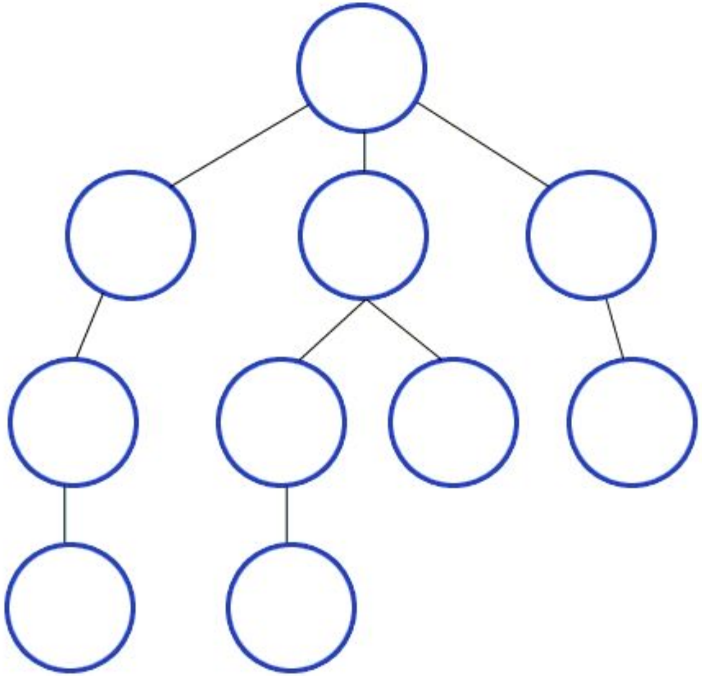
RUCODE festival

Дивизион D. День 4
Графы (BFS и Алгоритм Дейкстры)

Лектор: Дмитрий Руденко

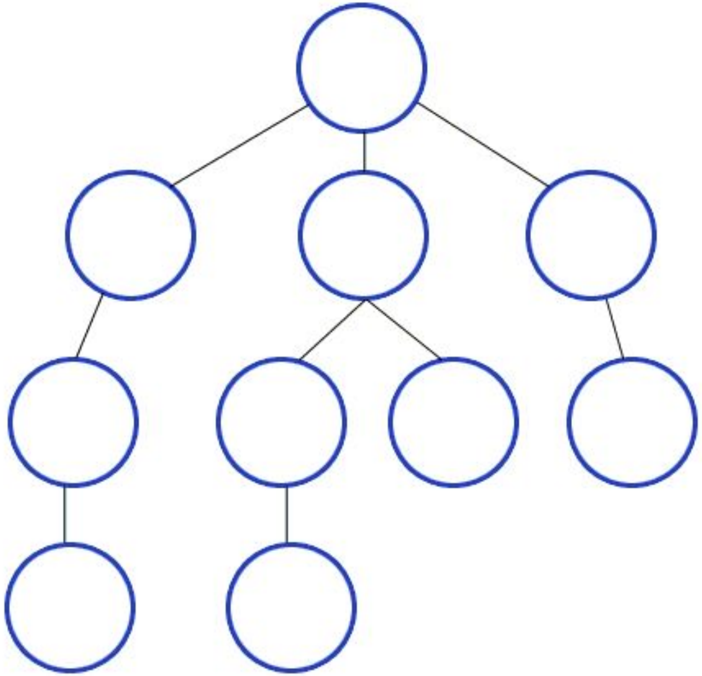
Поиск в ширину (Базовый)



Поиск в ширину (Базовый)

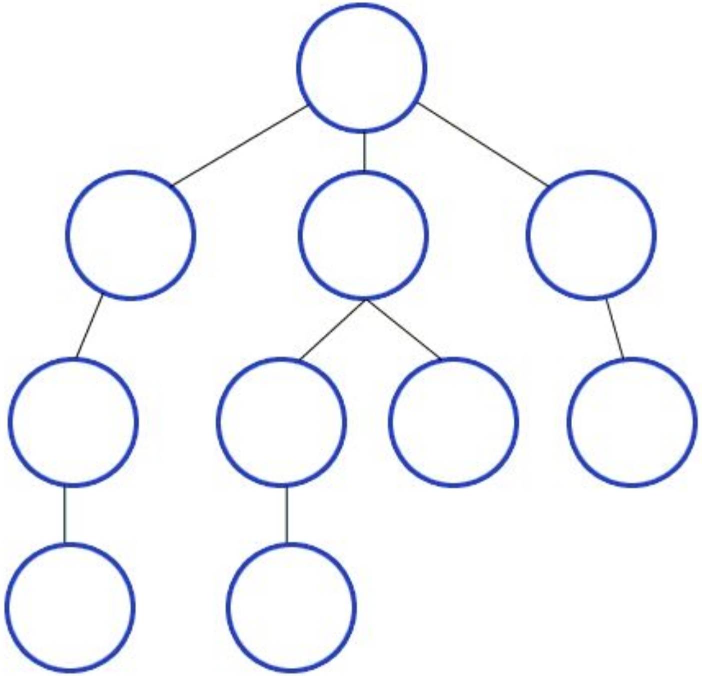
Реализация

1) Добавим первую вершину в очередь
2) Удаляем первую вершину очереди и добавляем смежные с ней вершины (еще не посещенные)
3) Так делаем для всех вершин, которые были в очереди в начале шага
4) Если очередь не пустая - переходим к пункту 2



Поиск в ширину (Базовый)

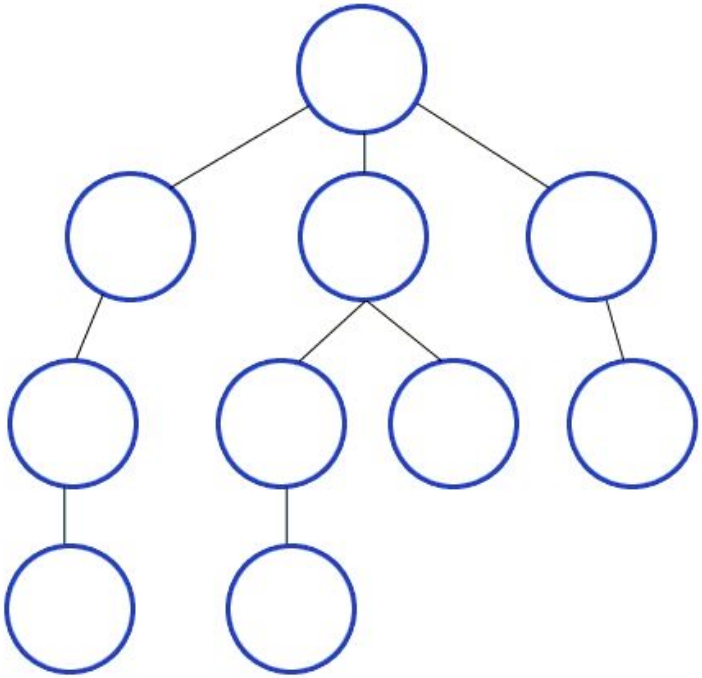
Реализация



Поиск в ширину (Базовый)

Реализация

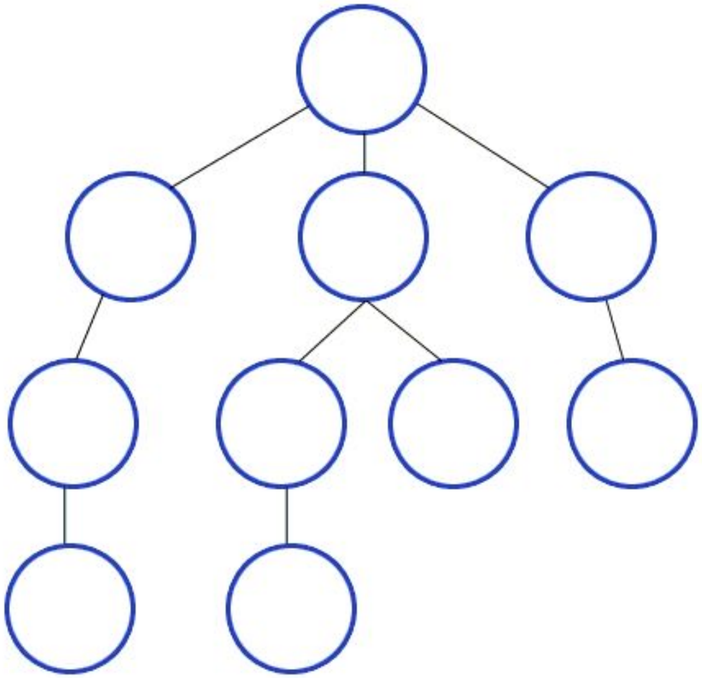
2	3	4		



Поиск в ширину (Базовый)

Реализация

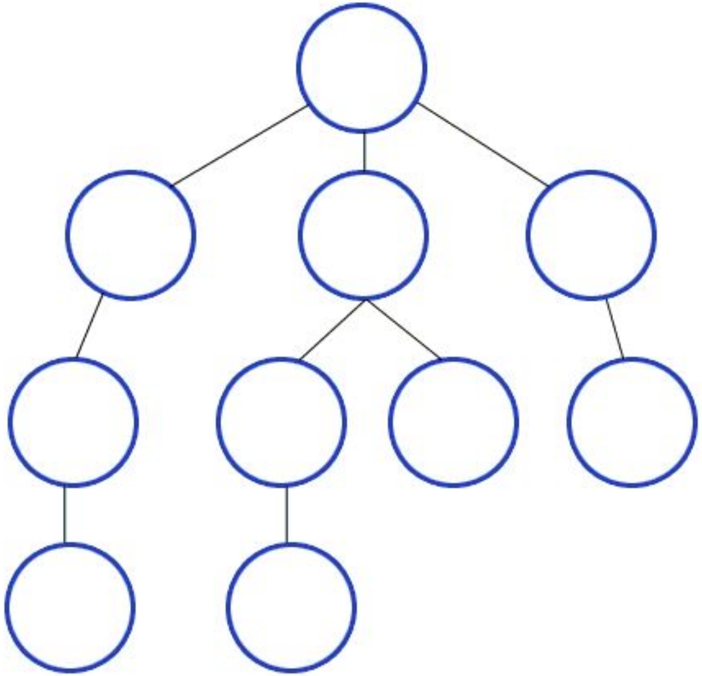
2	3	4	5	



Поиск в ширину (Базовый)

Реализация

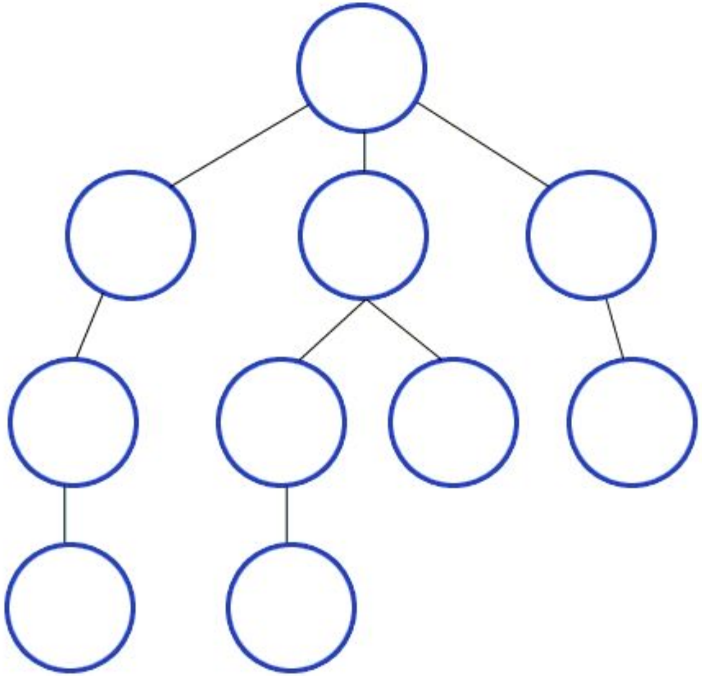
3	4	5		



Поиск в ширину (Базовый)

Реализация

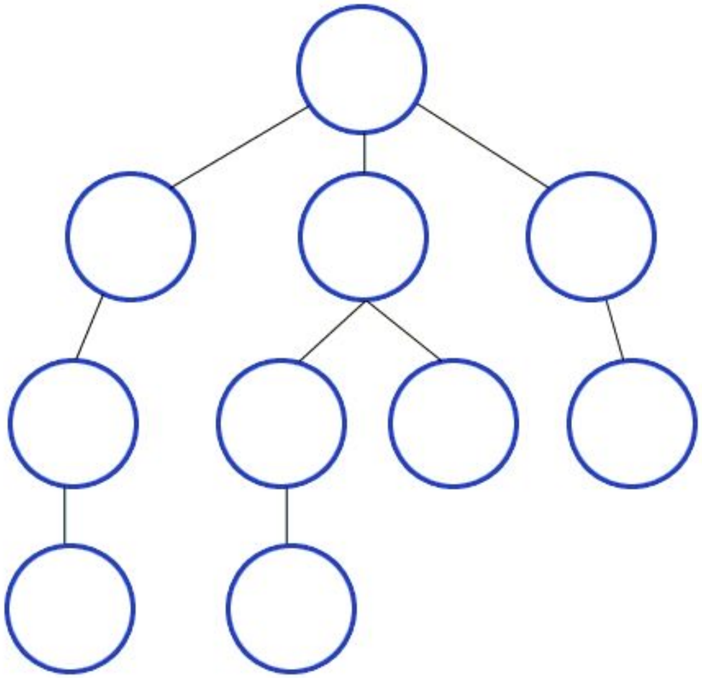
3	4	5	6	7



Поиск в ширину (Базовый)

Реализация

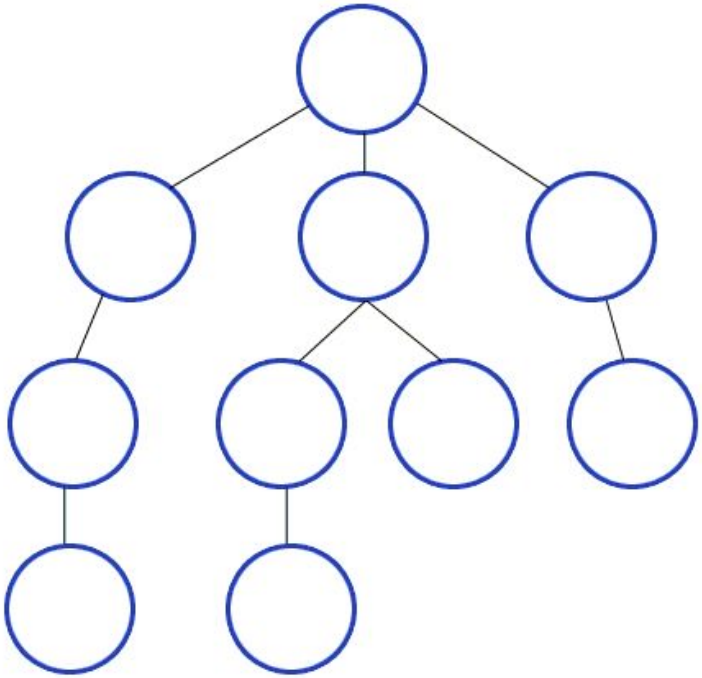
4	5	6	7	



Поиск в ширину (Базовый)

Реализация

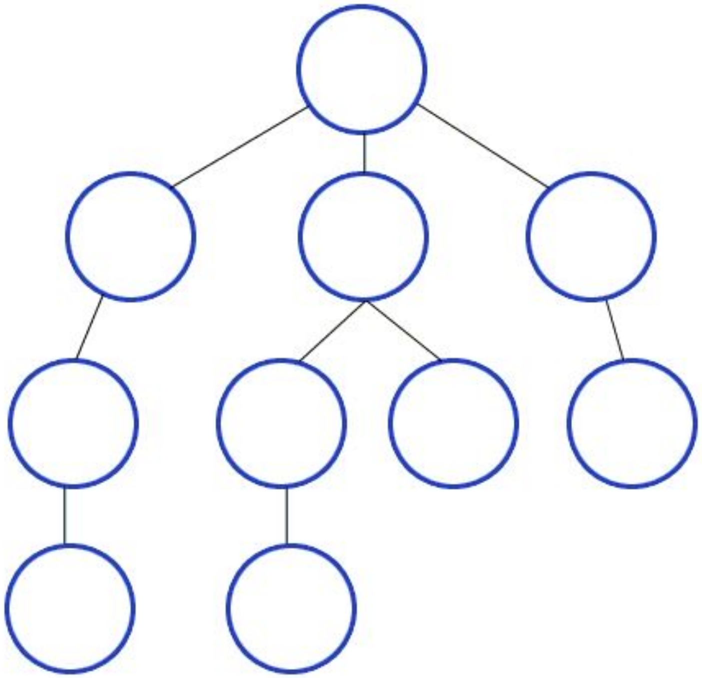
4	5	6	7	8



Поиск в ширину (Базовый)

Реализация

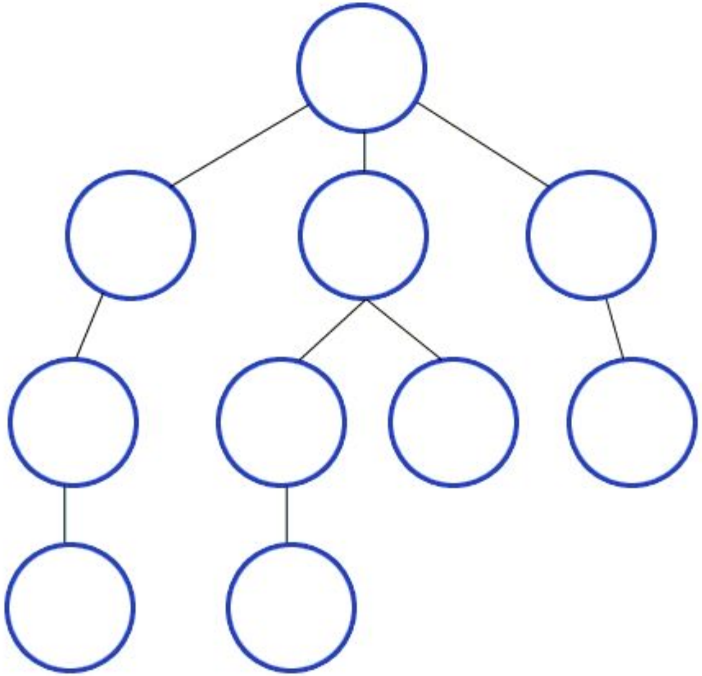
5	6	7	8	



Поиск в ширину (Базовый)

Реализация

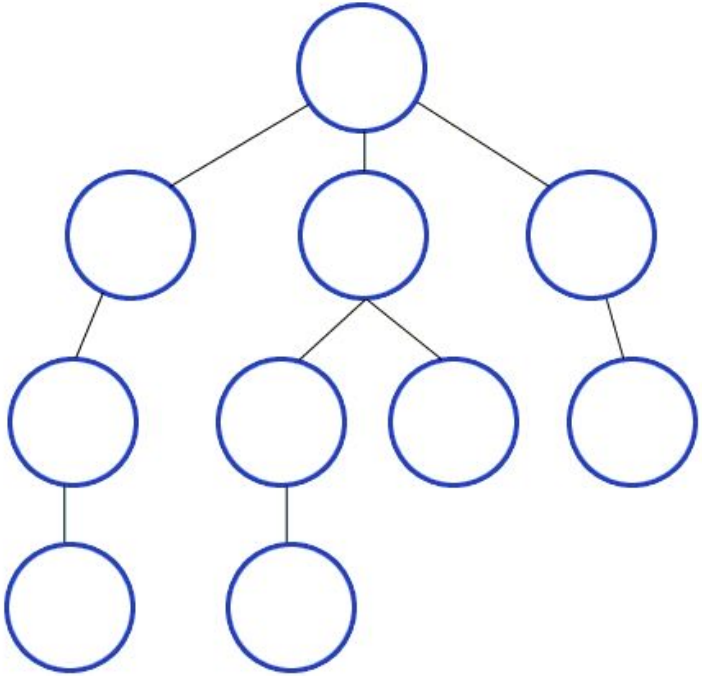
5	6	7	8	



Поиск в ширину (Базовый)

Реализация

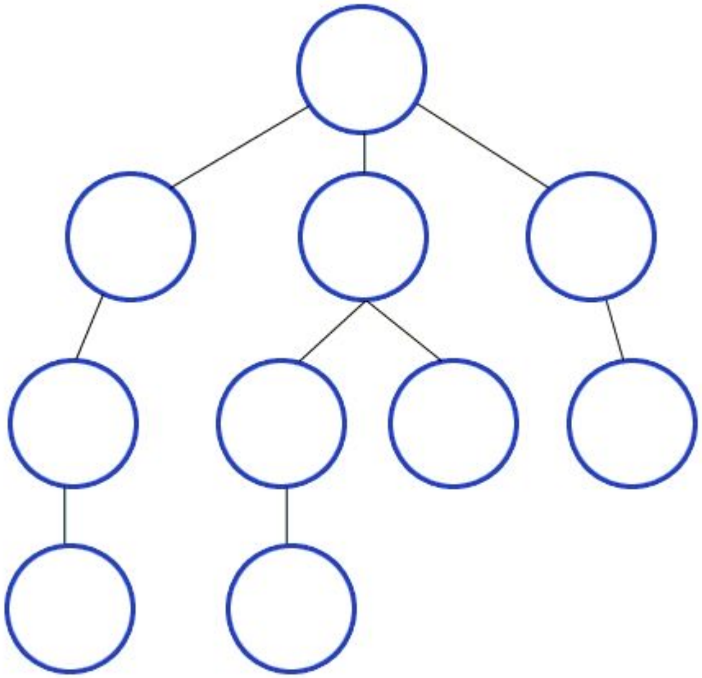
5	6	7	8	9



Поиск в ширину (Базовый)

Реализация

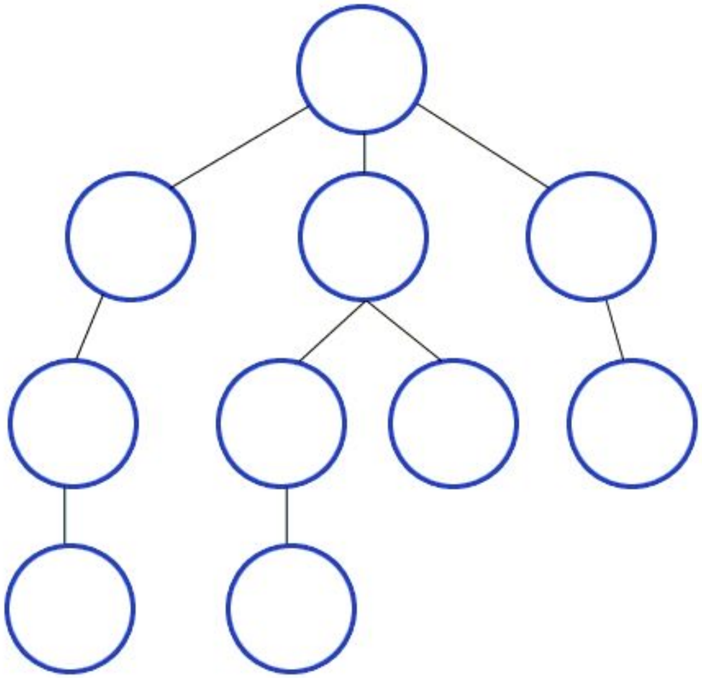
6	7	8	9	



Поиск в ширину (Базовый)

Реализация

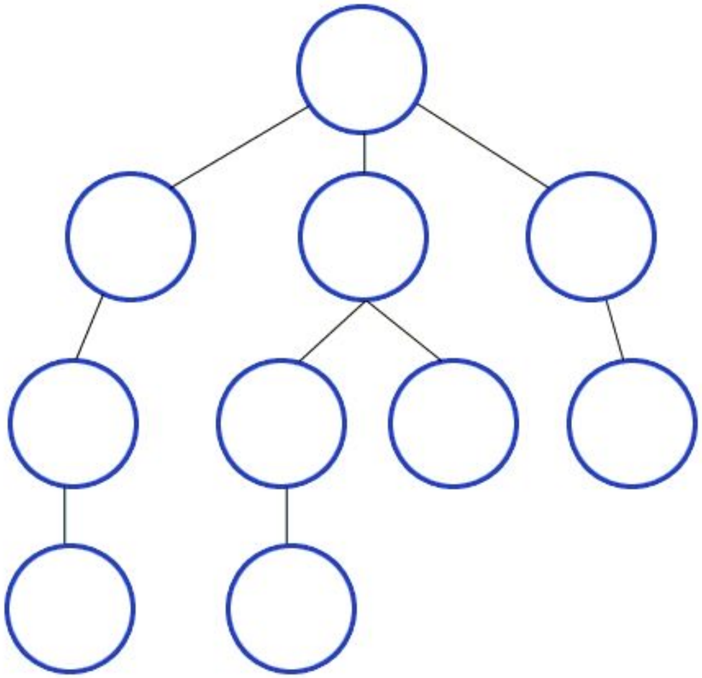
6	7	8	9	10



Поиск в ширину (Базовый)

Реализация

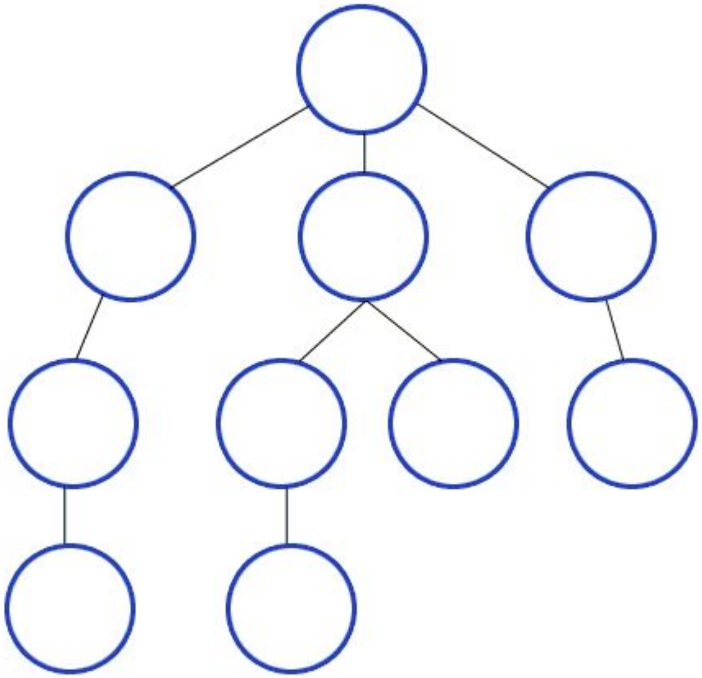
7	8	9	10	



Поиск в ширину (Базовый)

Реализация

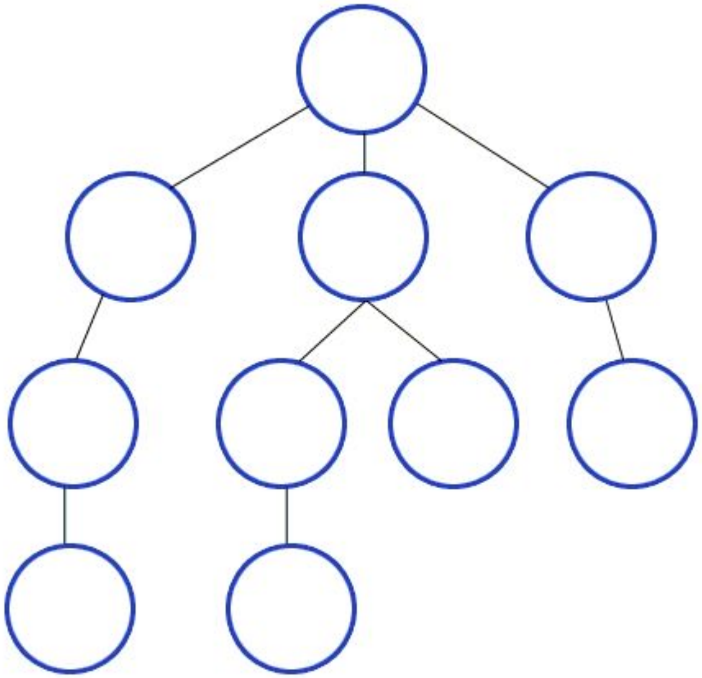
8	9	10		



Поиск в ширину (Базовый)

Реализация

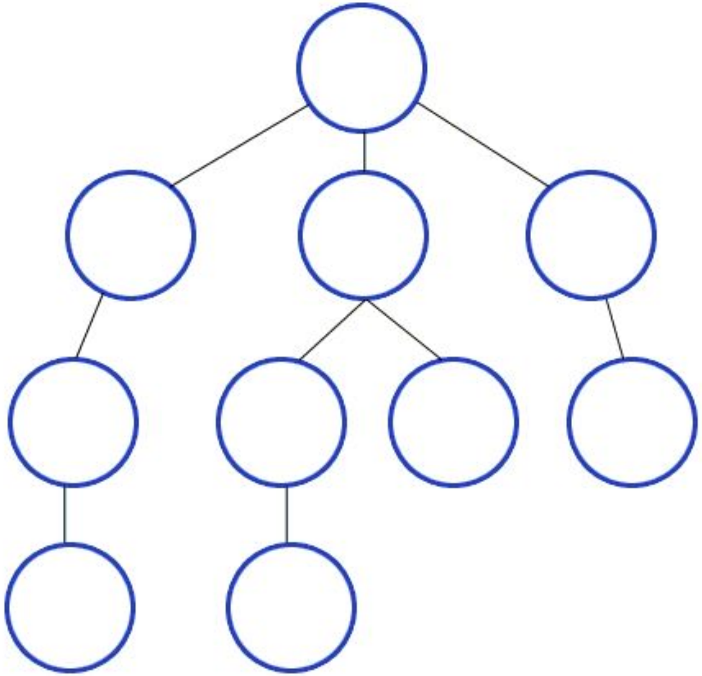
9	10			



Поиск в ширину (Базовый)

Реализация

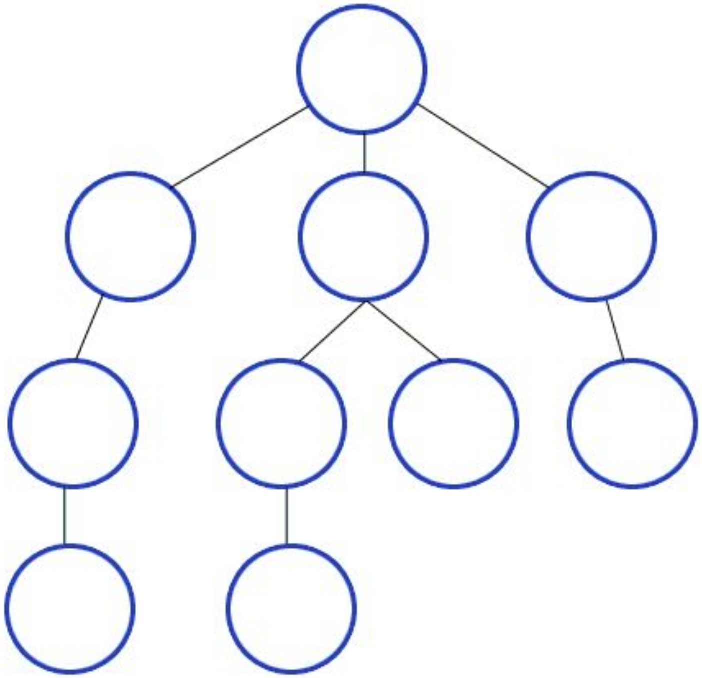
9	10			



Поиск в ширину (Базовый)

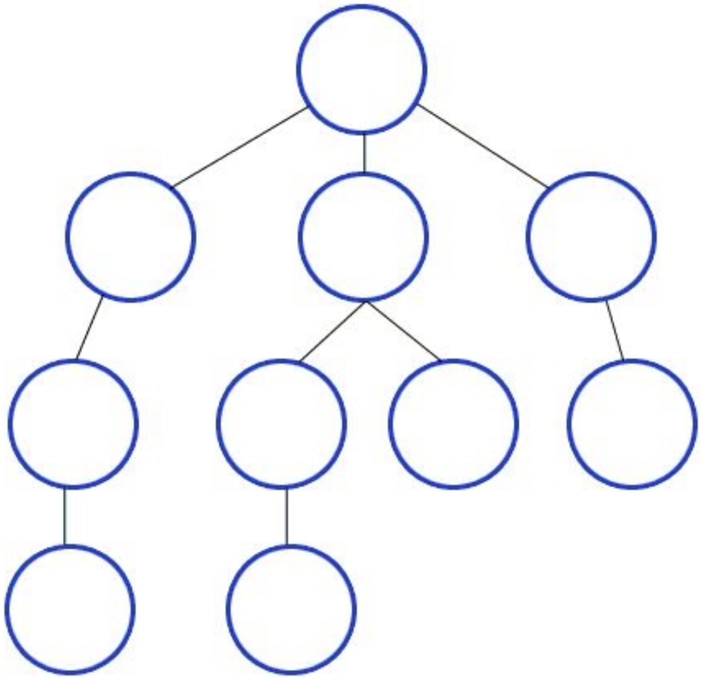
Реализация

10				



Поиск в ширину (Базовый)

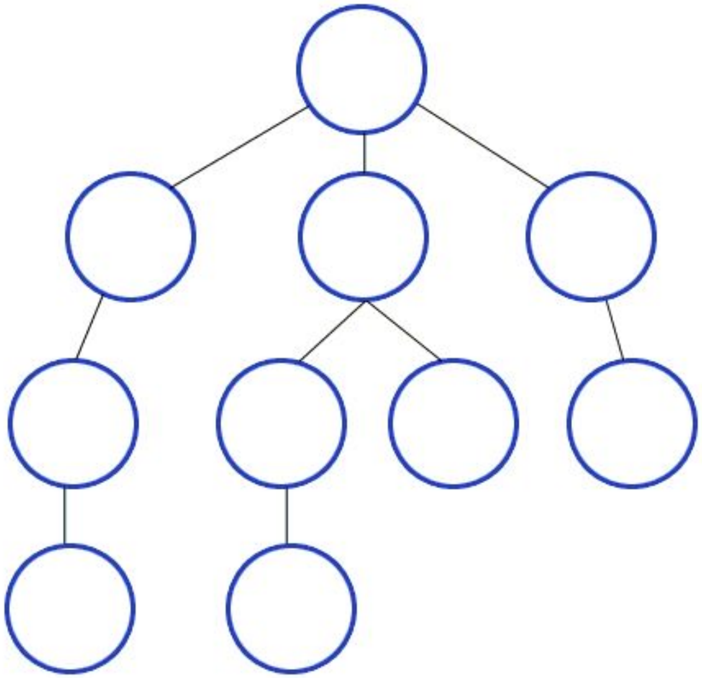
Реализация



Поиск в ширину (Базовый)

Реализация

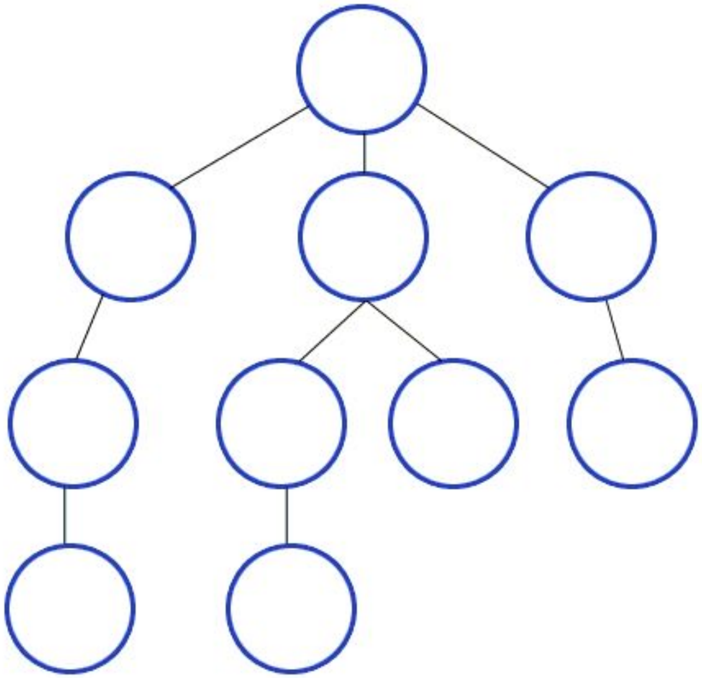
```
vector<int> graph[MAXN];
bool used[MAXN];
// graph[3] = { 6, 7 }
// graph[1] = { 2, 3, 4 }
int n, m;
cin >> n >> m;
for (int i = 0; i < m; i++) {
    int from, to;
    cin >> from >> to;
    graph[from].push_back(to);
    graph[to].push_back(from);
    // only graph[from].push_back(to); if oriented
}
```



Поиск в ширину (Базовый)

Реализация

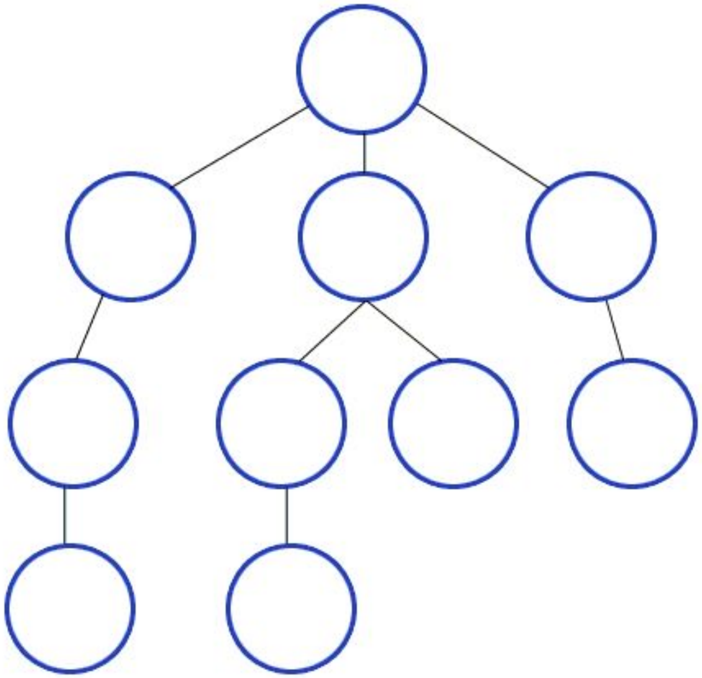
```
int distance[MAXN];
int parent[MAXN];
queue<int> q;

// s - стартовая вершина (нумерация с нуля)
q.push(s);
used[s] = true;
parent[s] = -1;
```



Поиск в ширину (Базовый)

Реализация

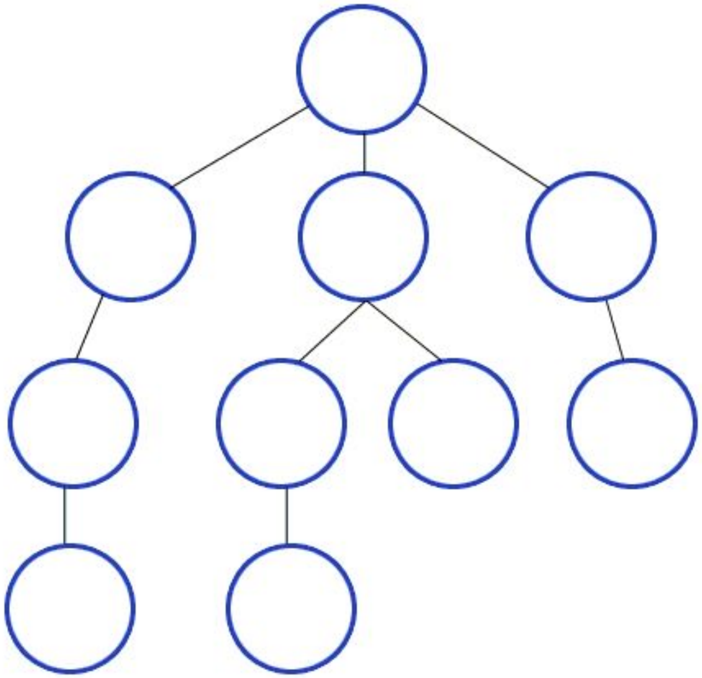
```
while (!q.empty()) {
    int v = q.front();
    q.pop();

    for (size_t i = 0; i < g[v].size(); ++i) {
        int to = g[v][i];

        if (!used[to]) {
            used[to] = true;
            q.push(to);
            distance[to] = d[v] + 1;
            parent[to] = v;
        }
    }
}
```



Поиск в ширину (Базовый)

Реализация

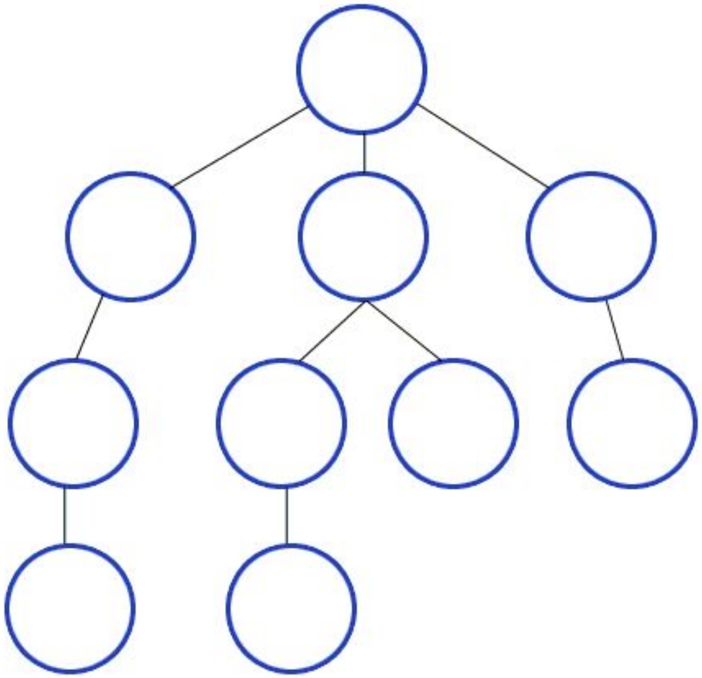
```
if (!used[to]) {
    cout << "No path!";
} else {
    vector<int> path;

    for (int v = to; v != -1; v = parent[v]) {
        path.push_back(v);
    }
    reverse (path.begin(), path.end());
    // cout << path
}
```



Поиск в ширину (Базовый)

Сложность алгоритма

1) O(V + E)
2) V - количество вершин
3) E - количество ребер



Поиск в ширину (1-k)

Условия

1) Добавляются веса к ребрам
2) Пусть вес ребра - w, 1 <= w <= k
3) Нужно найти кратчайшие расстояния
   от вершины s до всех остальных

Поиск в ширину (1-k)

Решение 1

1) Имеем путь из A -> B с весом w
2) Давайте разобьем это ребро на w ребер без веса
3) По итогу превратили граф из взвешенного в невзвешенный
4) Можем запустить базовый BFS
5) Сложность - O(V + Ek)
6) Подходит когда ребер мало

Поиск в ширину (1-k)

Решение 2

1) Имеем v вершин, значит максимальный путь будет иметь вес (v - 1) * k
2) Давайте хранить (v - 1) * k очередей, таких что в очереди d будут храниться вершины до которых оптимальный путь из s имеет вес d, или существует путь веса d, но не оптимальный
3) queue[0] = { s }

Поиск в ширину (1-k)

Решение 2

1) Идем по очередям с 0 до Vk - 1
2) Рассматриваем вершину v, которая находится в очереди d
3) Это значит что, из s в to существует путь с весом d
4) Тогда, если существует ребро между v и to (не посещенная ранее вершина) веса w, то путь от s до to в итоге будет не больше чем d + w
5) Давайте добавим вершину to в очередь d + w

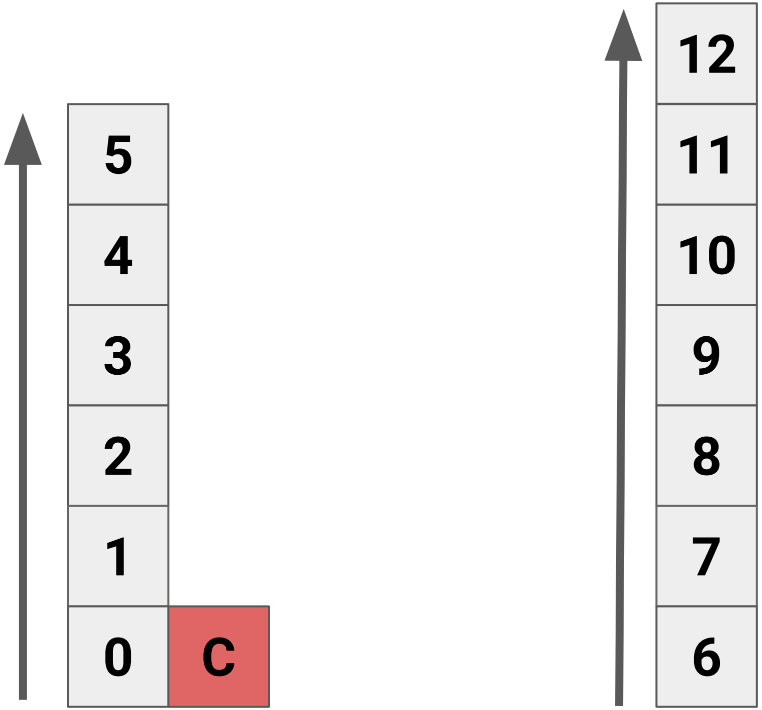
Поиск в ширину (1-k)

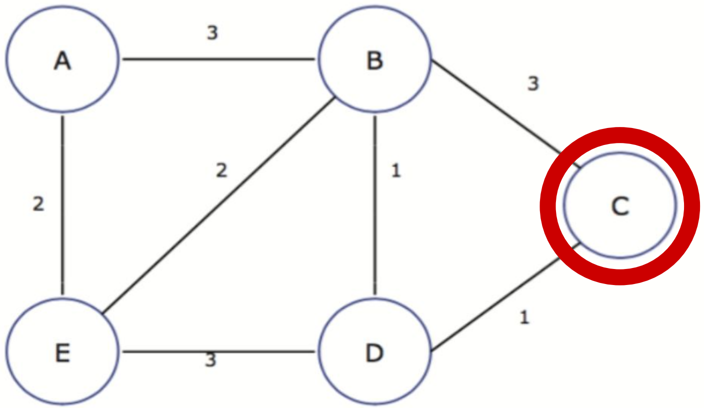
Решение 2

1) Если до уровня d расстояние для всех вершин уже посчитано оптимально, то в очереди d + 1 находятся вершины, до которых минимальное расстояние d + 1 или дубликат вершины на уровне ниже, до которой существует расстояние d + 1 и мы когда-то нашли более оптимальное

Поиск в ширину (1-k)

Решение 2

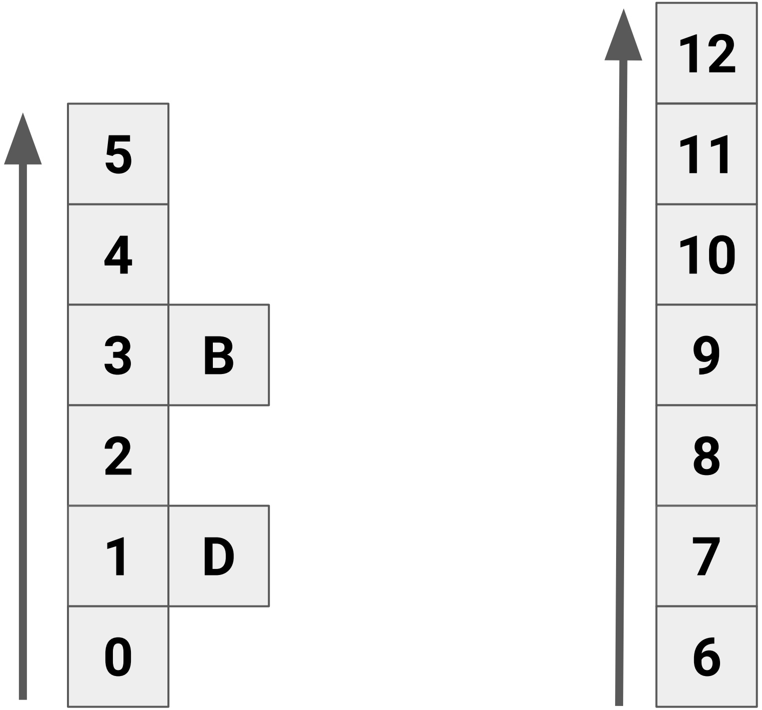


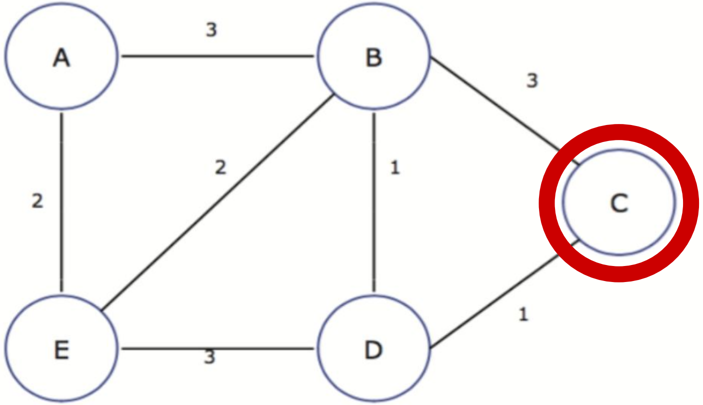


A	B	C	D	E
inf	inf	0	inf	inf

Поиск в ширину (1-k)

Решение 2

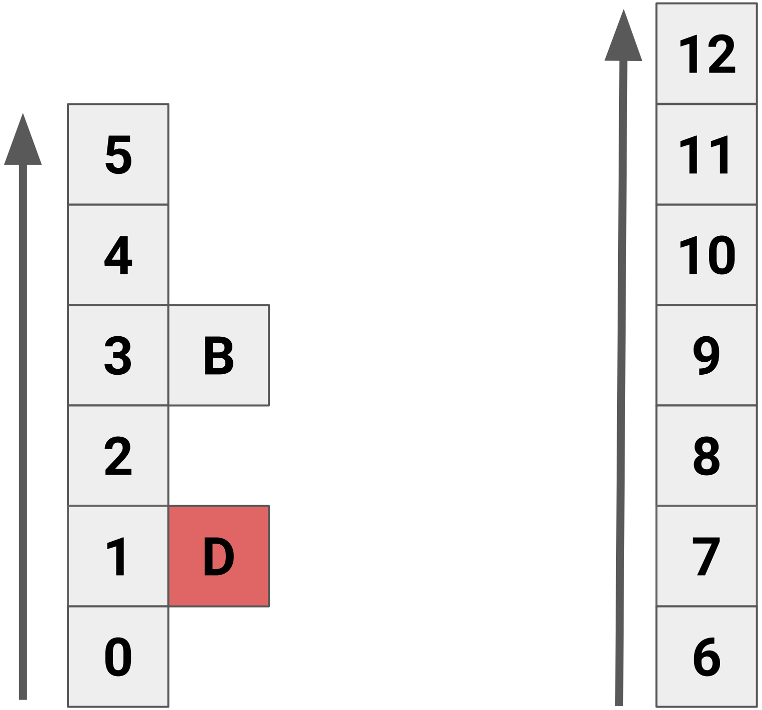


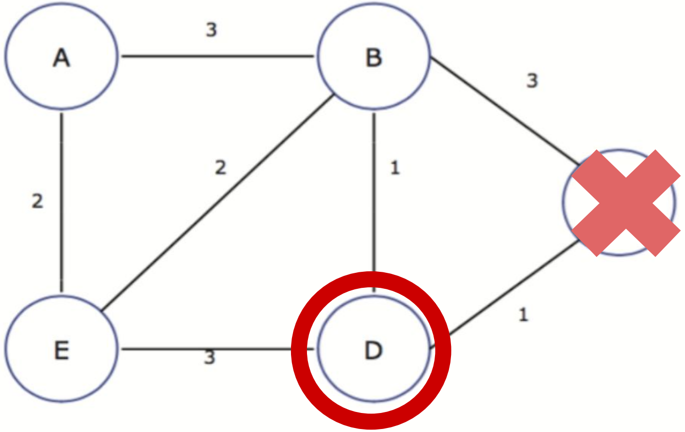


A	B	C	D	E
inf	inf	0	inf	inf

Поиск в ширину (1-k)

Решение 2

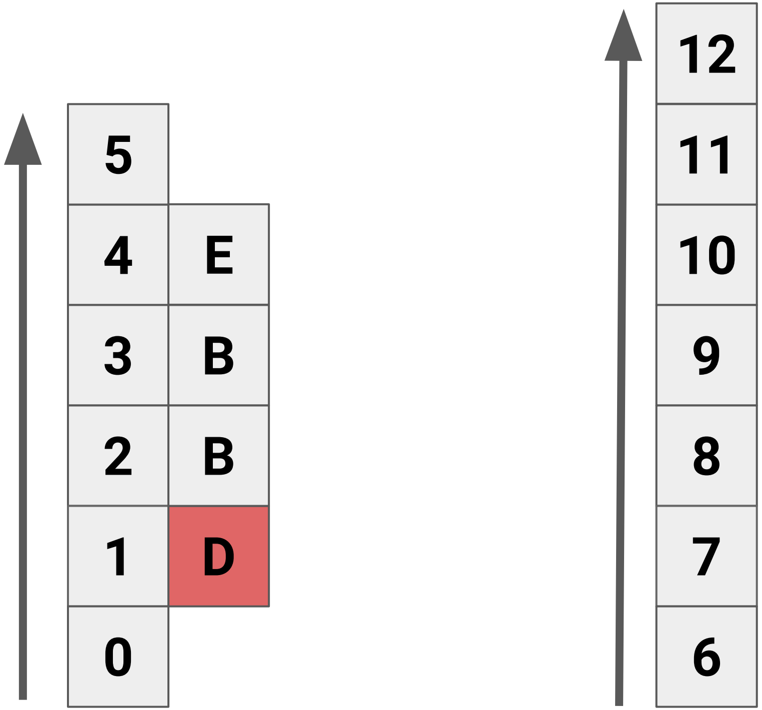


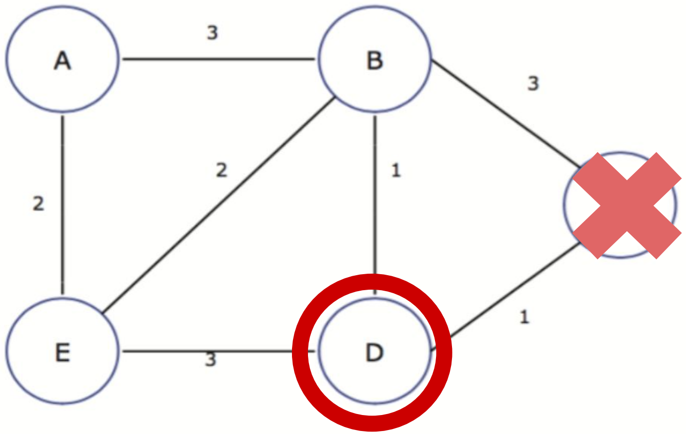


A	B	C	D	E
inf	inf	0	1	inf

Поиск в ширину (1-k)

Решение 2

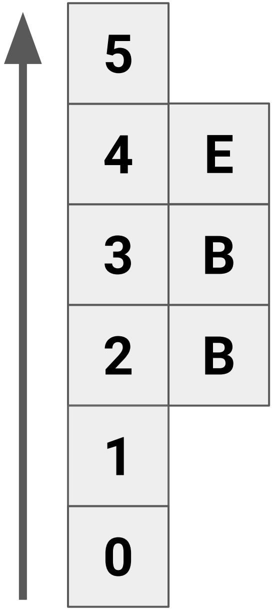


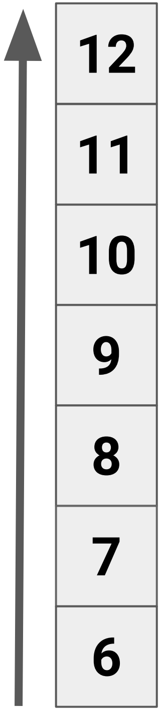


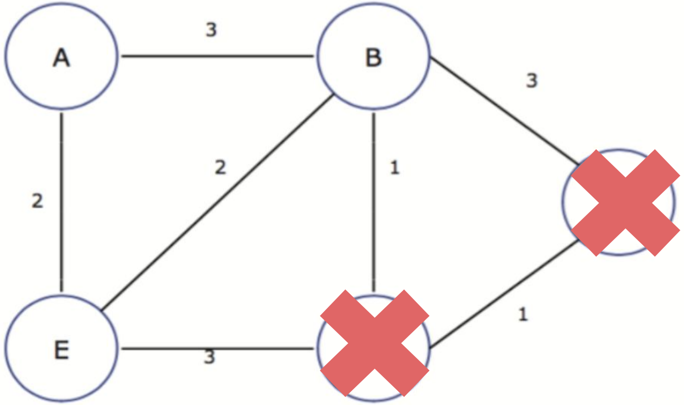
inf	inf	0	1	inf
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2



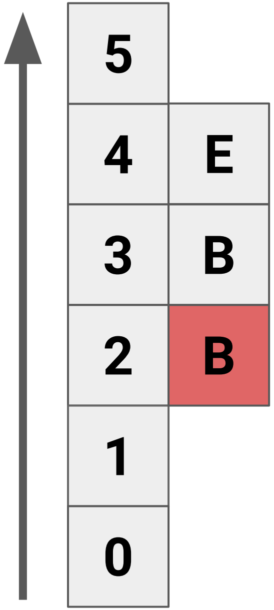


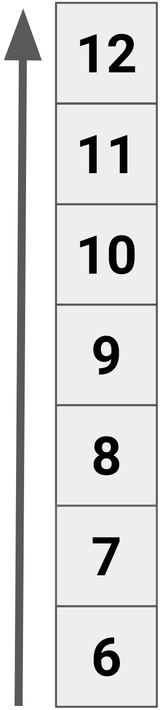


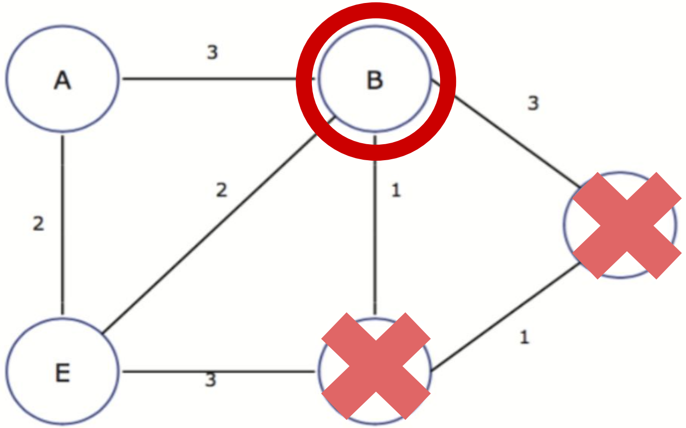
A	B	C	D	E
inf	inf	0	1	inf

Поиск в ширину (1-k)

Решение 2



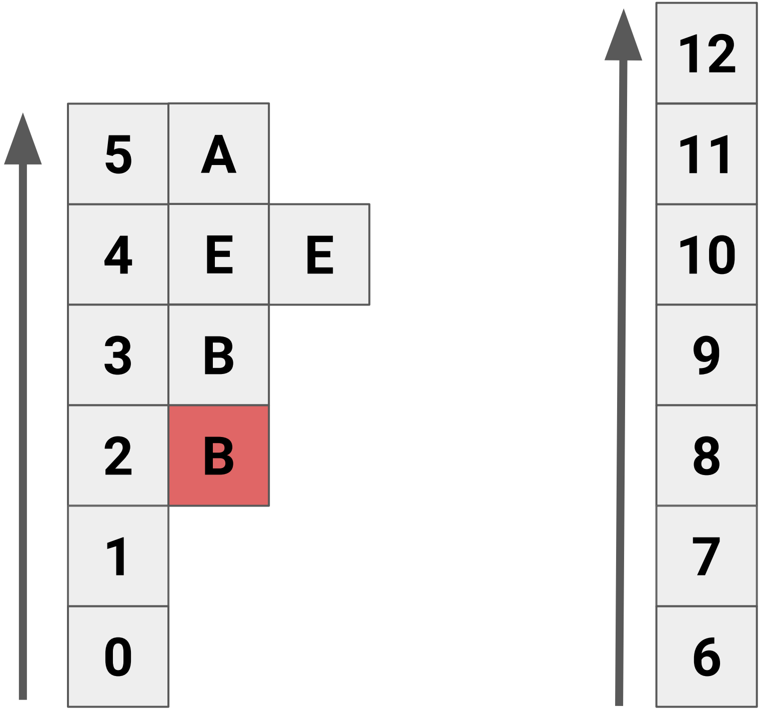


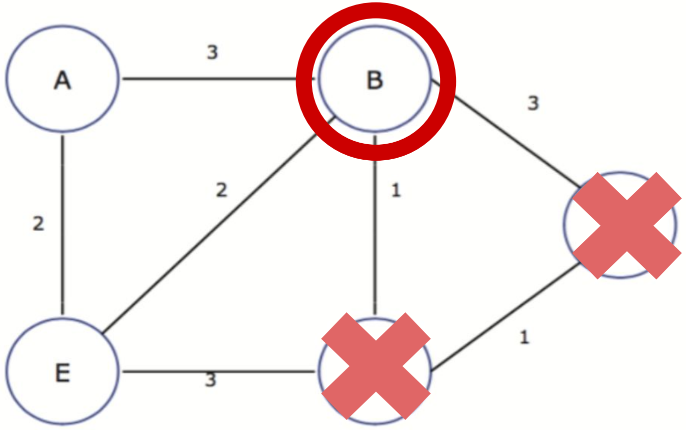


inf	2	0	1	inf
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2

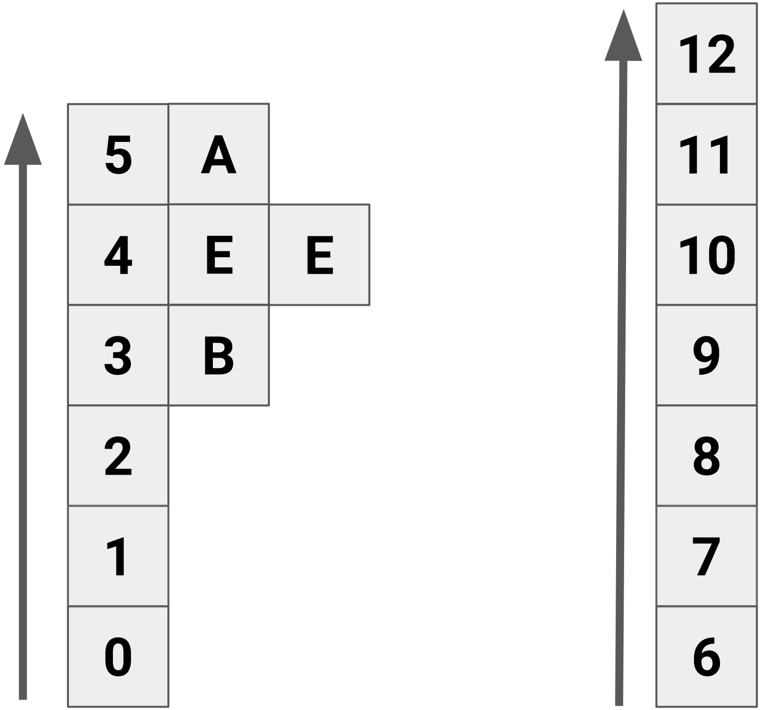


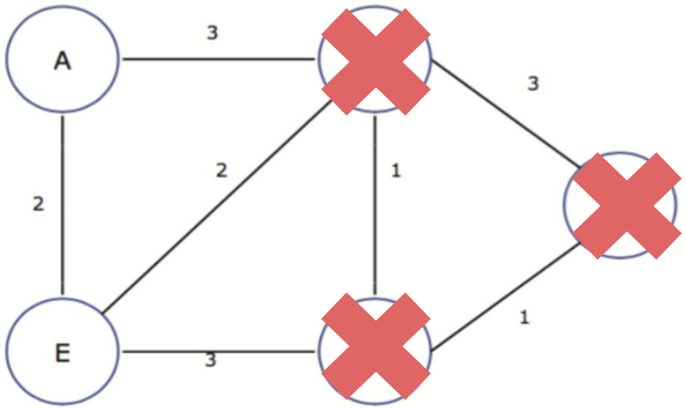


inf	2	0	1	inf
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2

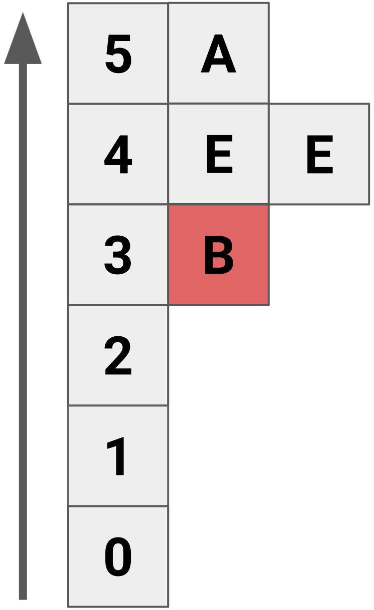


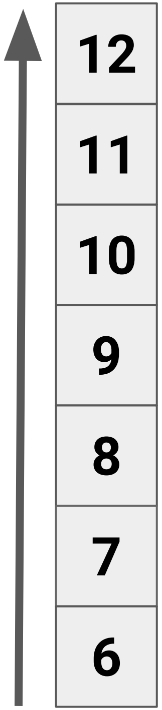


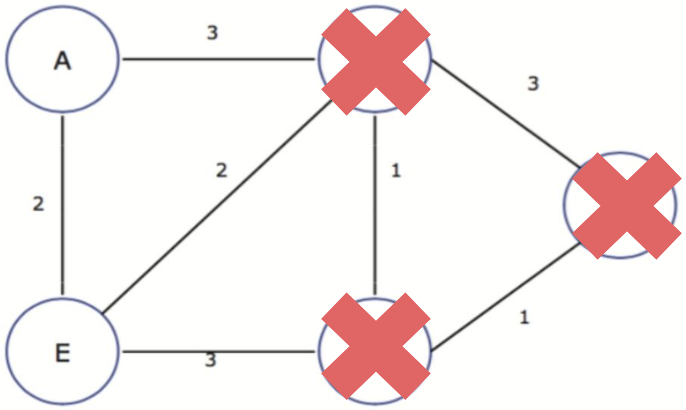
A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2



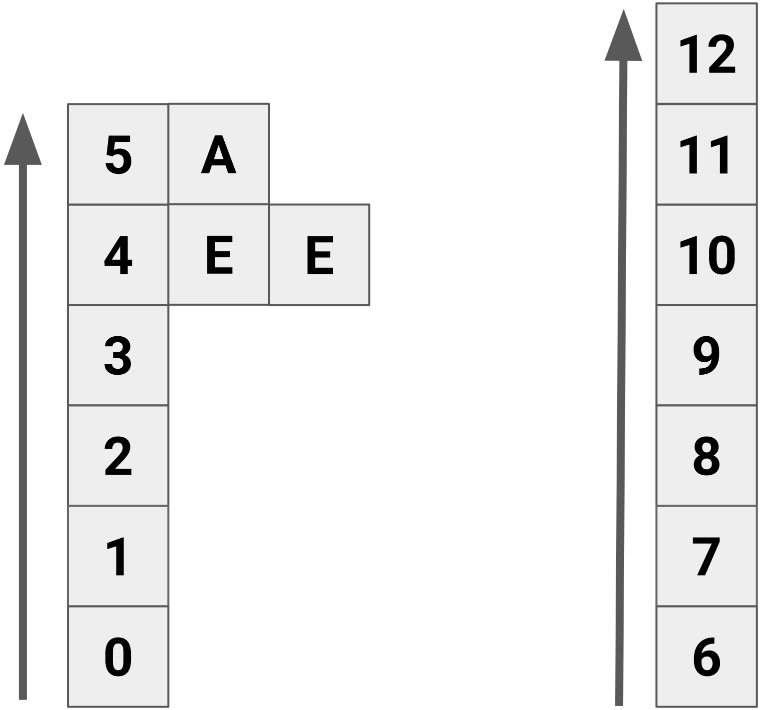


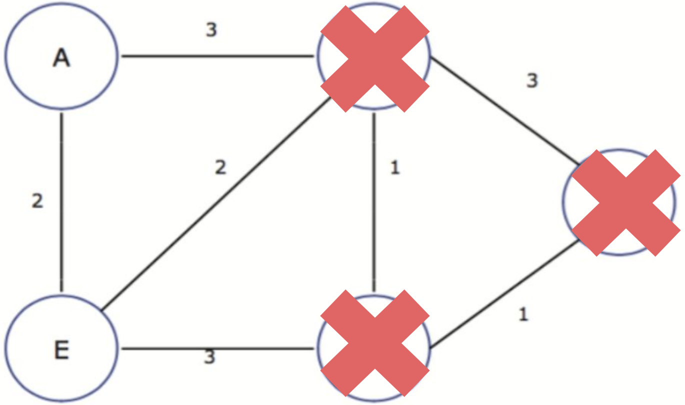


A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2

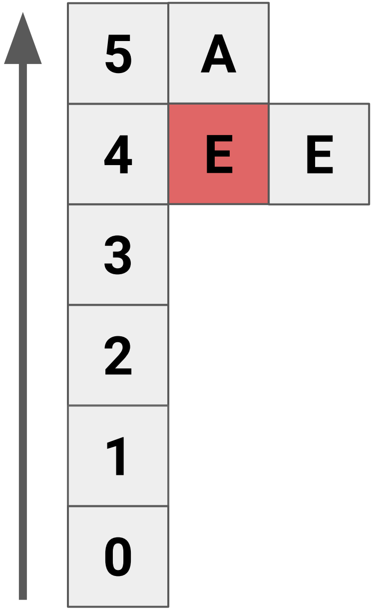


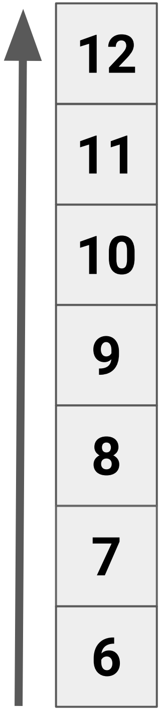


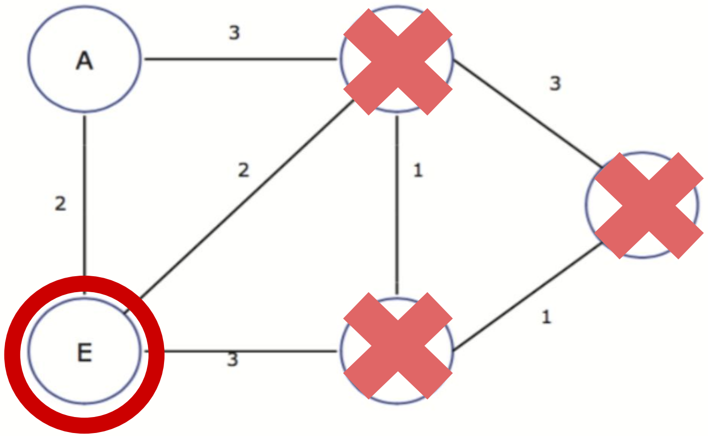
A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2



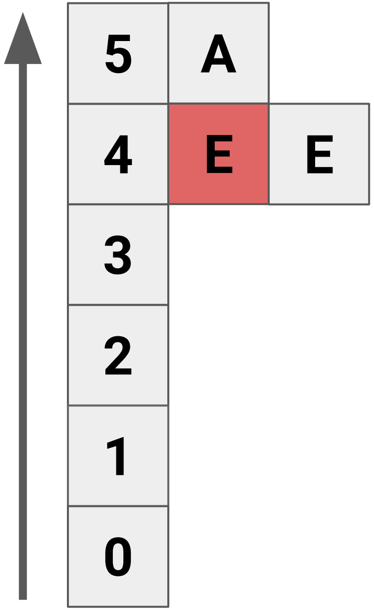


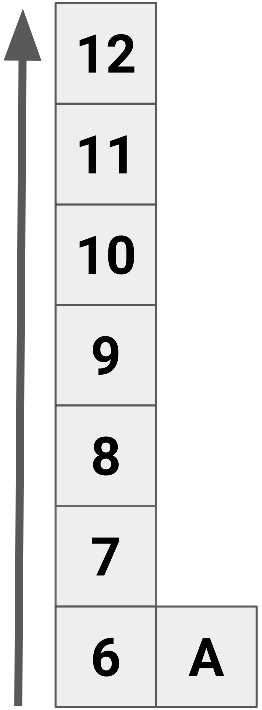


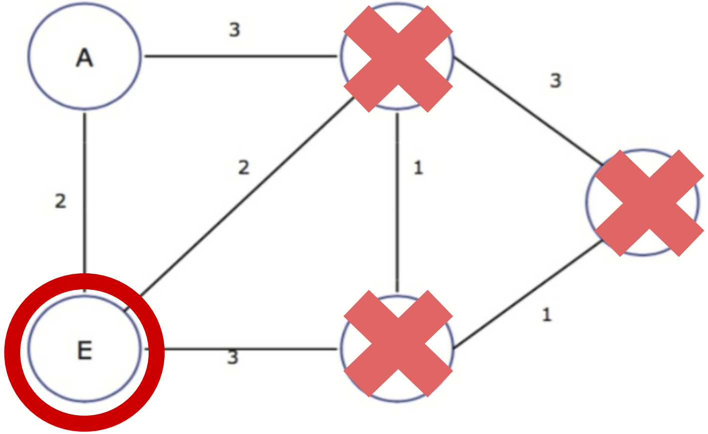
A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Решение 2



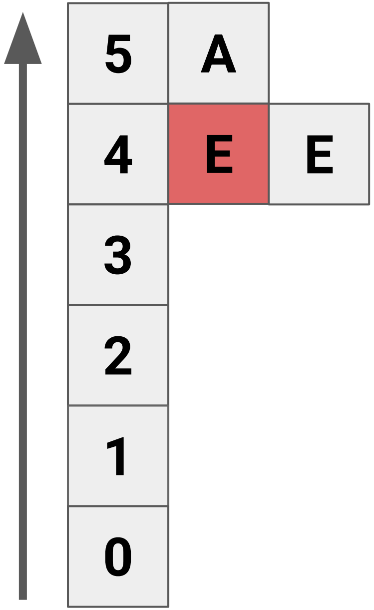


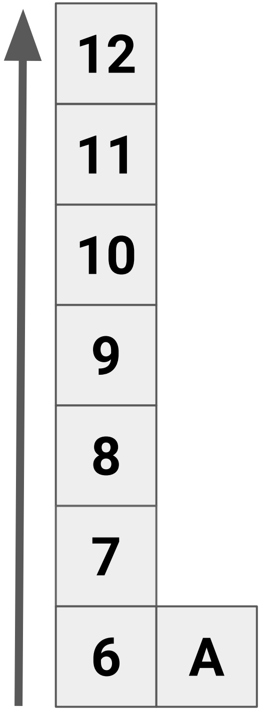


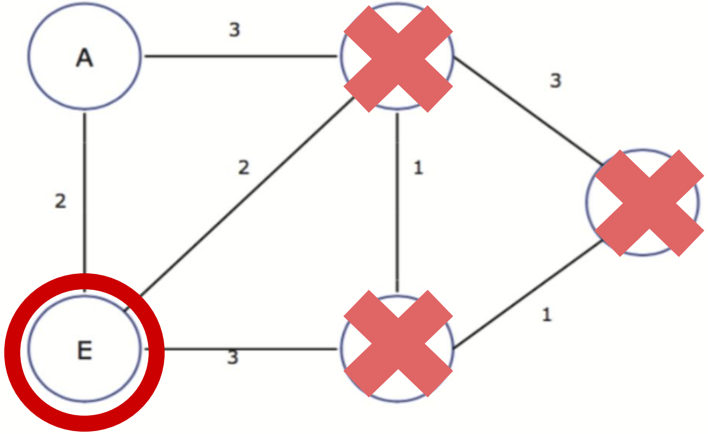
A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Решение 2



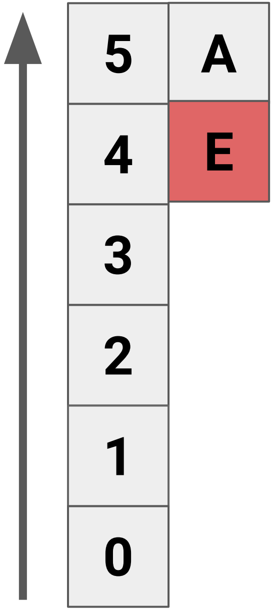


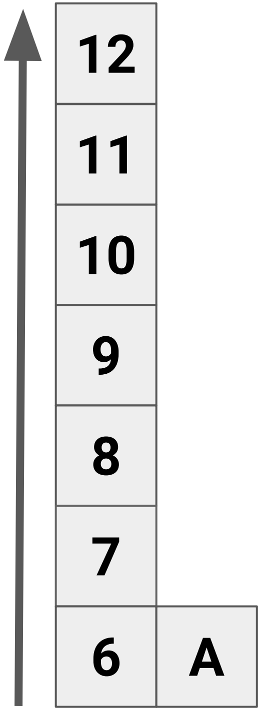


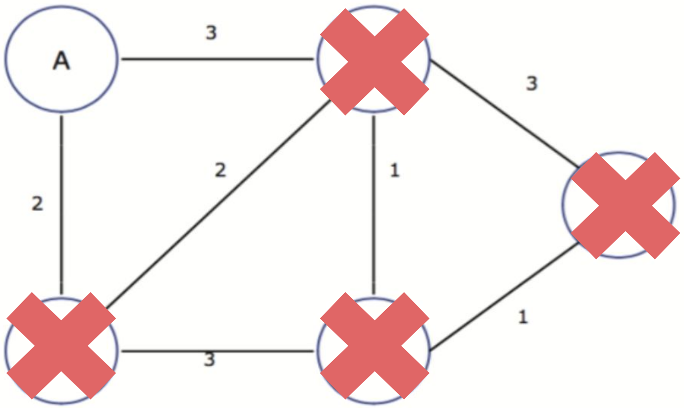
A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Решение 2



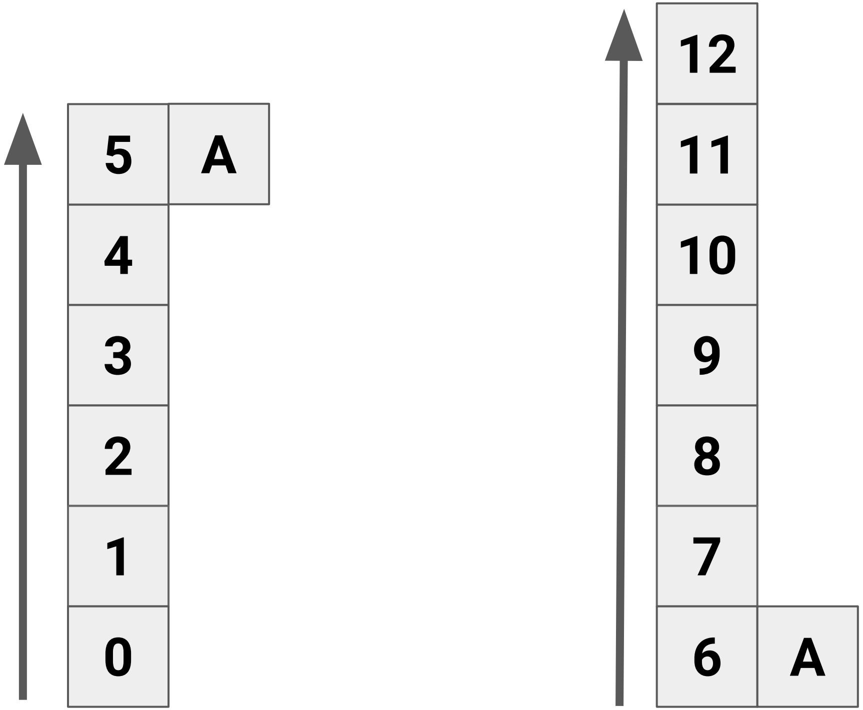


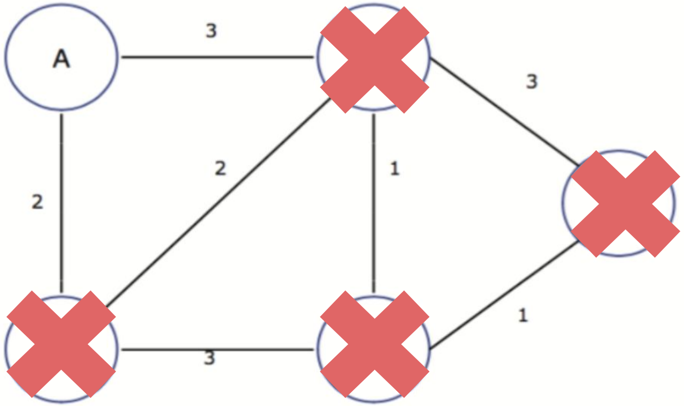


A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Решение 2

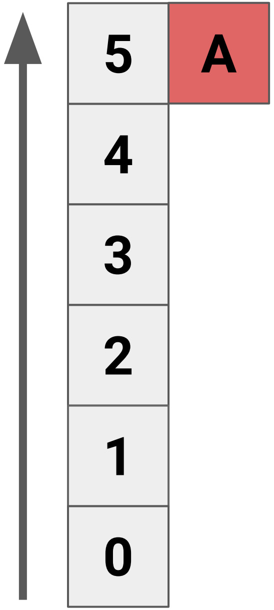


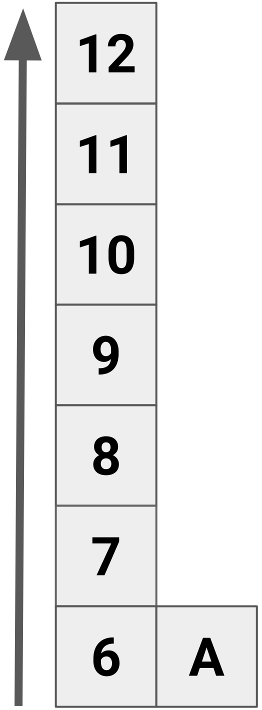


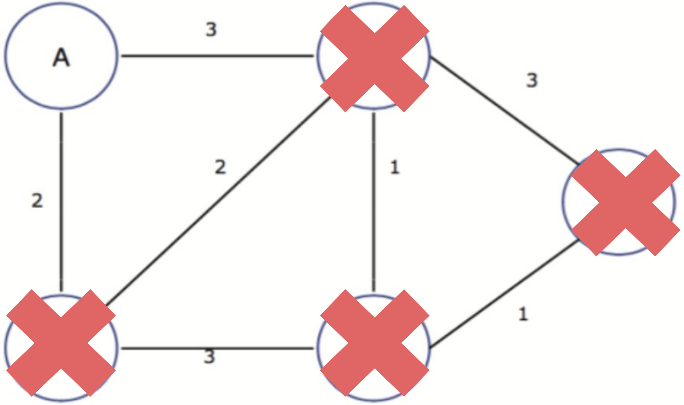
inf	2	0	1	4
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2



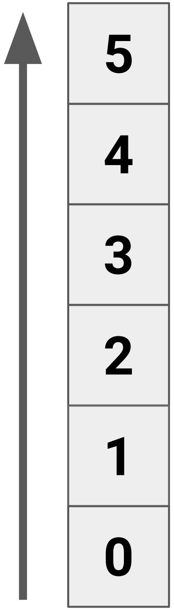


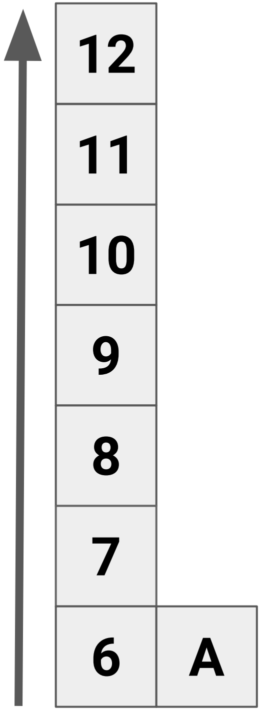


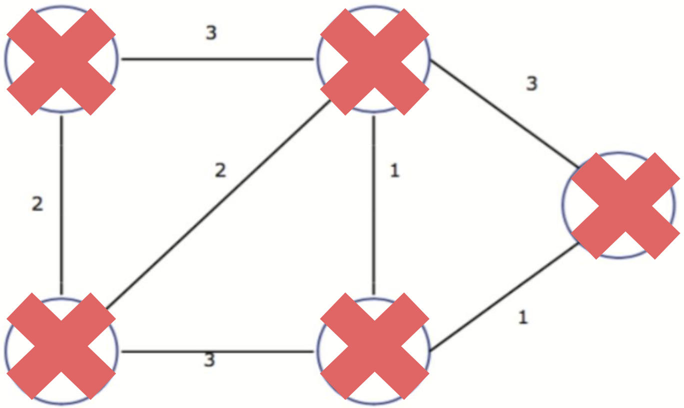
A	B	C	D	E
5	2	0	1	4

Поиск в ширину (1-k)

Решение 2



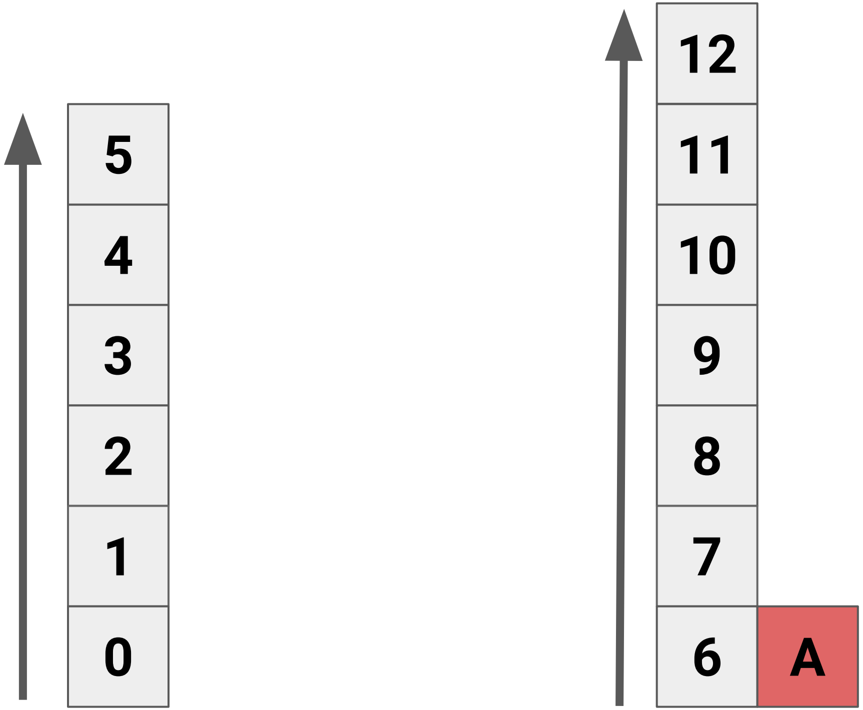


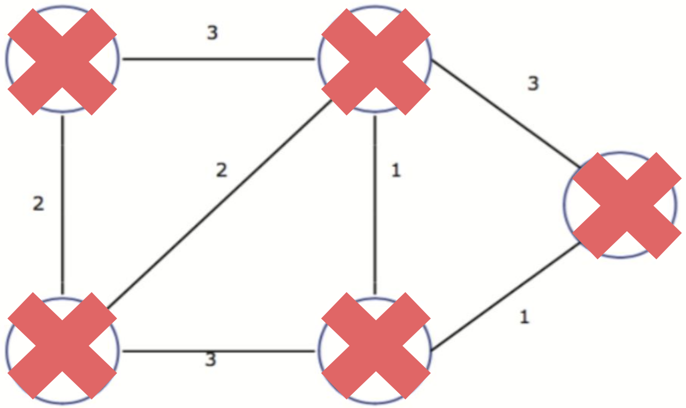


5	2	0	1	4
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2





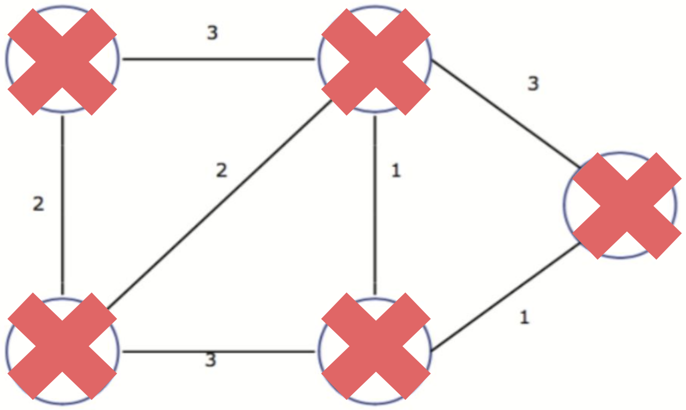
A	B	C	D	E
5	2	0	1	4

Поиск в ширину (1-k)

Решение 2

5
4
3
2
1
0

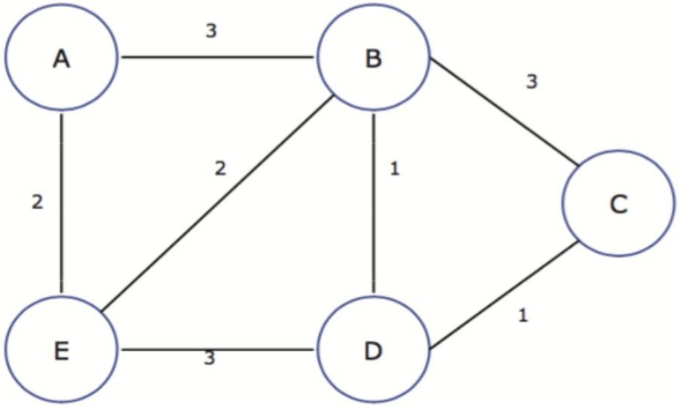
12
11
10
9
8
7
6



5	2	0	1	4
A	B	C	D	E

Поиск в ширину (1-k)

Решение 2



5	**2**	**0**	**1**	**4**
A	B	C	D	E

Поиск в ширину (1-k)

Замечание 1

А как же восстановление ответа?

Ответ:

1) заведем массив parent как в базовом случае
2) когда будем рассматривать v в очереди d и добавлять to в очередь d + w, давайте добавлять пару чисел { to, v }
3) когда впервые натыкаемся на вершину v (обновляем до нее расстояние), также можно записать в массив parent ее предка, которого мы сохранили вторым значением в паре

Поиск в ширину (1-k)

Замечание 2

O(Vk) - а не много ли памяти?

Ответ:

1) Действительно можно и меньше
2) Когда рассматриваем очередь d, то очевидно что элементов в очередях [0, d) уже нет и не будет
3) Давайте заведем всего k очередей, и когда они будут заканчиваться будем переиспользовать их сначала
4) по факту добавляем to в очередь (d + w) % queues.size()

Поиск в ширину (1-k)

Замечание 3

А что там с весом 0?

Ответ:
1) Существует и такая разновидность поиска (0-k)
2) Данный алгоритм также корректно работает и с 0 весами
3) Если между v и to, вес w = 0, то добавляем to в эту конец этой же очереди

Поиск в ширину (1-k)

Замечание 3

А что там с весом 0?

1) Существует даже специальный вид поиска 0-1 BFS
2) Из предыдущего шага можно понять что нужно 2 очереди.
3) Но можно обойтись одной “продвинутой” очередью - деком
4) Тогда делаем обычный BFS, только если встречаем нулевое ребро - добавляем вершину в начало дека (0-вая очередь)
5) Ребра второго типа будем добавлять в конец (1-ая очередь)

Поиск в ширину (1-k)

Замечание 3

А что там с весом 0?

```cpp
#include <deque>

...

dist[0] = 0;
deque<int> d;
d.push_back(0);

while (!d.empty()) {
    int v = d.front();
    d.pop_front();

    for (int i = 0; i < graph[v].size(); ++i) {
        int to = graph[v][i].first;
        int w = graph[v][i].second;

        if (dist[to] > dist[v] + w) {
            dist[to] = dist[v] + w;
            parent[to] = v;
            if (w == 0) {
                d.push_front(to);
            } else {
                d.push_back(to);
            }
        }
    }
}
```

Поиск в ширину (1-k)

Замечание 3

А что там со сложностью алгоритма?

Ответ:

O(Vk + E) - поскольку в худшем случае пройдем все Vk очередей, хоть они и не пустые - каждая вершина будет встречаться только в k очередях, что не портит асимптотику

Поиск в ширину (1-k)

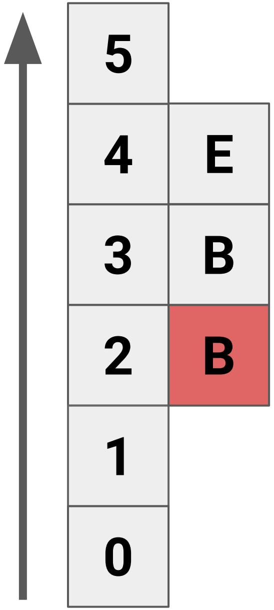
Замечание 4

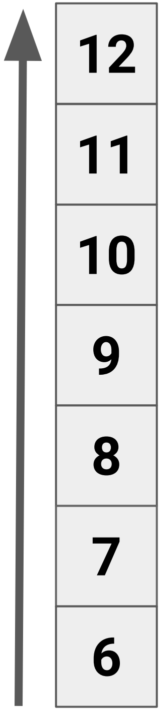
А можно как-то вообще избавиться от k в сложности?

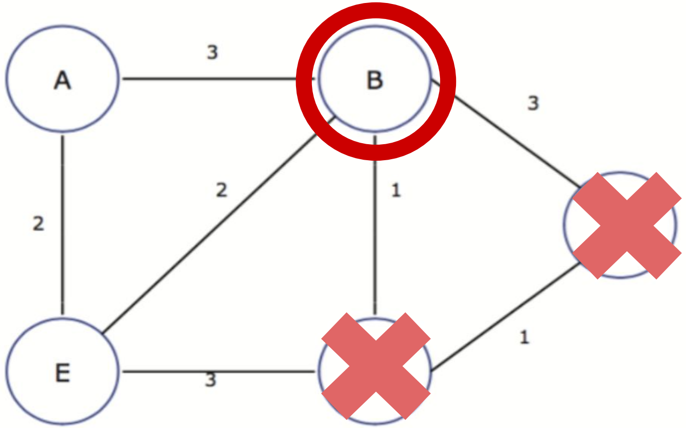
Ответ:
1) Да, можно, давайте посмотрим как

Поиск в ширину (1-k)

Решение 2







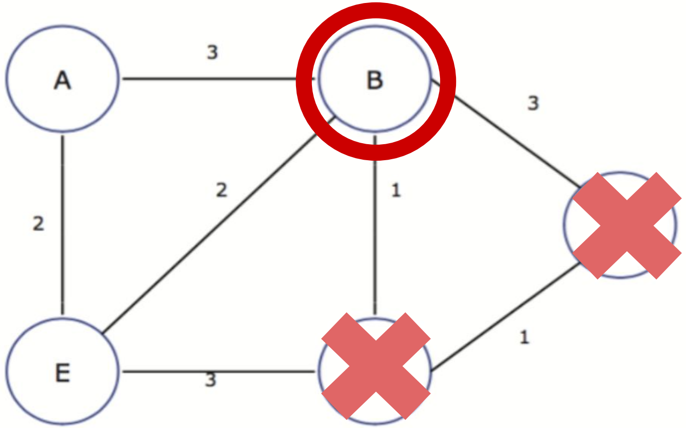
A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2

5	A	
4	E	E
3	B	
2	B	
1		
0		

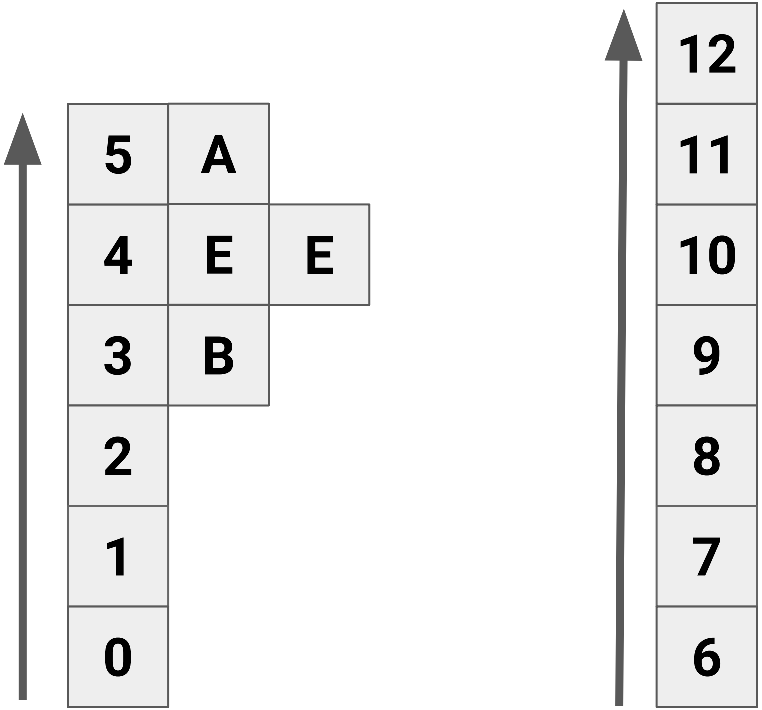
12
11
10
9
8
7
6

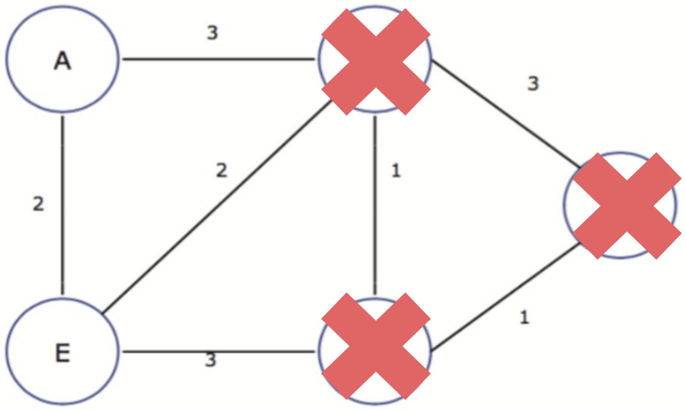


A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2

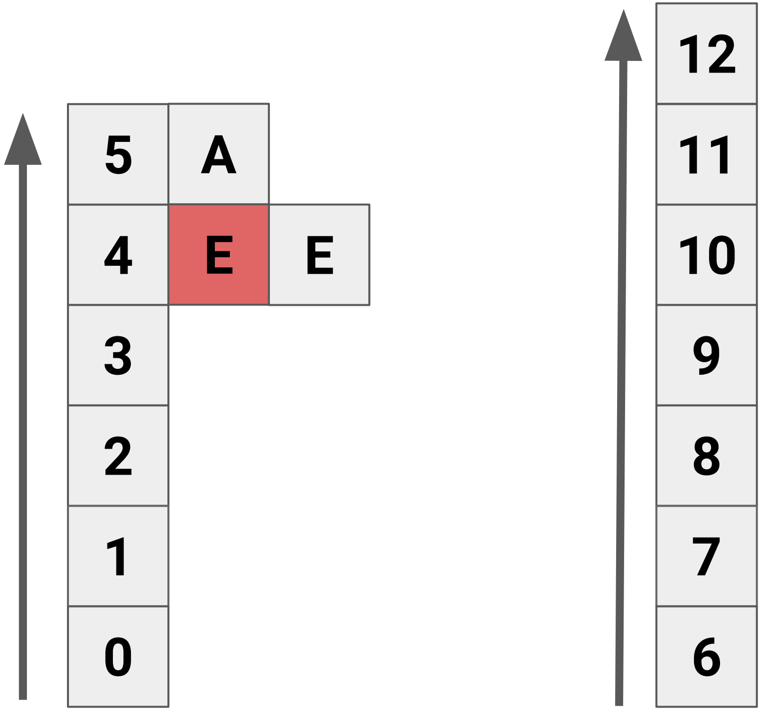


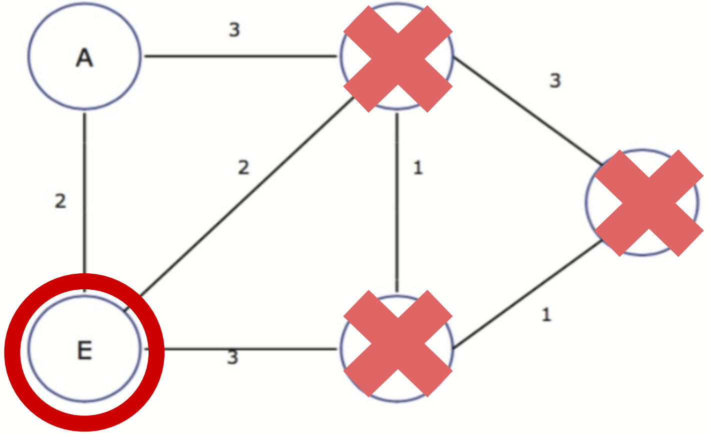


A	B	C	D	E
inf	2	0	1	inf

Поиск в ширину (1-k)

Решение 2





A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Замечание 4

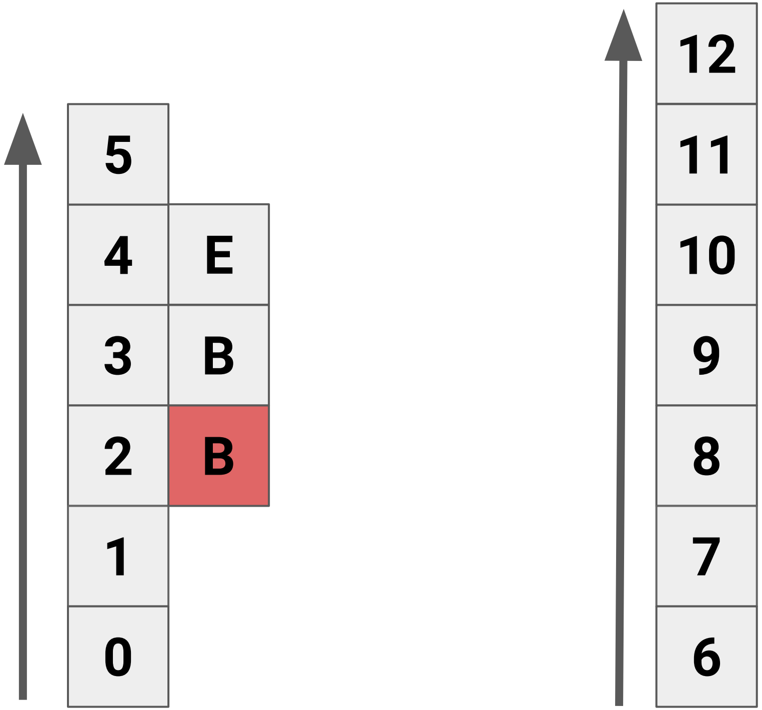
А можно как-то вообще избавиться от k в сложности?

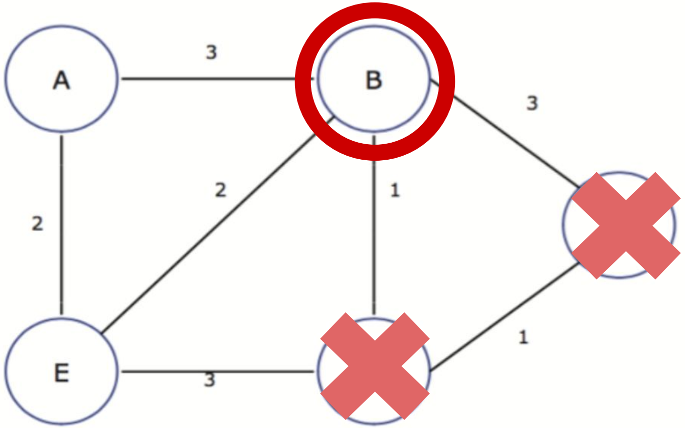
Ответ:

1) Рассматриваем вершину v и to, вес w, давайте сразу попробуем обновить dist[to]
2) dist[to] = min(dist[to], dist[v] + w)
3) После того как закончили с v, помечаем ее посещенной и переходим к следующей вершине next, которая еще не посещена и среди всех не посещенных dist[next] - минимальный

Поиск в ширину (1-k)

Модификация решения 2

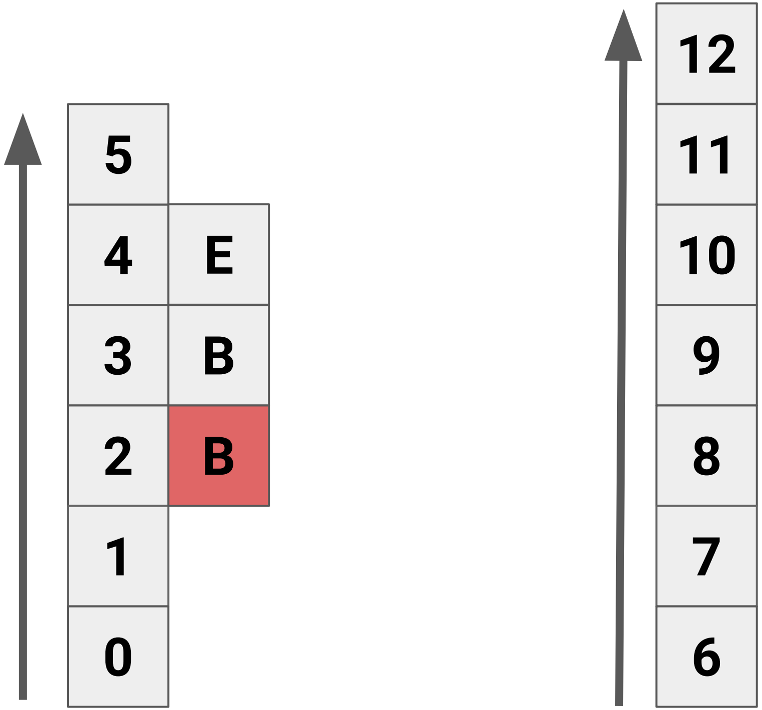


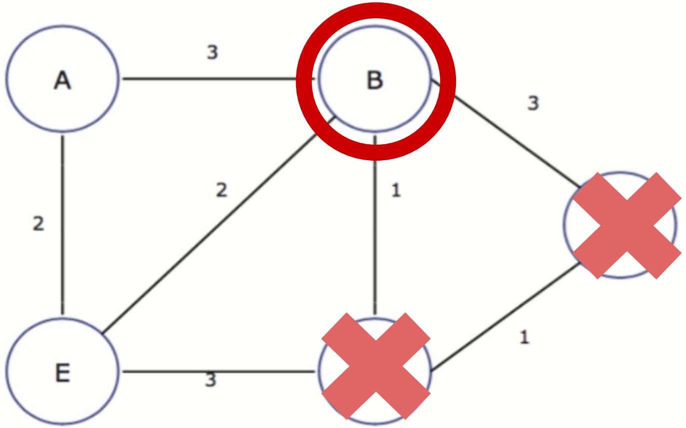


inf	2	0	1	inf
A	B	C	D	E

Поиск в ширину (1-k)

Модификация решения 2

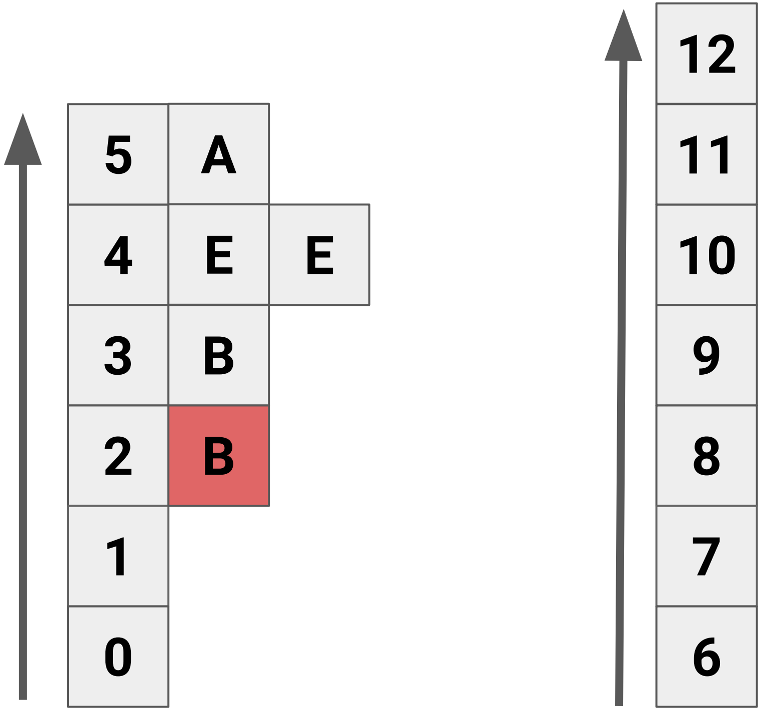


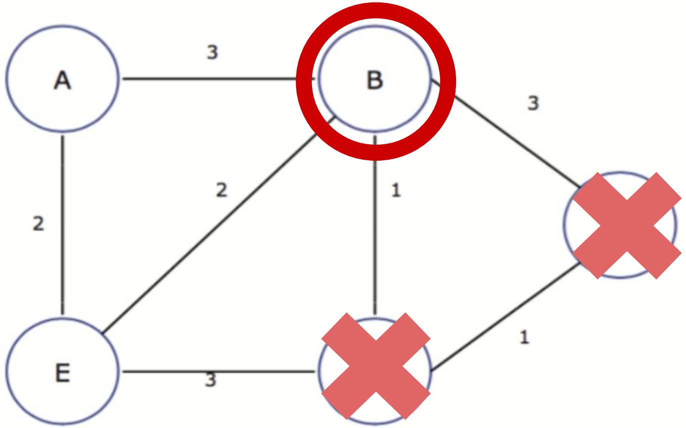


A	B	C	D	E
inf	2	0	1	4

Поиск в ширину (1-k)

Модификация решения 2

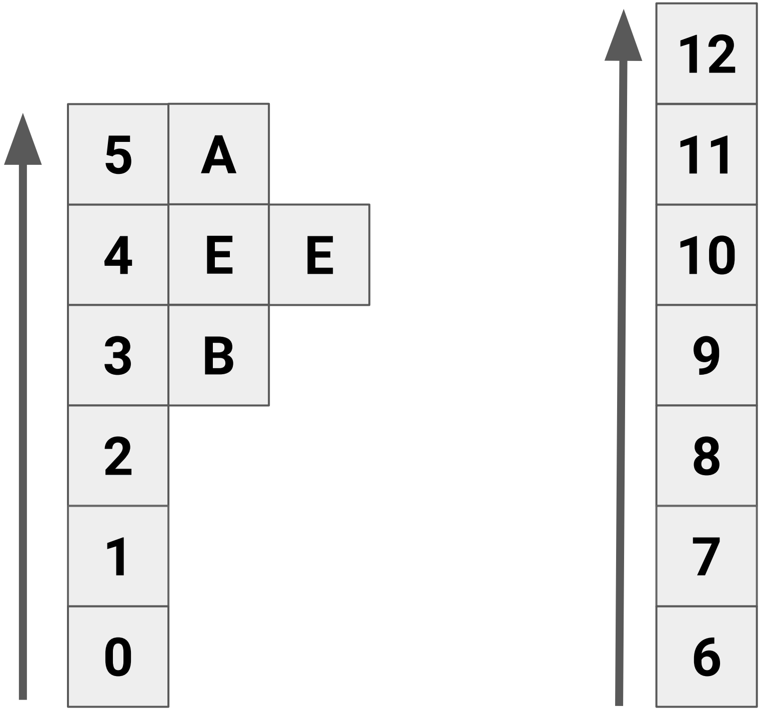


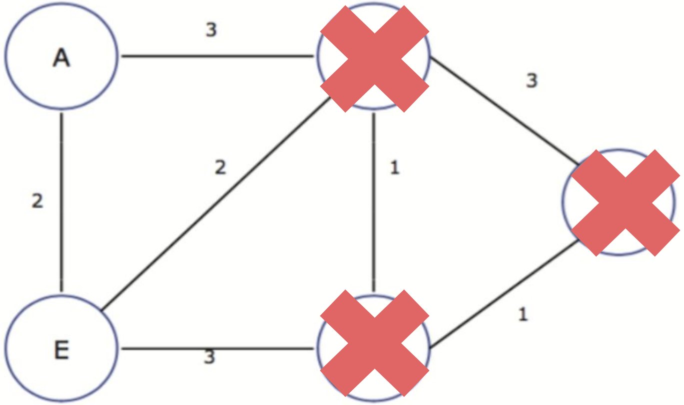


A	B	C	D	E
5	2	0	1	4

Поиск в ширину (1-k)

Модификация решения 2

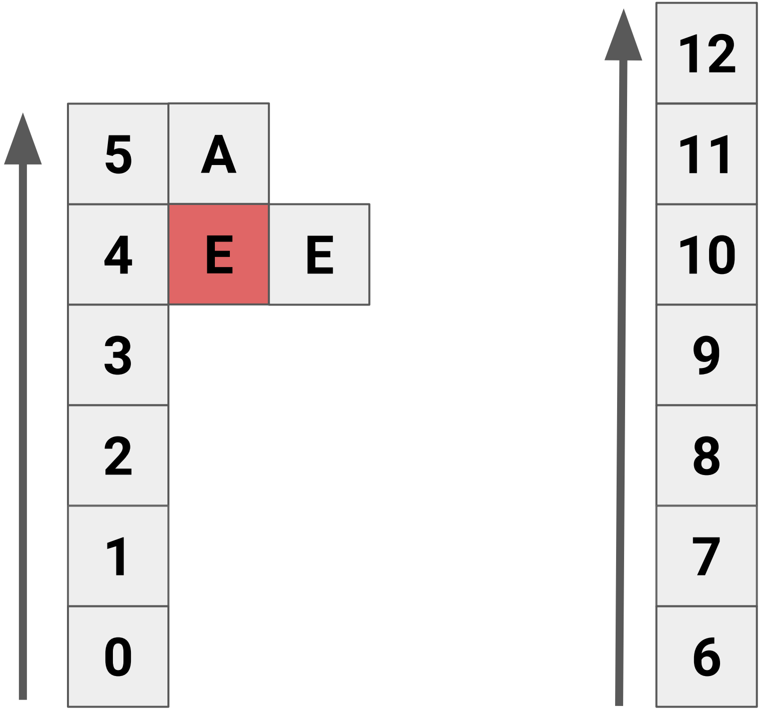


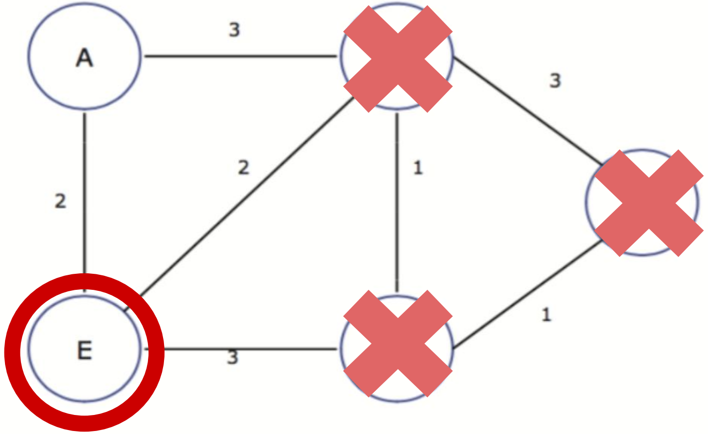


A	B	C	D	E
4	2	0	1	5

Поиск в ширину (1-k)

Модификация решения 2





A	B	C	D	E
5	2	0	1	4

Поиск в ширину (1-k)

Замечание 5

Окей, от К избавились, а сложность то какая теперь?

Ответ:

1) Пусть рассмотрели вершину V, теперь ее помечаем и хотим найти минимальную среди не помеченных O(V)
2) Так как после шага 1 вершина V помечена, то к ней мы больше не вернемся
3) Значит всего операций поиска минимума будет V
4) Итоговая сложность O(V * V + E) = O(V * V) поскольку E < V * V (в полном графе V * (V - 1) / 2 ребер)

Поиск в ширину (1-k)

Замечание 6

Какие еще преимущества мы получили?

Ответ:
1) Можем работать с любыми неотрицательными весами (поскольку избавились от ограничения по K)
2) Веса могут быть даже вещественные!
3) Алгоритм можно улучшить: искать минимум за log V, а не V

Поиск в ширину (1-k)

Замечание 6

Какие еще преимущества мы получили?

Ответ:

1) Можем работать с любыми неотрицательными весами (поскольку избавились от ограничения по K)
2) Веса могут быть даже вещественные!
3) Алгоритм можно улучшить: искать минимум за log V, а не V

4) Это и есть алгоритм Дейкстры :)

Алгоритм Дейкстры

Реализация

```
ios_base::sync_with_stdio(false);
cin.tie(nullptr);

const int INF = ...;
vector<pair<int, int>> graph[MAXN];
// graph[v][i].first = to
// graph[v][i].second = w
// n - вершин, m - ребер, s - стартовая
(0-индексация)
// ... считываем граф... (слайд 22)

vector<int> dist(n, INF);
vector<int> parent(n, - 1);
distance[s] = 0;
vector<bool> used(n, false);
```

Алгоритм Дейкстры

Реализация

```
for (int i = 0; i < n; ++i) {
    int v = -1;
    for (int j = 0; j < n; ++j) {
        if (!used[j] && (v == -1 || dist[j] < dist[v])) {
            v = j;
        }
    }

    if (dist[v] == INF)
        break;

    used[v] = true;

    for (size_t j = 0; j < g[v].size(); ++j) {
        int to = graph[v][j].first, w = graph[v][j].second;
        if (dist[v] + w < dist[to]) {
            dist[to] = dist[v] + w;
            parent[to] = v;
        }
    }
}
```

Обозначения

v, to - вершины (из v в to)

w - вес ребра

d - расстояние