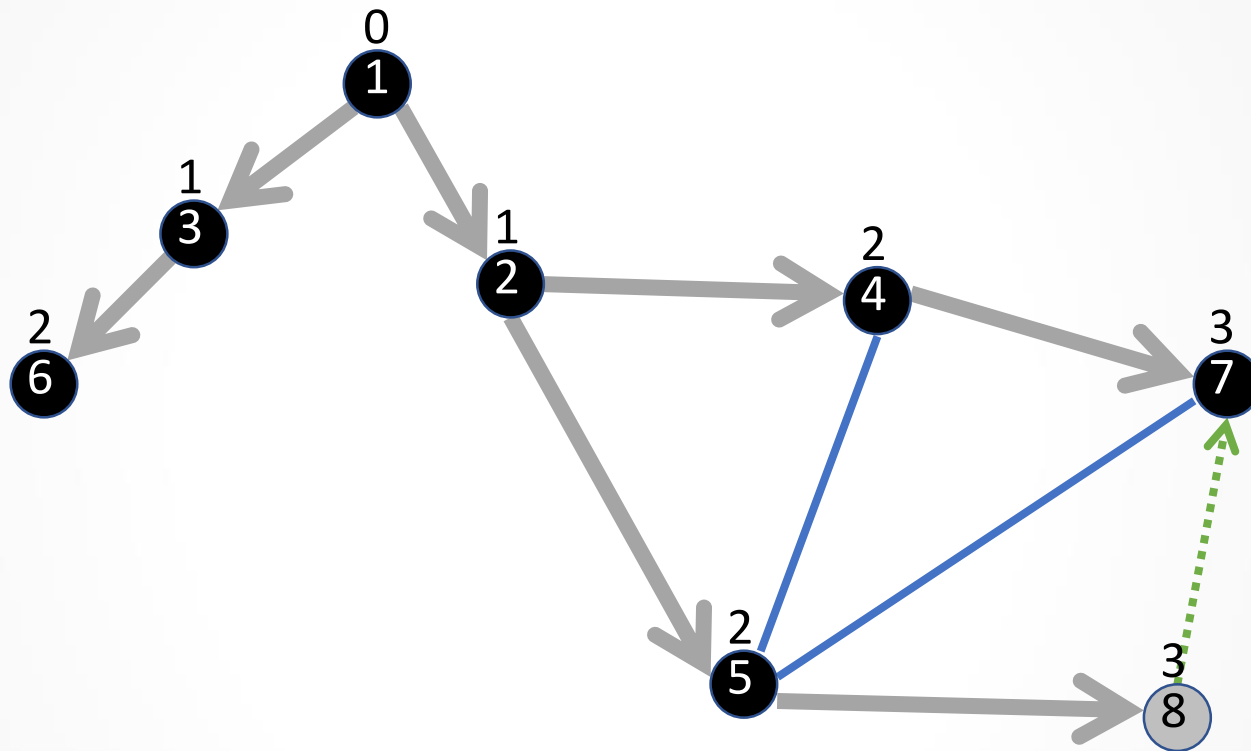
# Обход в ширину: пример



Q:

Просматриваемая дуга: (8,5)

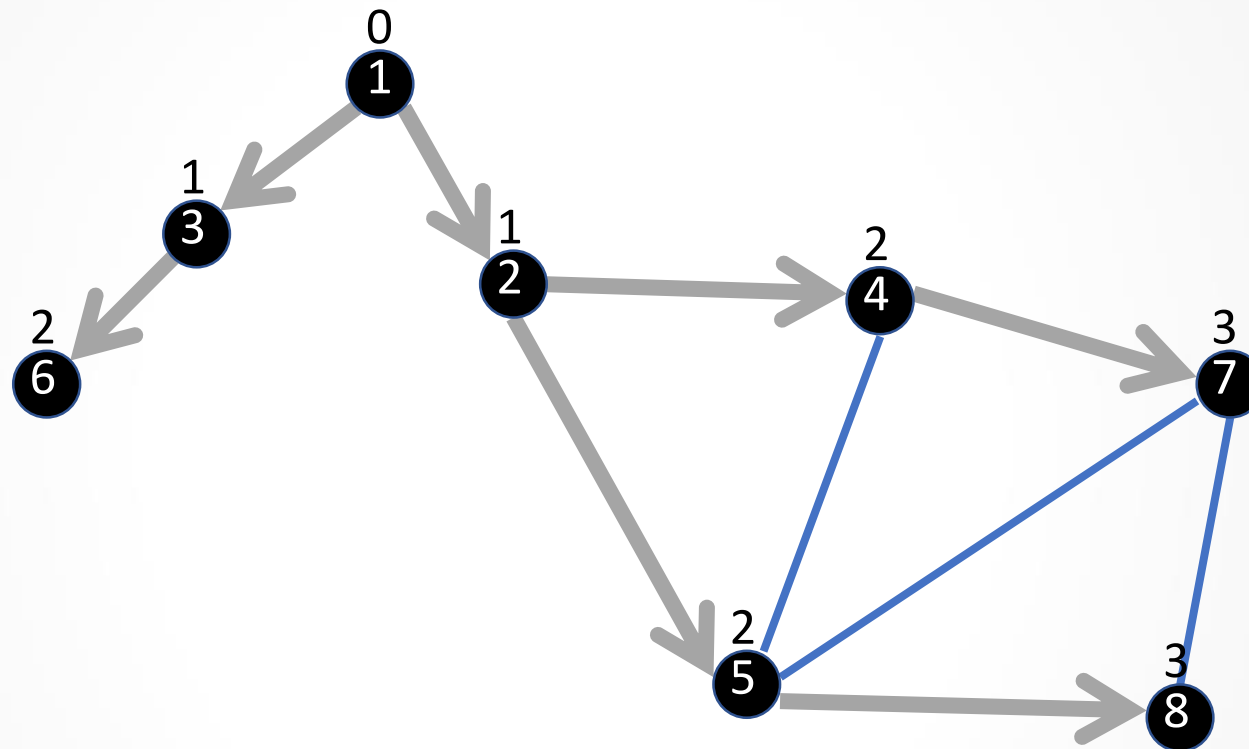
# Обход в ширину: пример



Q:

Просматриваемая дуга: (8,7)

# Обход в ширину: пример



Q:

Просматриваемая дуга:  $\emptyset$

Вершина 8 становится *обработанной*



# Алгоритм обхода в ширину

- Для обхода графа в ширину нужны две базовых структуры данных:
  - ✓ Для учёта статуса вершины – массив `State[]`.
  - ✓ Для учёта порядка вершин, которые уже отобрали для посещения, но ещё не посещены – очередь `Queue`.
- Для сохранения дополнительной информации, необходимой для решения некоторых задач: сохранение предков вершин в массиве `Pred[]`.

# Алгоритм обхода в ширину

BFS (G)

For each  $v \in V$ :

    State[v] := 'unvisited';

    Pred[v] := NULL;

Queue.Empty();

For each  $v \in V$ :

    If State[v] = 'unvisited'

        BFS\_Visit(v);

# Алгоритм обхода в ширину

BFS Visit (s)

Queue.Enqueue (s) ;

While (!Queue.IsEmpty ())

    v = Queue.Dequeue () ;

    For each u in Adj (v)

        If State[u] = 'unvisited'

            State[u] := 'visited' ;

            Pred[u] := v ;

            Queue.Enqueue (u) ;

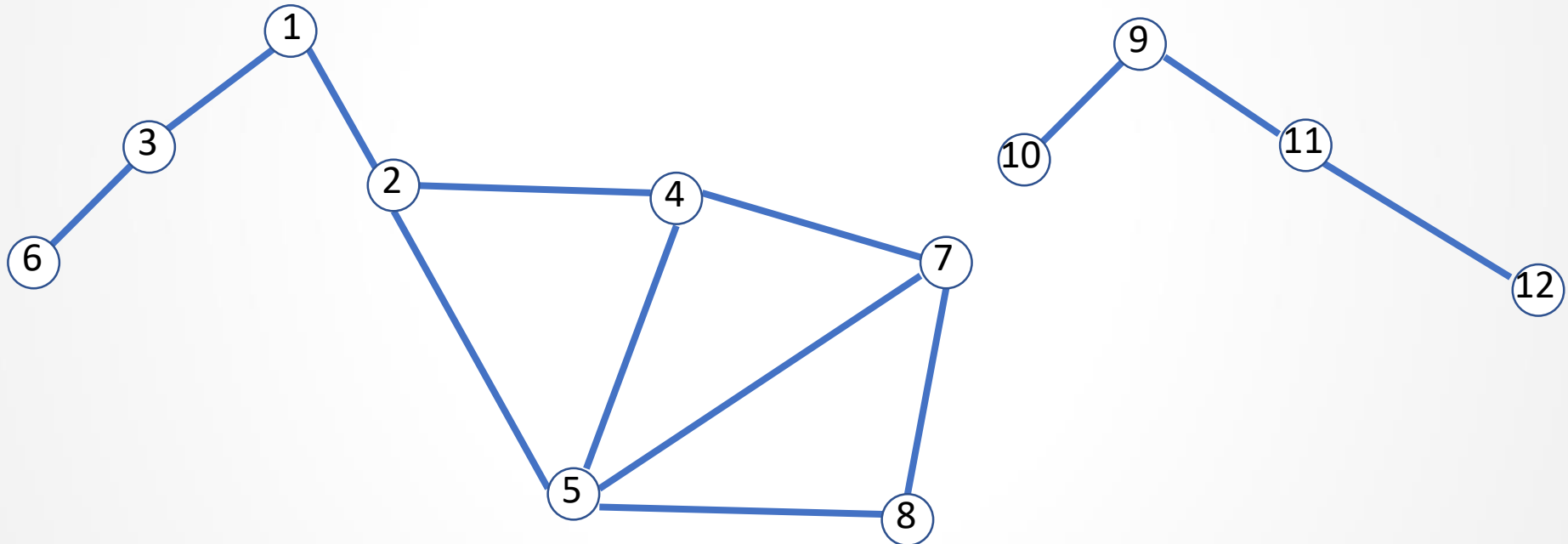
    State[v] := 'processed' ;

# Применение обхода в ширину

- ✓ Поиск компонент связности
- ✓ Поиск кратчайших путей на невзвешенных графах
- ✓ Обнаружение циклов

# Поиск компонент связности

Идея алгоритма: все вершины, которые алгоритм проходит в рамках одного вызова `BFS_Visit`, относятся к одной компоненте связности.



Если граф *связен*, то в рамках одного вызова мы обойдём все вершины графа.

# ПОИСК КОМПОНЕНТ СВЯЗНОСТИ

BFS (G)

For each  $v \in V$ :

    State[v] := 'unvisited';

    Pred[v] := NULL;

    Component[v] := 0;

Queue.Empty();

CurComp = 0;

For each  $v \in V$ :

    If State[v] = 'unvisited'

        CurComp ++;

        BFS\_Visit(v, CurComp);

# ПОИСК КОМПОНЕНТ СВЯЗНОСТИ

```
BFS Visit (s, CurComp)  
Queue.Enqueue (s) ;  
While (!Queue.IsEmpty ())  
    v = Queue.Dequeue () ;  
    For each u in Adj (v)  
        If State[u] = 'unvisited'  
            State[u] := 'visited' ;  
            Pred[u] := v ;  
            Queue.Enqueue (u) ;  
    State[v] := 'processed' ;  
    Component[v] := CurComp ;
```

# Обнаружение циклов

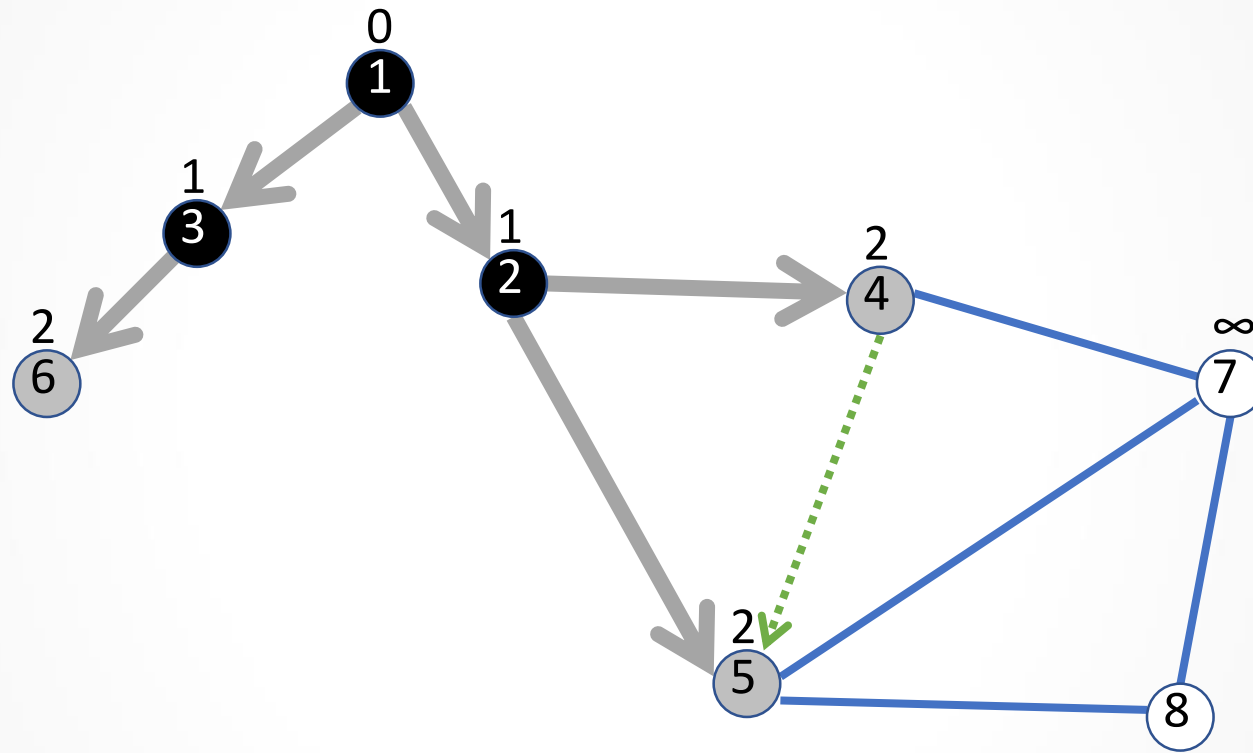
Идея алгоритма: если при анализе вершин, смежных с текущей вершиной

For each  $u$  in  $Adj(v)$

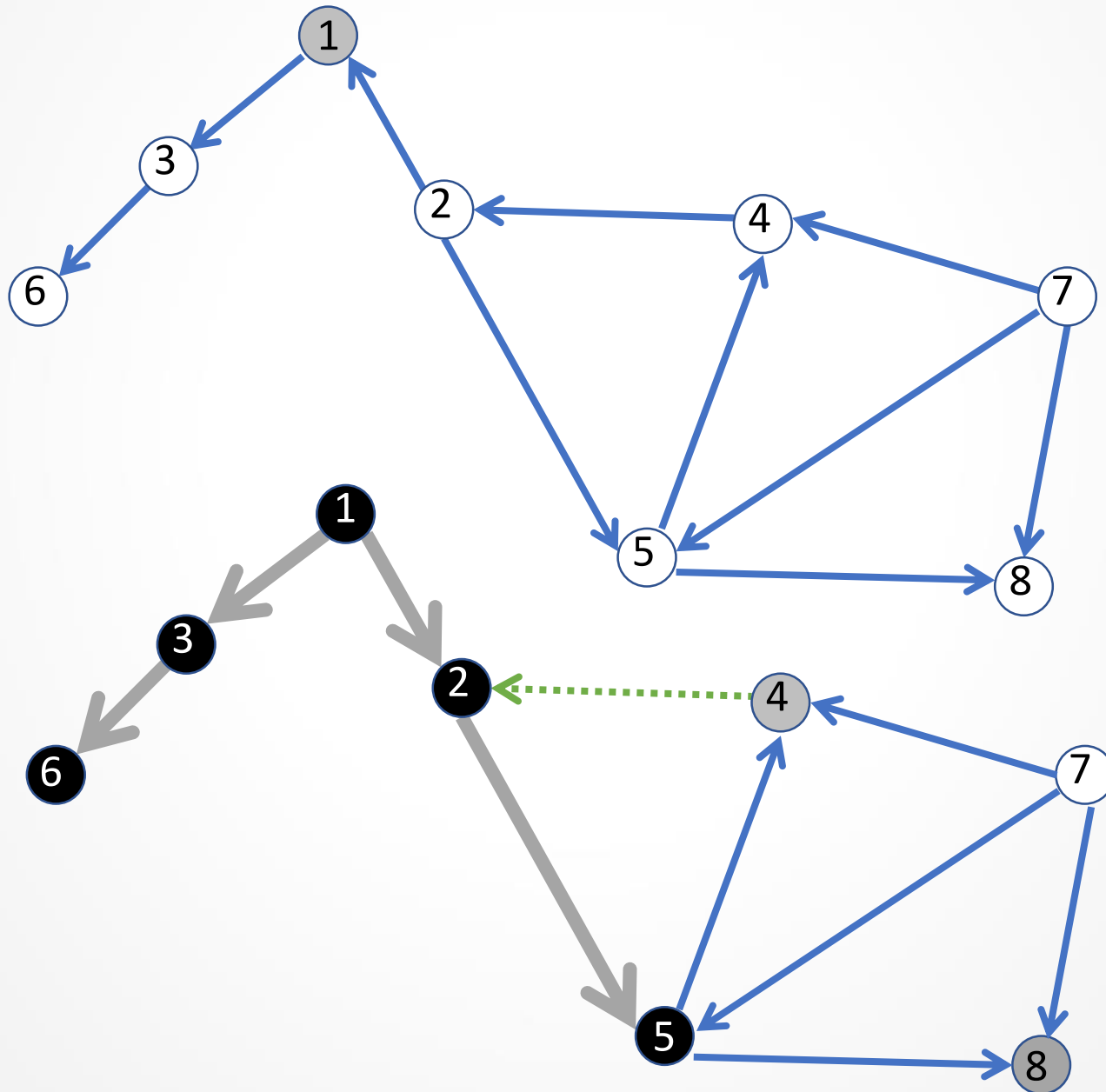
Мы обнаруживаем, что смежная вершина ( $u$ ) уже посещена или обработана и, в случае неориентированных графов, не является непосредственным предком, то мы обнаружили цикл.



# Обход в ширину: пример



# Обход в ширину: пример



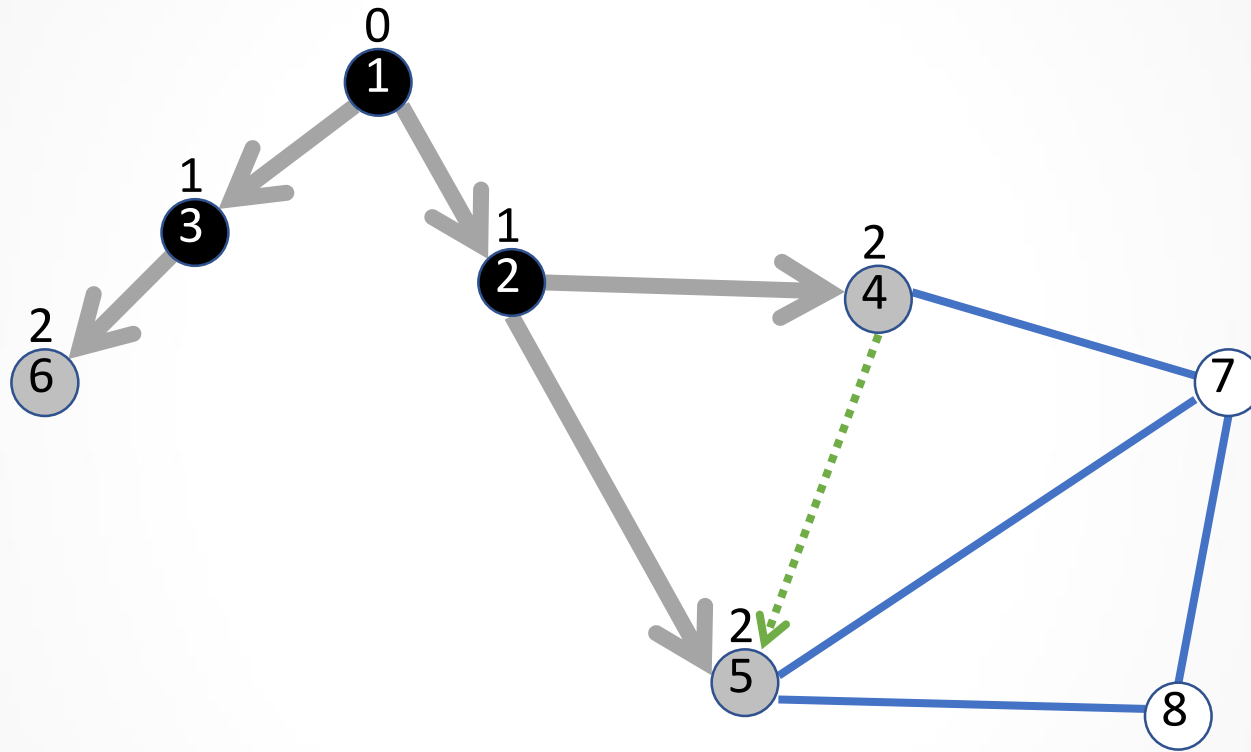
# Обнаружение циклов

Описанная идея позволяет обнаружить факт наличия цикла.  
Как найти сам цикл?

Идея алгоритма определения самого цикла в виде последовательности рёбер:

- Для каждой вершины ведём список всех предшественников (через `Pred[]`).
- При обнаружении цикла – сравниваем списки предшественников для вершин-концов просматриваемого ребра:
  - Находим ближайшего (к концам списков, т.е. к текущим вершинам) общего предшественника.
  - Цикл включает объединение трёх путей: двух путей от ближайшего общего предшественника до двух текущих вершин и текущего просматриваемого ребра.

# Обнаружение циклов



`PredList[4] = [2]`

`PredList[5] = [2]`

`Цикл = {(4, 2), (2, 5), (5, 4)}`

# Поиск кратчайших путей

Обход в ширину можно использовать для расчёта расстояний и поиска кратчайших путей на невзвешенном графе.

Кратчайший путь из вершины  $u$  в вершину  $v$  – путь (цепь), состоящий из наименьшего количества рёбер.

Идея алгоритма: пусть  $s$  – вершина, с которой начался обход при вызове `BFS_Visit`. Алгоритм `BFS_Visit` обходит граф  $G$  «волнами», каждая волна формируется в рамках одного цикла

```
For each  $u$  in  $Adj(v)$ 
```

Номер волны = расстоянию от  $s$  до заданной вершины.  
Сами пути можно восстановить по `Pred`.

# Поиск кратчайших путей

BFS Visit(s)

For each  $v \in V \setminus \{s\}$  :  $\text{Dist}[v] := +\infty$ ;

$\text{Dist}[s] := 0$ ;

Queue.Enqueue(s);

While (!Queue.IsEmpty())

$v = \text{Queue.Dequeue}()$ ;

    For each  $u$  in Adj( $v$ )

        If State[ $u$ ] = 'unvisited'

            State[ $u$ ] := 'visited';

            Pred[ $u$ ] :=  $v$ ;

            Dist[ $u$ ] := Dist[ $v$ ]+1;

            Queue.Enqueue( $u$ );

    State[ $v$ ] := 'processed';

# Поиск кратчайших путей

