Алгоритмы и структуры данных-2

Графы-2. Минимальный остов

Практическое занятие 05 10–15.02.2025

2024-2025 учебный год

План

Повторение: связность, топологическая сортировка, эйлеровы графы

Построение минимального остовного дерева: алгоритм Краскала

Реализация структура данных `UNION-FIND`

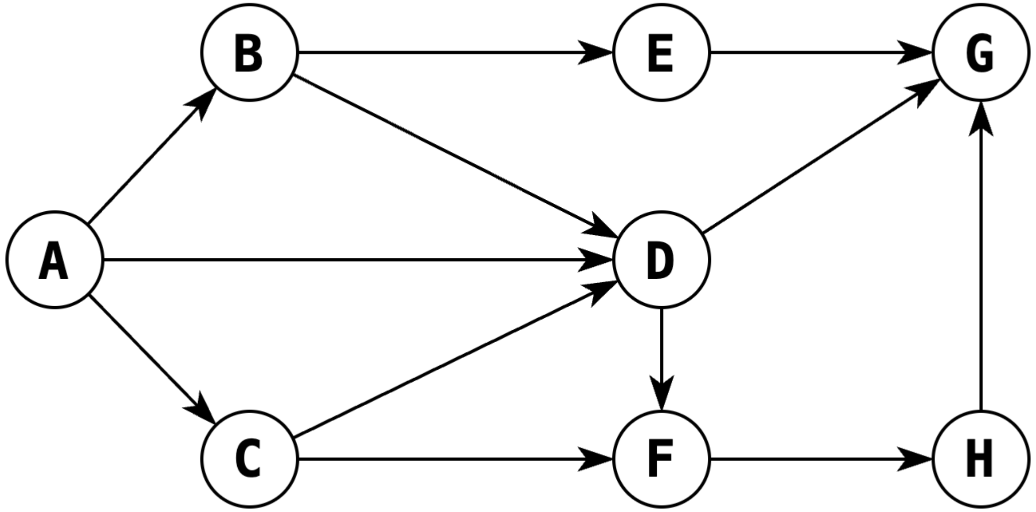
Warm-up

алгоритмы на графах

Упражнение 1 – Анализ связности

1. Какое минимальное число ребер надо добавить в ориентированный граф, чтобы он стал сильно связным?
2. Предположим, что задан неориентированный связный граф. Каким образом ориентировать его ребра так, чтобы получившийся граф стал сильно связным?
3. Определите максимально возможное количество мостов и точек сочленения в графе из n вершин.

Упражнение 2 – TOPOLOGICAL SORT



1. Постройте возможный топологический порядок на вершинах данного графа путем последовательного удаления вершин с нулевыми степенями захода.
2. Сколько всего существует топологических порядков на вершинах данного графа?
3. При каких условиях граф будет иметь более одного топологического порядка?

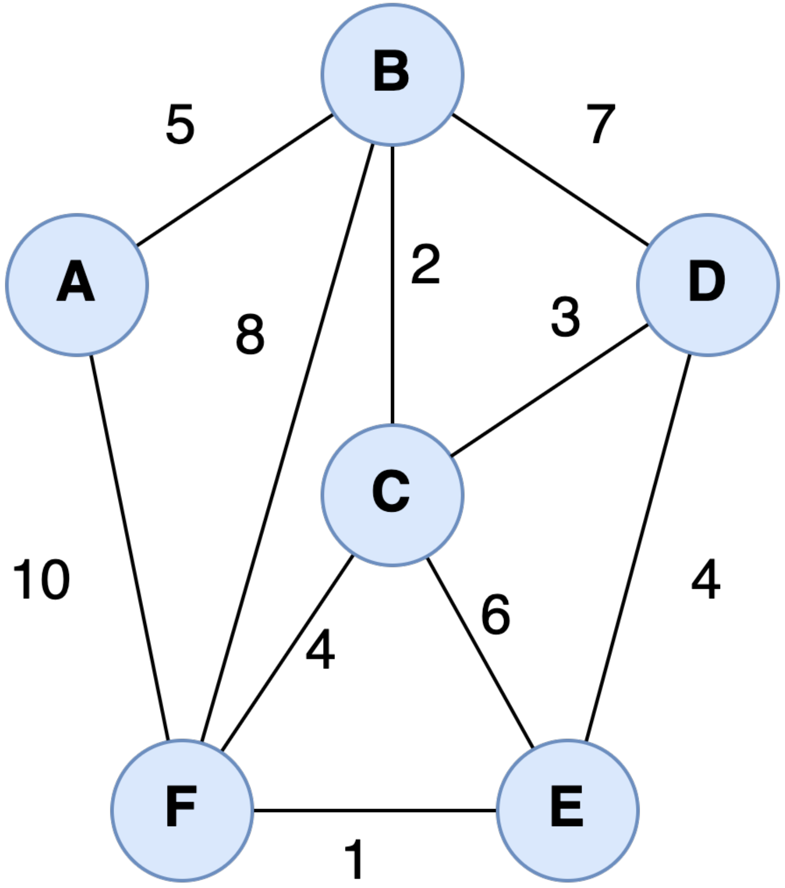
Упражнение 3 – Эйлеровы графы

Какое минимальное количество ребер надо добавить в граф, чтобы в нем образовался эйлеров цикл в случае:

- связного неориентированного графа,
- связного ориентированного графа,
- несвязного неориентированного графа,
- несвязного ориентированного графа.

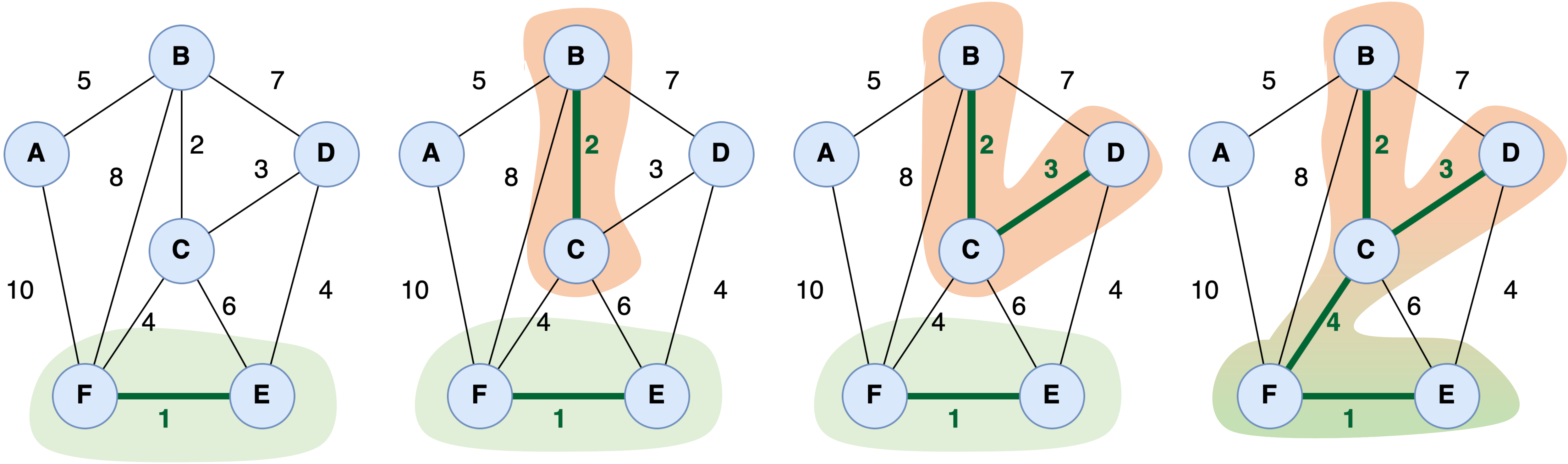
Минимальное остовное дерево **MST**

алгоритм Краскала



SlowKruskal.cpp

```
G = (V, E) - граф

отсортировать ребра E в порядке неубывания

MST = {}

for (edge : sorted_E)
    if (добавление edge не образует цикл)
        MST.add(edge)

return MST
```



Добавление ребра с минимальным весом D–E невозможно, так как образуется цикл C–D–E–F–C

Для улучшения базовой реализации алгоритма Краскала и быстрой проверки на возможность образования цикла используется структура UNION-FIND

UNION-FIND

особенности реализации

О структуре

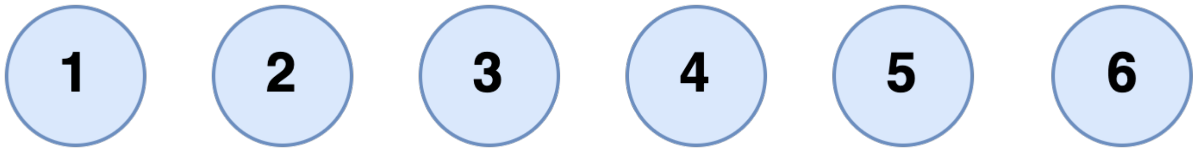
Представление и обработка семейств дизъюнктивных (непересекающихся) множеств, состоящих из однородных объектов

Bernard A. Galler
Michael J. Fischer
1964

DisjointSetUnion

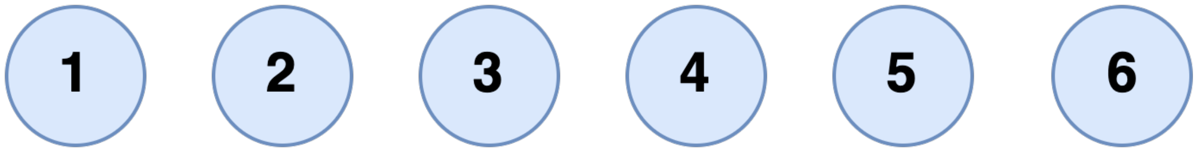
```cpp
template<typename ValueType>
class UnionFind {
public:
    void makeSet(ValueType value) {
        // создает множество из одного элемента
    }

    ValueType find(ValueType value) {
        /* в каком множестве содержится value?
           выводит "представителя" множества */
    }

    void union(ValueType value1, ValueType value2) {
        /* производит объединение множеств,
           содержащих value1 и value2 */
    }

private:
    /* внутренний контейнер для хранения
       обычно: одномерный массив */
    std::vector<ValueType> array_;
}
```

Начальное состояние:
множества-синглтоны




```
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
```

Начальное состояние:
множества-синглтоны

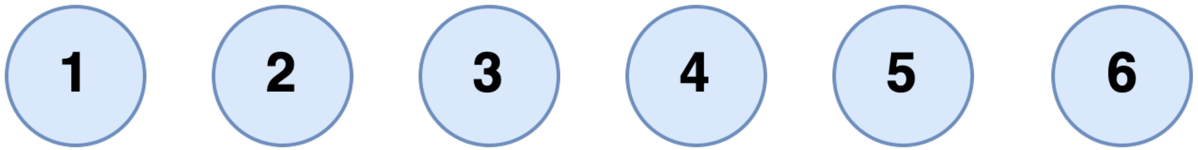



```
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
```

Объединение множеств:

```
union(1,2)
union(3,4)
union(2,5)
```

Начальное состояние:
множества-синглтоны

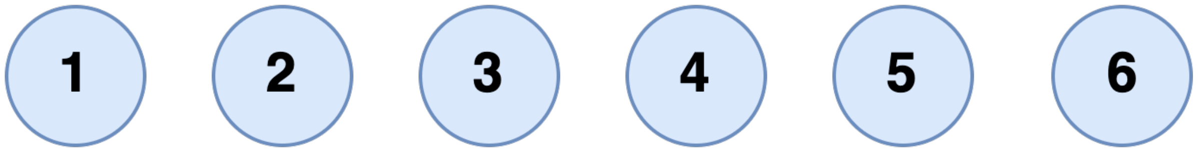



```
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
```

Объединение множеств:

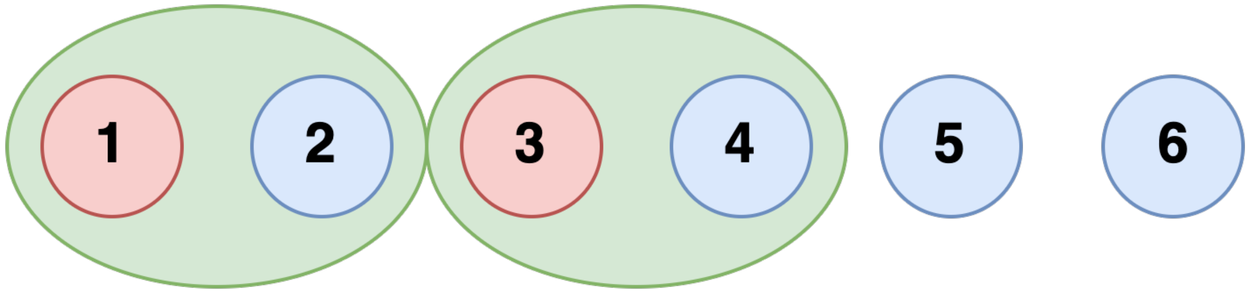
```
union(1,2)
union(3,4)
union(2,5)
```

Начальное состояние:
множества-синглтоны



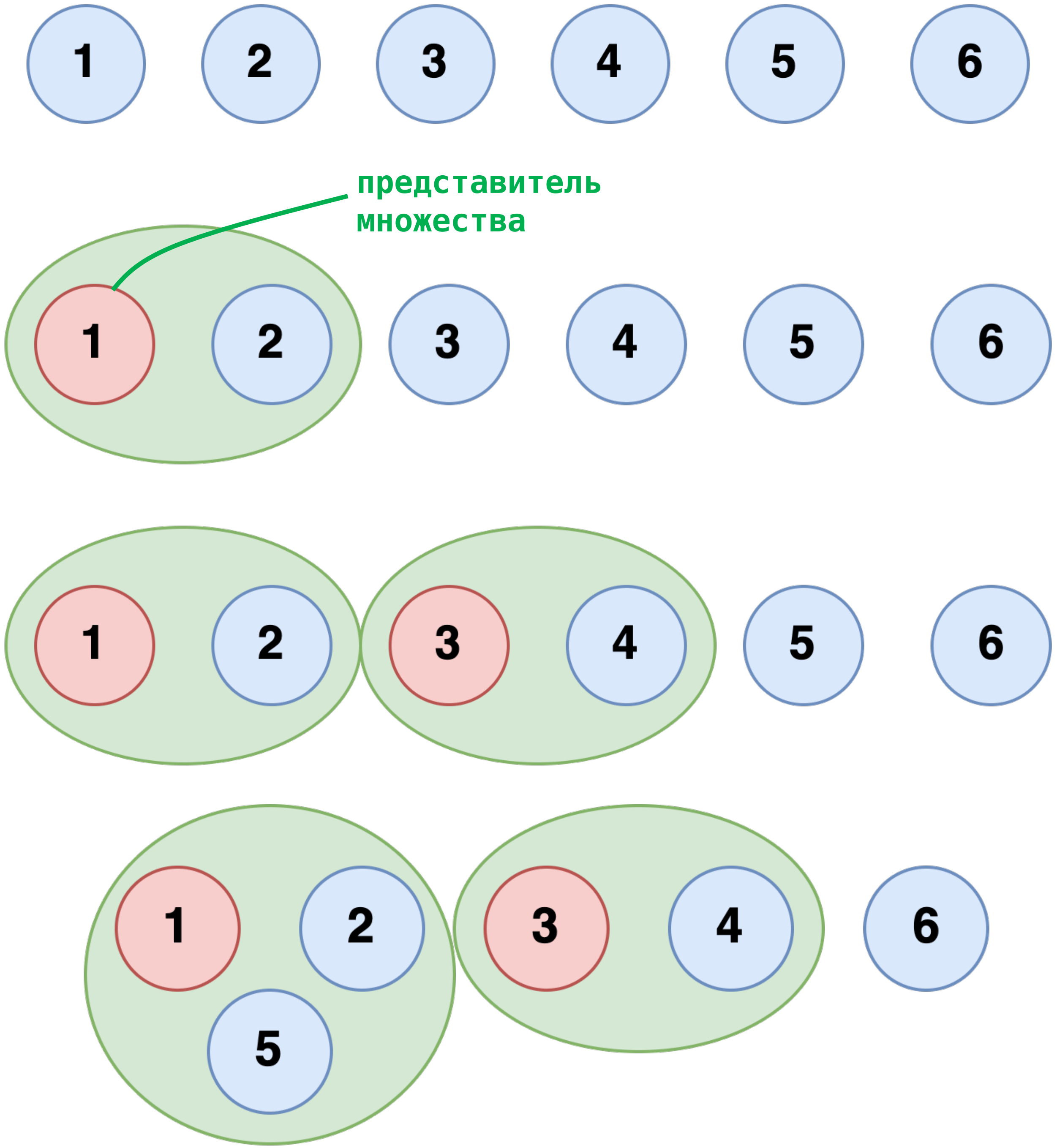

```
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
```

Объединение множеств:




```
union(1,2)
union(3,4)
union(2,5)
```

Начальное состояние: множества-синглтоны




```
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
```

Объединение множеств:

```
union(1,2)
union(3,4)
union(2,5)
```

Начальное состояние:
множества-синглтоны

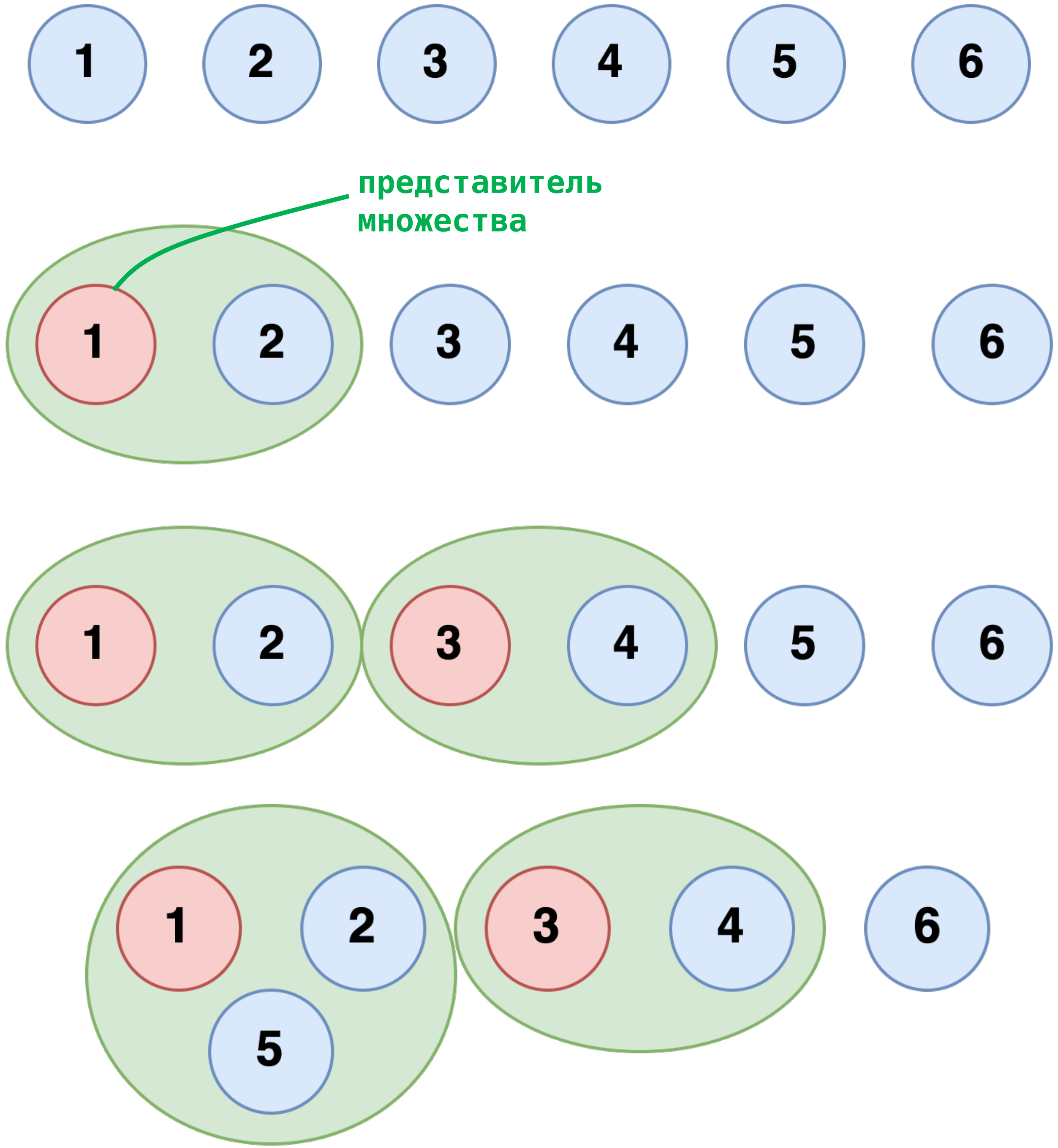
```
makeSet(1) makeSet(2) makeSet(3)
makeSet(4) makeSet(5) makeSet(6)
```

Объединение множеств

```
union(1,2)
union(3,4)
union(2,5)
```

В каком множестве объект?

```
find(1)
find(4)
find(6)
```



Начальное состояние:
множества-синглтоны

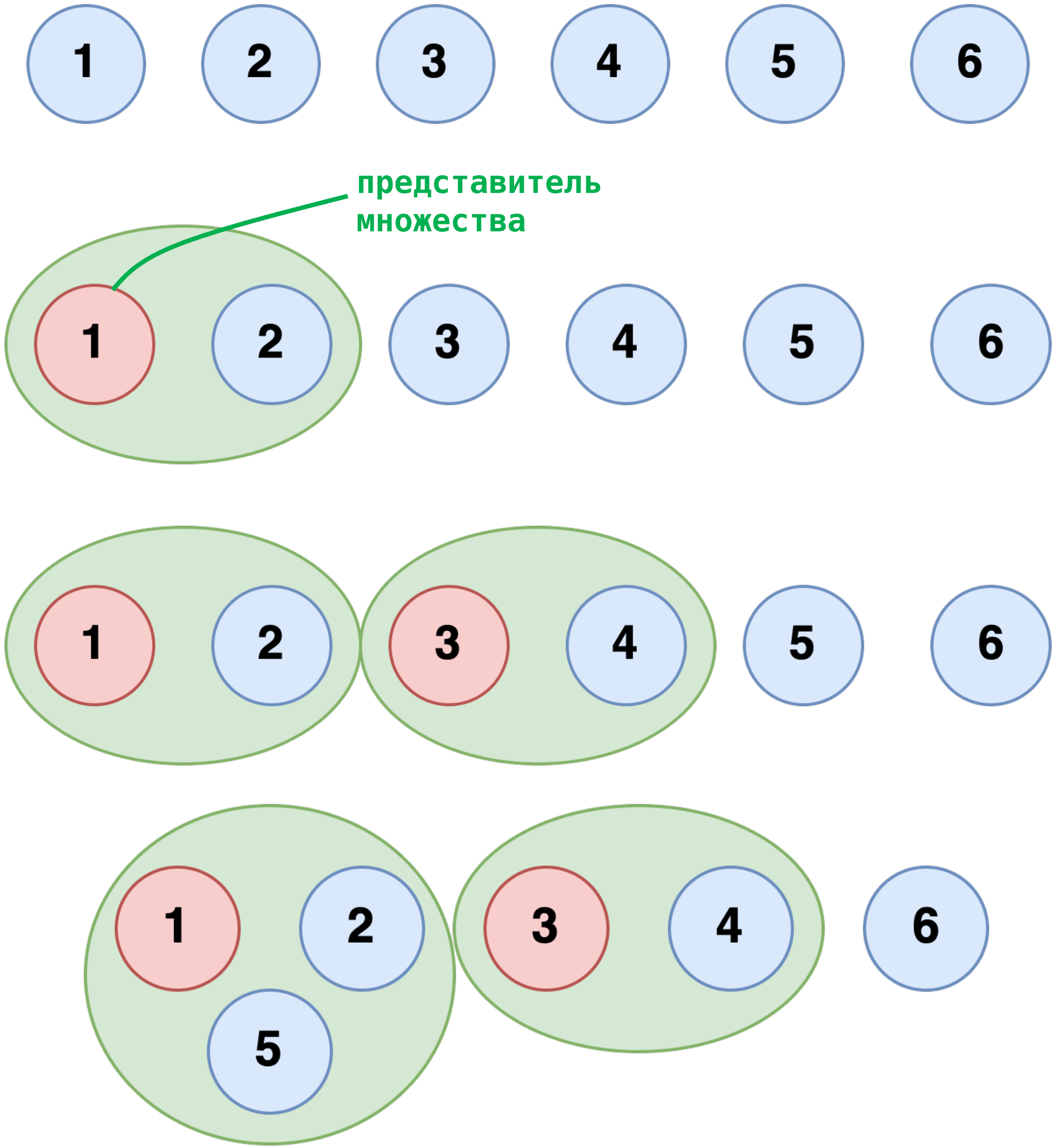
```
makeSet(1) makeSet(2) makeSet(3)
makeSet(4) makeSet(5) makeSet(6)
```

Объединение множеств

```
union(1,2)
union(3,4)
union(2,5)
```

В каком множестве объект?

```
find(1) = 1
find(4) = 3
find(6) = 6

find(1) == find(5)
find(2) != find(4)
```



Реализация UNION-FIND

1. На основе **массива**, в котором хранятся представители множества
2. На основе **списка**, в котором хранятся указатели на представителя множества
3. На основе **«деревьев»**, корень которых является представителем множества

НА ОСНОВЕ МАССИВА

	0	1	2	3	4	5	6	7	8	9
`int *dsu`	0	1	2	3	4	5	6	7	8	9

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

НА ОСНОВЕ МАССИВА

	0	1	2	3	4	5	6	7	8	9
int *dsu	0	1	2	3	4	5	6	7	8	9

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

```
union(0, 2) ->
union(0, 4) ->
union(7, 9) ->
union(7, 2) ->
```

НА ОСНОВЕ МАССИВА

	0	1	2	3	4	5	6	7	8	9
int *dsu	0	1	0	3	4	5	6	7	8	9

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

```
union(0, 2) -> dsu[2] = dsu[0]
union(0, 4) ->
union(7, 9) ->
union(7, 2) ->
```

НА ОСНОВЕ МАССИВА

	0	1	2	3	4	5	6	7	8	9
int *dsu	0	1	0	3	0	5	6	7	8	9

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

```
union(0, 2) -> dsu[2] = dsu[0]
union(0, 4) -> dsu[4] = dsu[0]
union(7, 9) ->
union(7, 2) ->
```

НА ОСНОВЕ МАССИВА

	0	1	2	3	4	5	6	7	8	9
int *dsu	0	1	0	3	0	5	6	7	8	7

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

```
union(0, 2) -> dsu[2] = dsu[0]
union(0, 4) -> dsu[4] = dsu[0]
union(7, 9) -> dsu[9] = dsu[7]
union(7, 2) ->
```

НА ОСНОВЕ МАССИВА

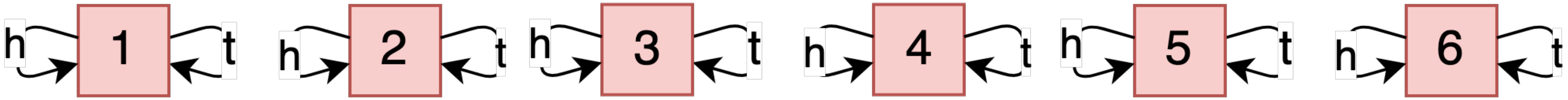
	0	1	2	3	4	5	6	7	8	9
int *dsu	0	1	0	3	0	5	6	7	8	7

Исходное состояние – синглтоны, которые являются самостоятельными представителями множеств

```
union(0, 2) -> dsu[2] = dsu[0]
union(0, 4) -> dsu[4] = dsu[0]
union(7, 9) -> dsu[9] = dsu[7]
union(7, 2) -> dsu[0] = dsu[7]; dsu[2] = dsu[7]; dsu[4] = dsu[7];
```

Выполнение **`union(x, y)`** потребует в худшем случае **`O(n)`** операций перезаписи представителей множеств

НА ОСНОВЕ СПИСКА

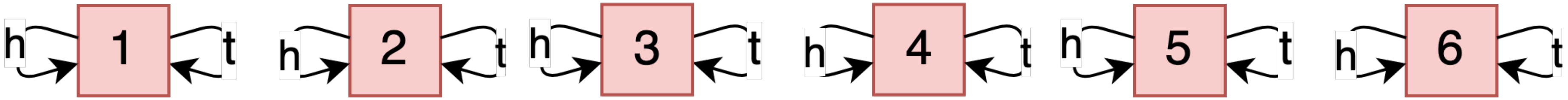


Исходное состояние – **зацикленные** списки из 1 элемента

Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```

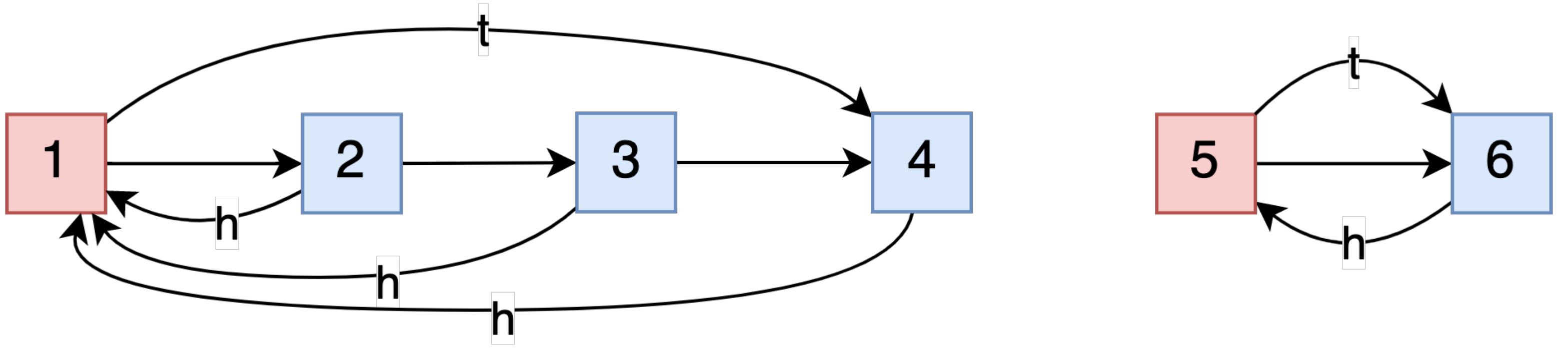
НА ОСНОВЕ СПИСКА



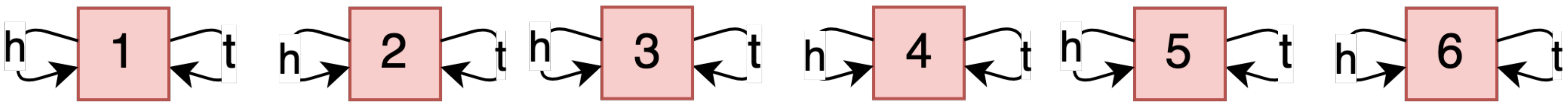
Исходное состояние – **зацикленные** списки из 1 элемента

Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```



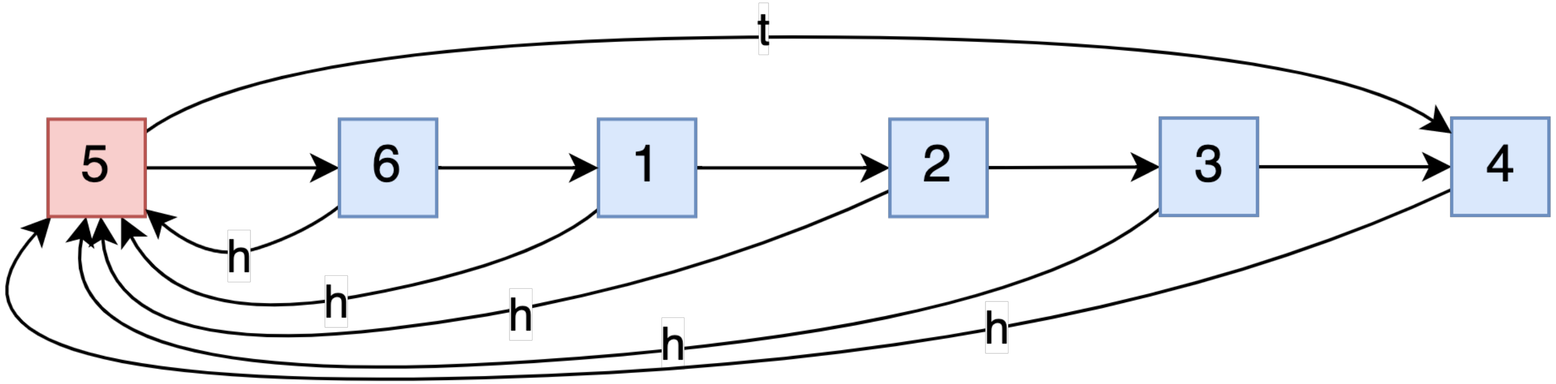
НА ОСНОВЕ СПИСКА



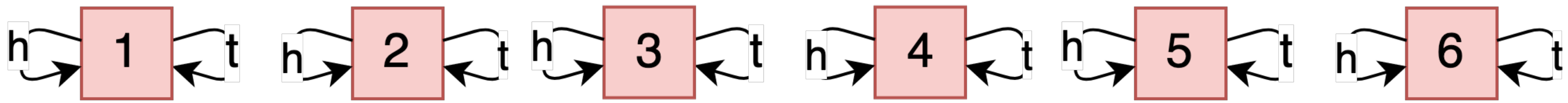
Исходное состояние – **зацикленные** списки из 1 элемента

Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```



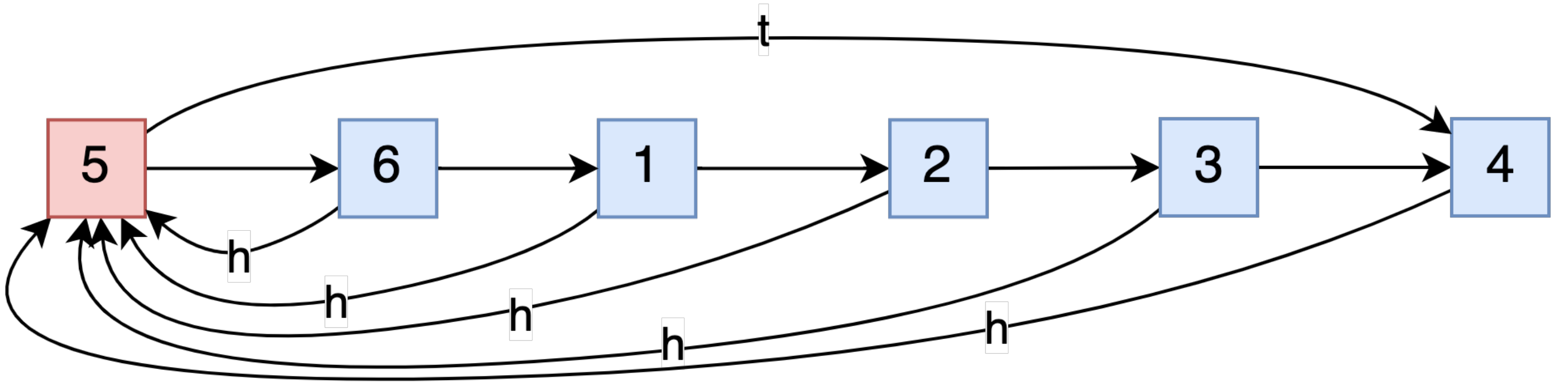
НА ОСНОВЕ СПИСКА



Исходное состояние – **зацикленные** списки из 1 элемента

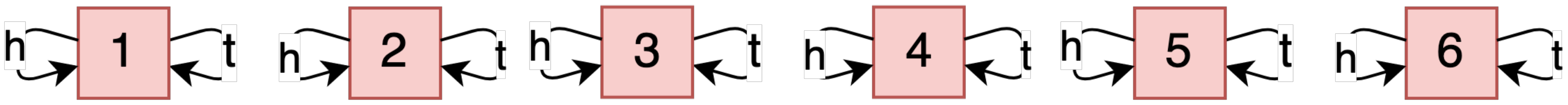
Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```



Выполнение **`union(x, y)`** потребует в худшем случае **O(n)** операций перезаписи указателя на голову (представителя множества)

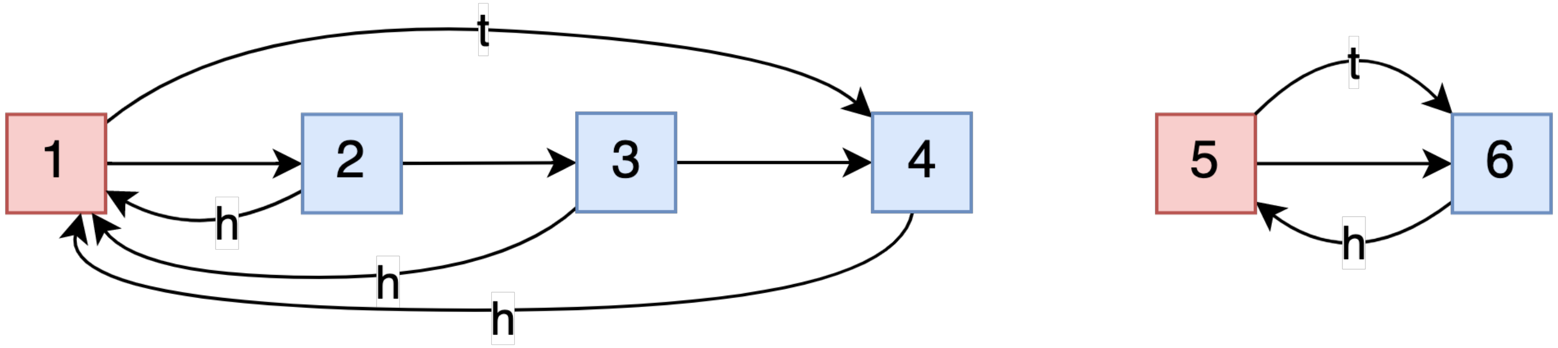
НА ОСНОВЕ СПИСКА



Исходное состояние – **зацикленные** списки из 1 элемента

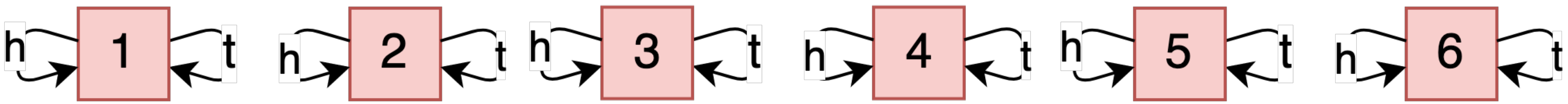
Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```



При объединении будем множество меньшего размера добавлять к множеству большего размера

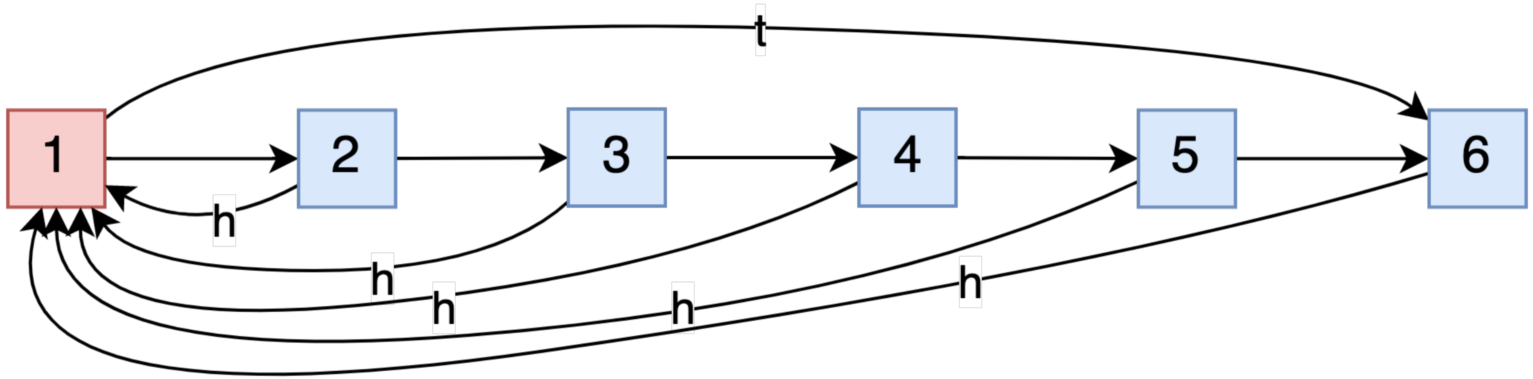
НА ОСНОВЕ СПИСКА



Исходное состояние – **зацикленные** списки из 1 элемента

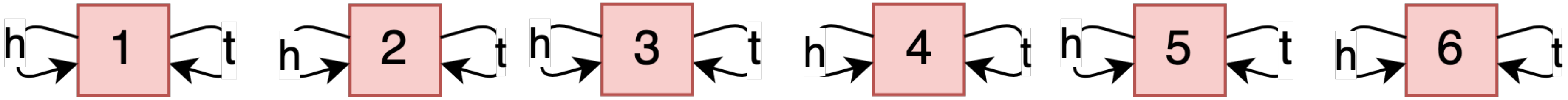
Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```



При объединении будем множество меньшего размера добавлять к множеству большего размера

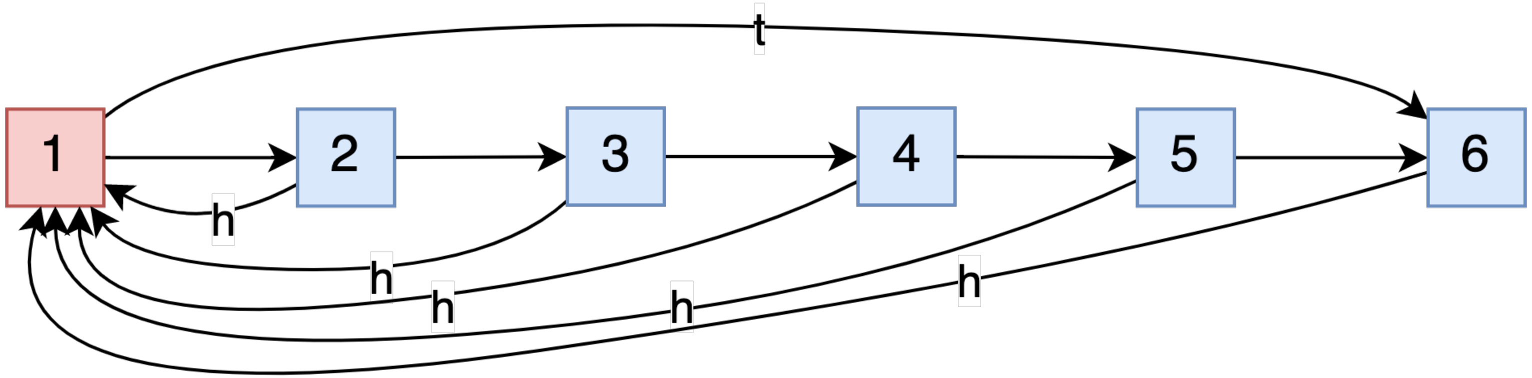
НА ОСНОВЕ СПИСКА



Исходное состояние – **зацикленные** списки из 1 элемента

Голова списка – **представитель** множества. Дополнительно хранится указатель на хвост списка для выполнения объединения

```
union(1, 2)
union(1, 3)
union(3, 4)
union(5, 6)
union(6, 1)
```

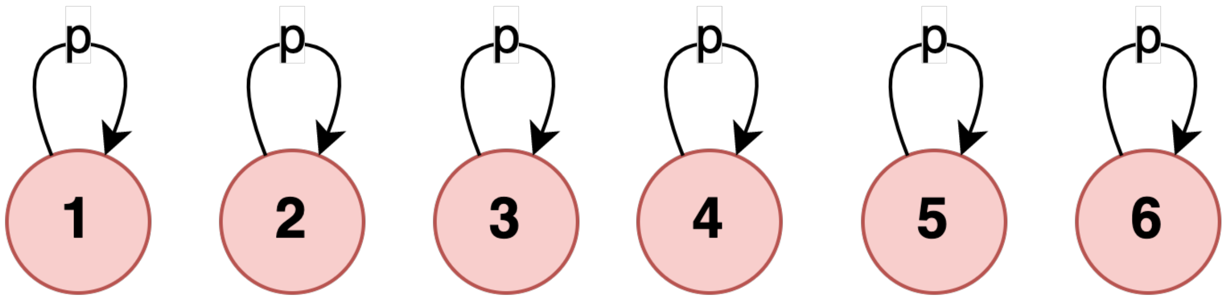


Количество необходимых операций перезаписи указателя на голову снижается до `O(log n)`

НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

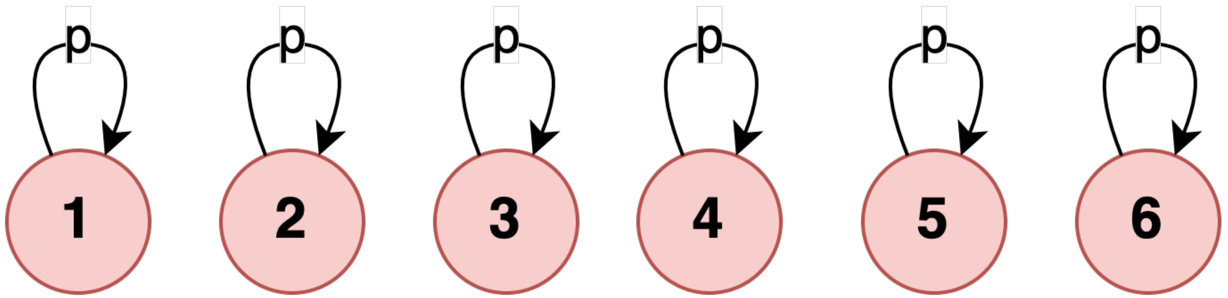
Указатель p на «родителя»



НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»



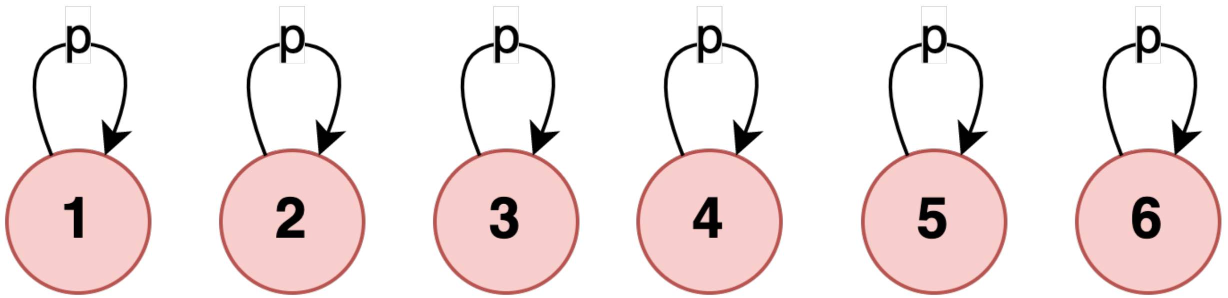

```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```

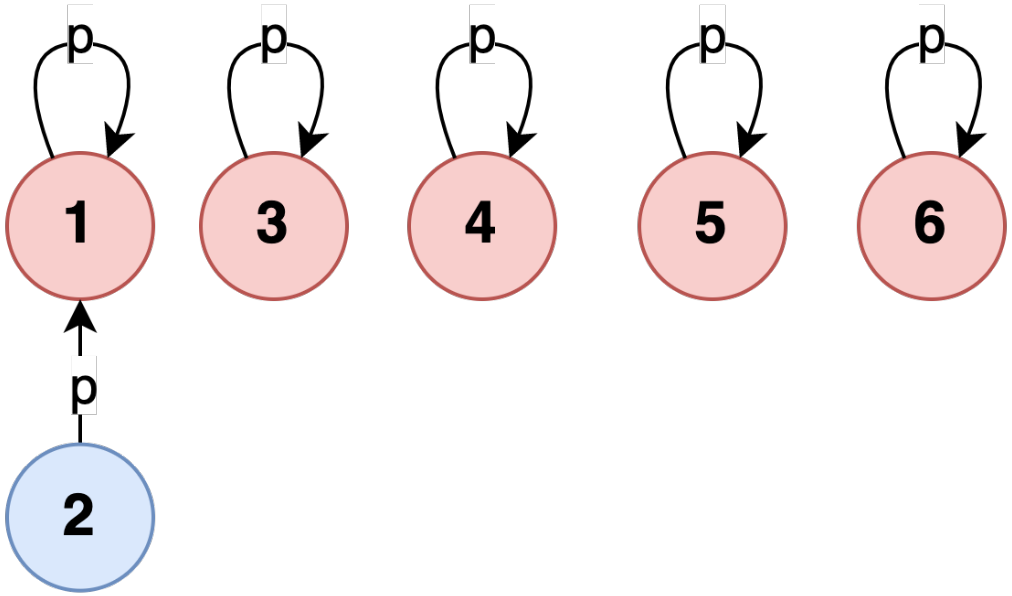
НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```



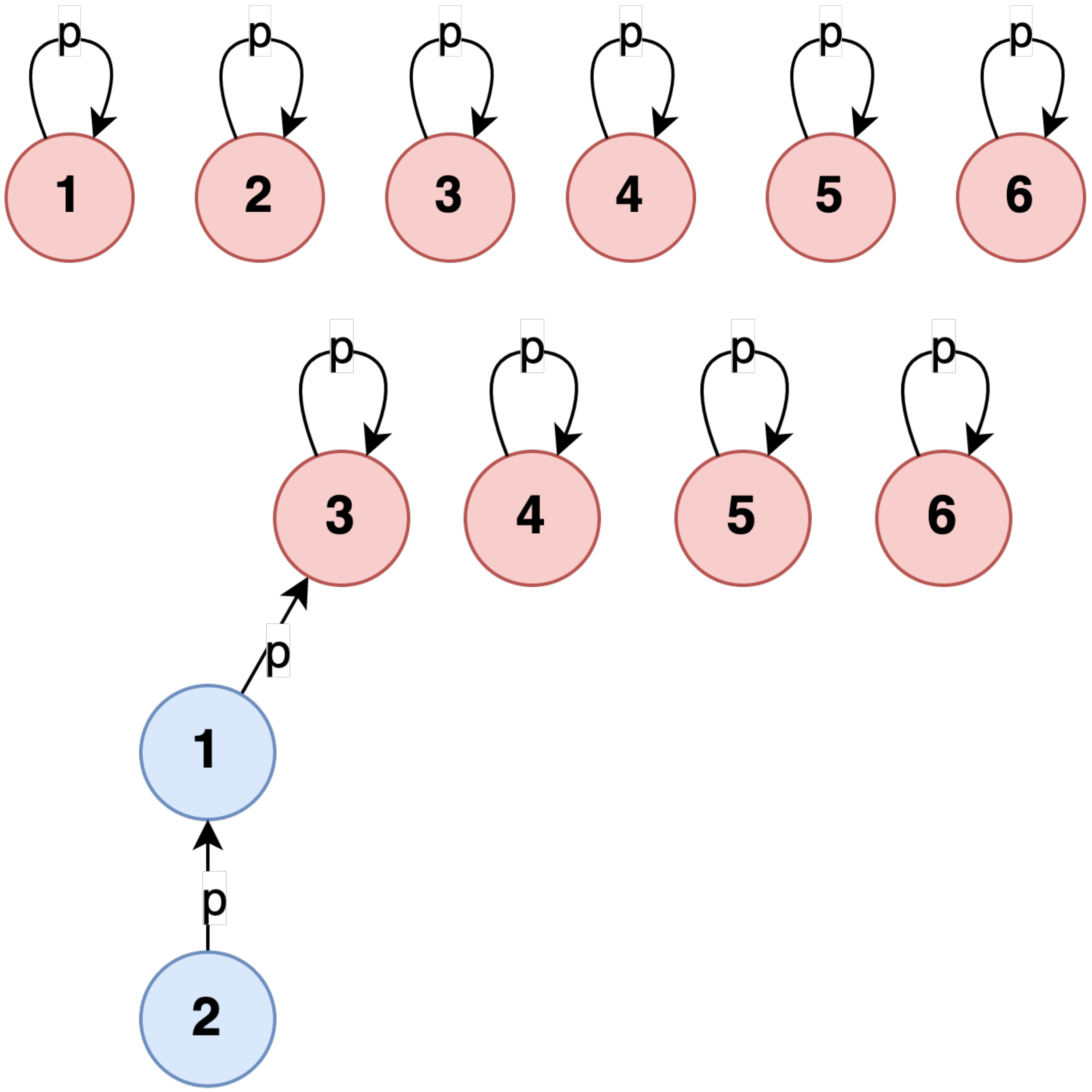


НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```

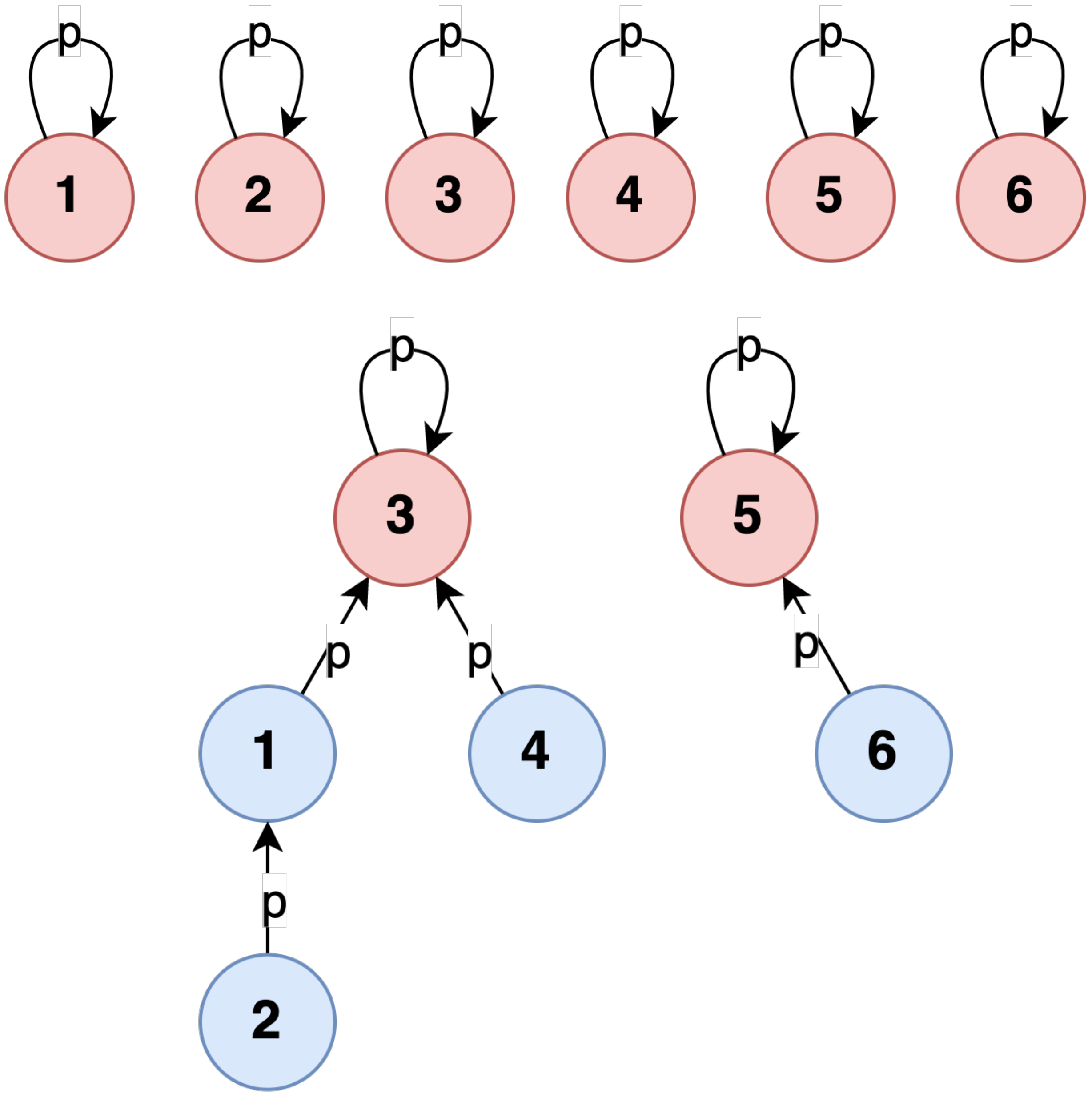


НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

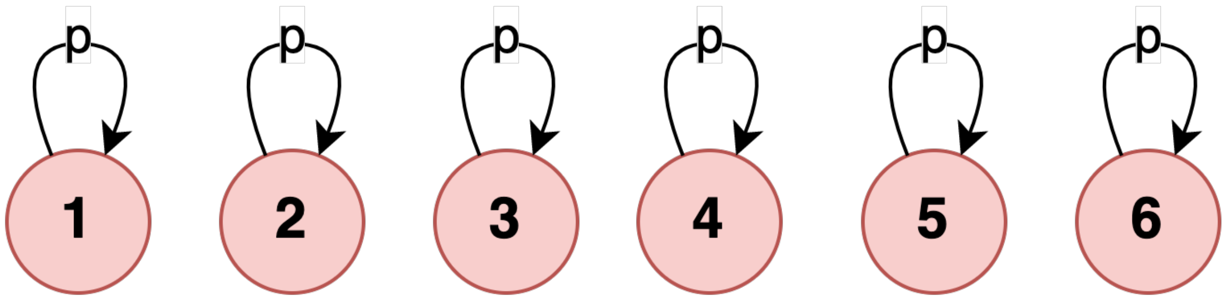
```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```



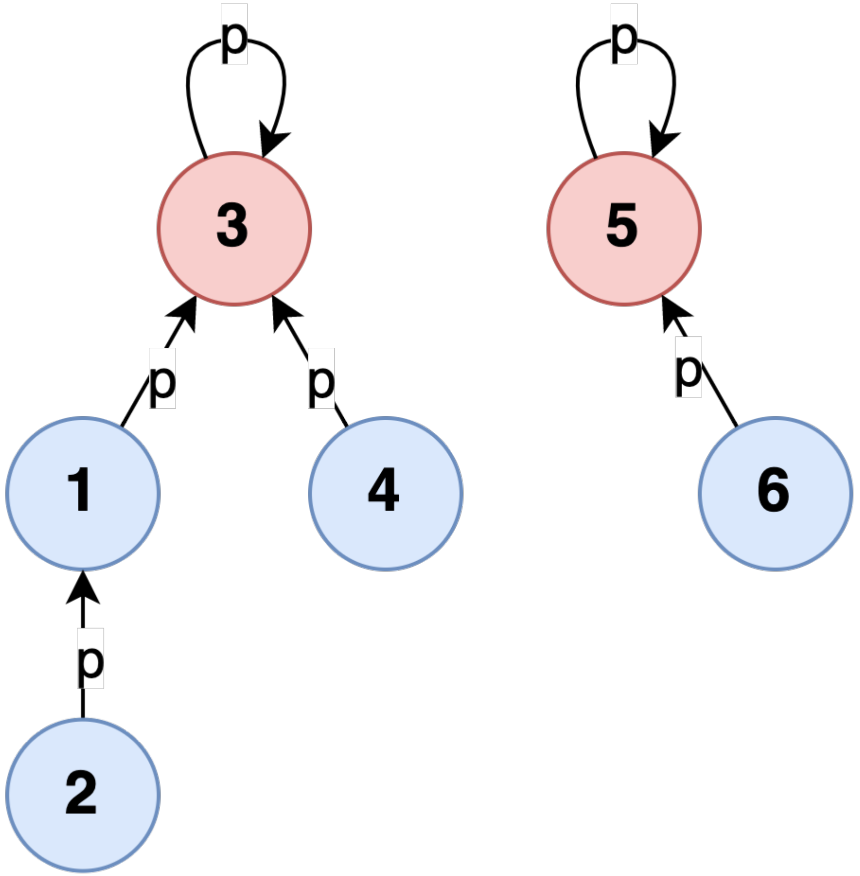
НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»




```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```



Внутреннее хранение — в **массиве** номеров объектов-предков

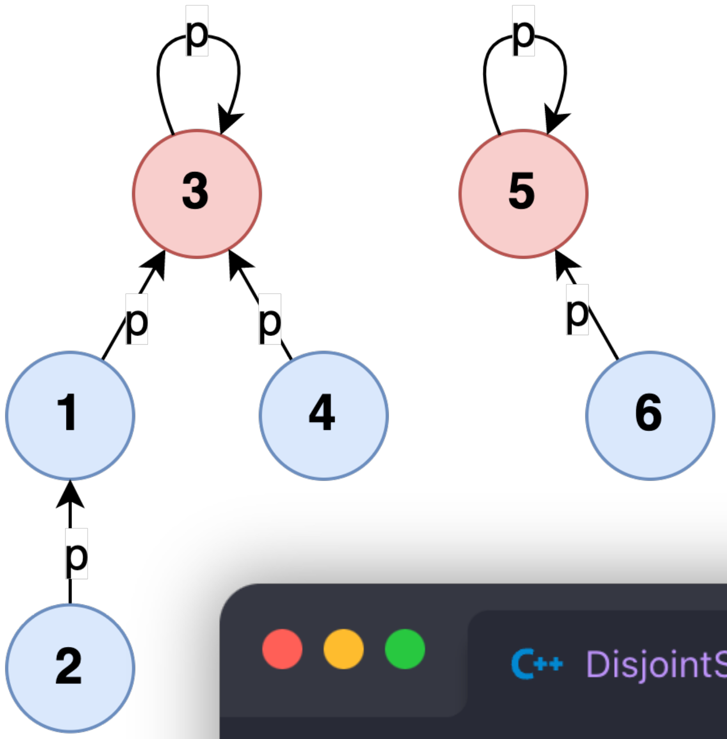
	1	2	3	4	5	6
`int *p`	3	1	3	3	5	5

НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```



Внутреннее хранение — в **массиве** номеров объектов-предков

	1	2	3	4	5	6
`int *p`	3	1	3	3	5	5

DisjointSetUnion

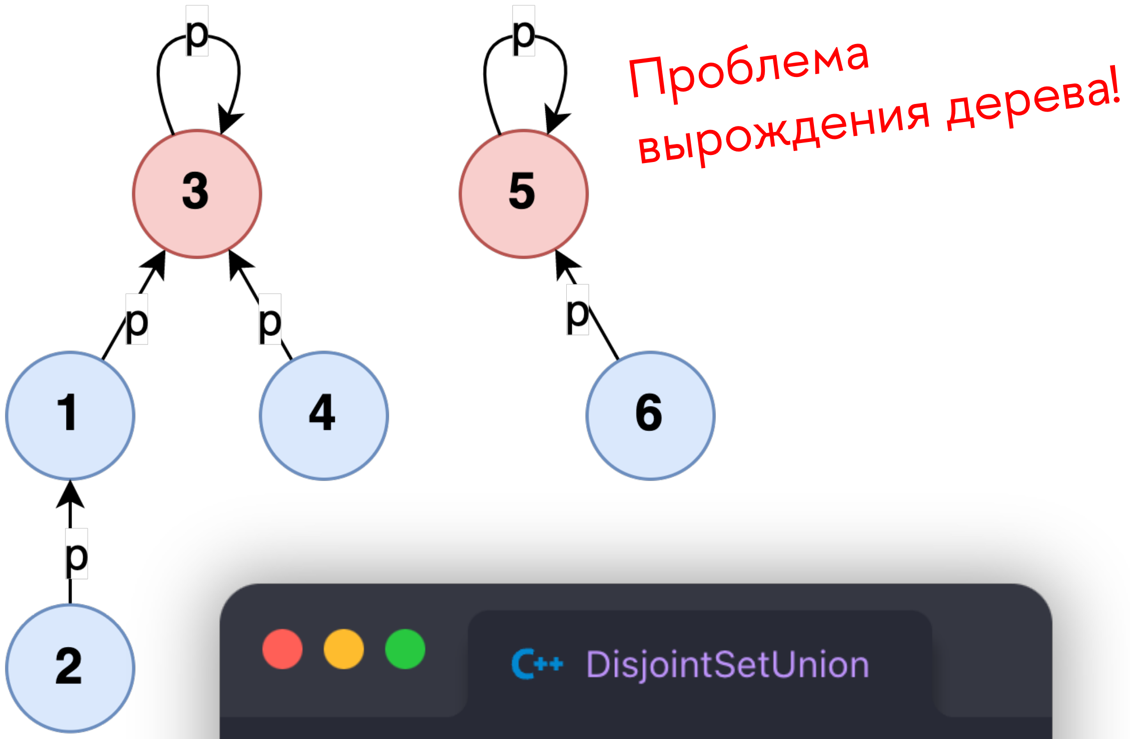
```cpp
int find(int key) {
    if (p[key] == key)
        return key;
    else
        return find(p[key])
}
```

НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

```
union(1, 2)
union(3, 2)
union(2, 4)
union(5, 6)
```



Внутреннее хранение — в **массиве** номеров объектов-предков

	1	2	3	4	5	6
int *p	3	1	3	3	5	5

DisjointSetUnion

```cpp
int find(int key) {
    if (p[key] == key)
        return key;
    else
        return find(p[key])
}
```

НА ОСНОВЕ «ДЕРЕВЬЕВ»

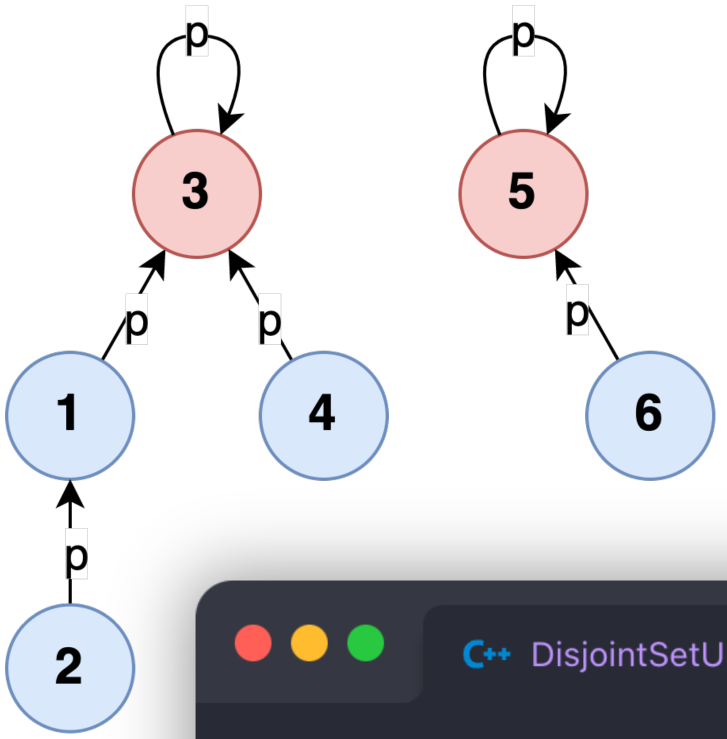
Корнем дерева является **представитель** множества

Указатель p на «родителя»

Проблема вырожденного дерева решается путем **оптимизации** операции `find` — объект сразу «привешивается» к корню

Внутреннее хранение — в **массиве** номеров объектов-предков

	1	2	3	4	5	6
int *p	3	1	3	3	5	5



C++ DisjointSetUnion

```
int findOptimized(int key) {
    if (p[key] == key)
        return key;
    else
        return
          p[key] = find(p[key]);
}
```

НА ОСНОВЕ «ДЕРЕВЬЕВ»

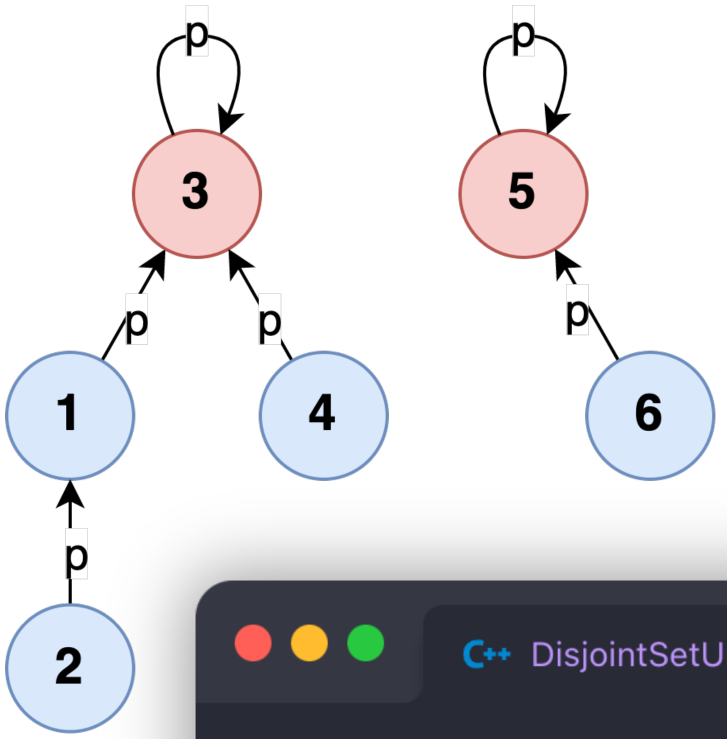
Корнем дерева является **представитель** множества

Указатель p на «родителя»

Проблема вырожденного дерева решается путем **оптимизации** операции `find` — объект сразу «привешивается» к корню

Внутреннее хранение — в **массиве** номеров объектов-предков

	1	2	3	4	5	6
`int *p`	3	1	3	3	5	5



`find(2)`

DisjointSetUnion

```cpp
int findOptimized(int key) {
    if (p[key] == key)
        return key;
    else
        return
          p[key] = find(p[key]);
}
```

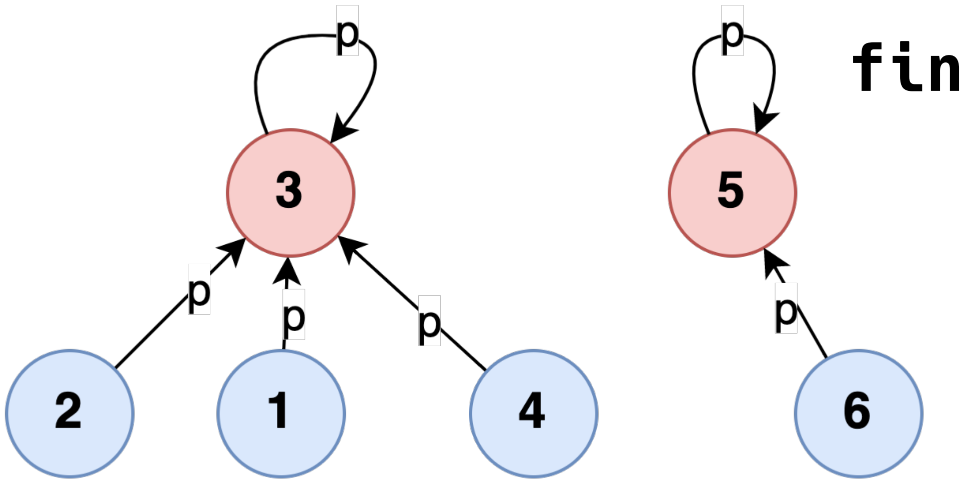
НА ОСНОВЕ «ДЕРЕВЬЕВ»

Корнем дерева является **представитель** множества

Указатель p на «родителя»

Проблема вырожденного дерева решается путем **оптимизации** операции `find` — объект сразу «привешивается» к корню

Внутреннее хранение — в **массиве** номеров объектов-предков



	1	2	3	4	5	6
`int *p`	3	3	3	3	5	5

DisjointSetUnion

```
int findOptimized(int key) {
    if (p[key] == key)
        return key;
    else
        return
          p[key] = find(p[key]);
}
```

НА ОСНОВЕ «ДЕРЕВЬЕВ»

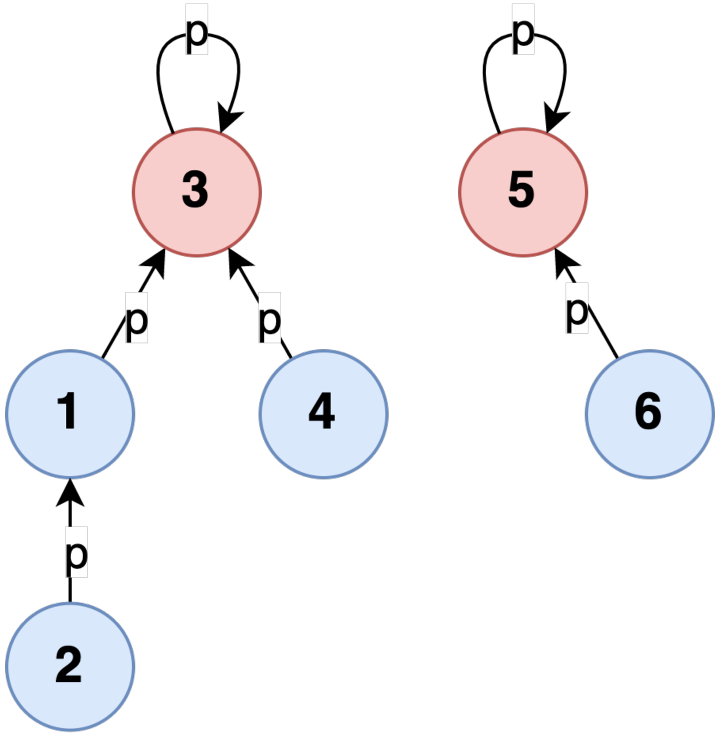
Корнем дерева является **представитель** множества

Указатель p на «родителя»

Проблема вырожденного дерева решается путем **оптимизации** операции `union` — «легкое» множество к «тяжелому»

Внутреннее хранение — в **массиве** номеров объектов-предков и **размеров множеств**

	1	2	3	4	5	6
`int *p`	3	1	3	3	5	5
`int *s`	2	1	4	1	2	1



`union(6,1)`

НА ОСНОВЕ «ДЕРЕВЬЕВ»

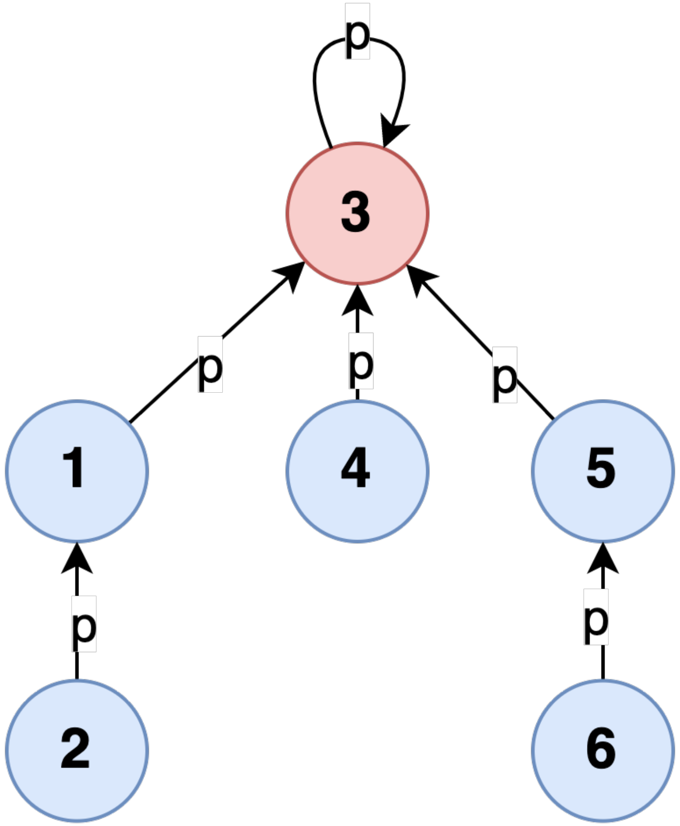
Корнем дерева является **представитель** множества

Указатель p на «родителя»

Проблема вырожденного дерева решается путем **оптимизации** операции `union` — «легкое» множество к «тяжелому»

Внутреннее хранение — в **массиве** номеров объектов-предков и **размеров множеств**

	1	2	3	4	5	6
`int *p`	3	1	3	3	3	5
`int *s`	2	1	6	1	2	1



`union(6,1)`

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**

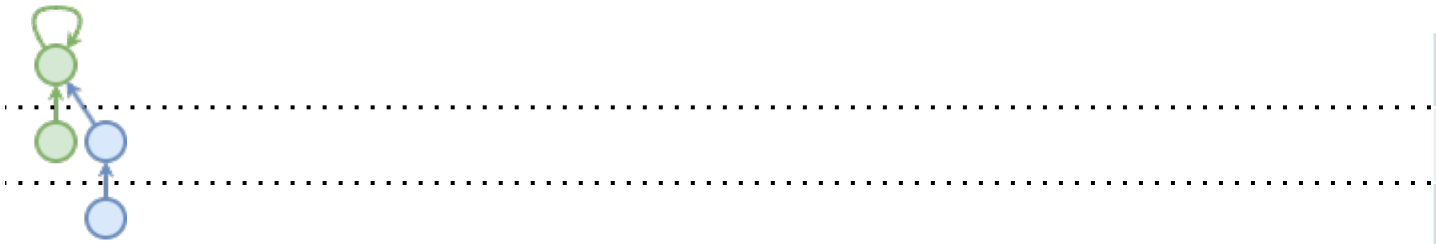
0	1
1	1

Шаг 1 Объединение двух деревьев-множеств нулевой высоты

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



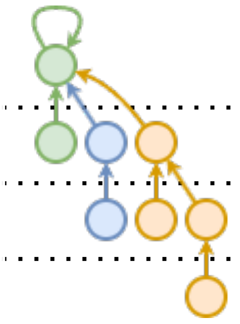
0	1
1	2
2	1

Шаг 2 Объединение двух деревьев-множеств единичной высоты

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



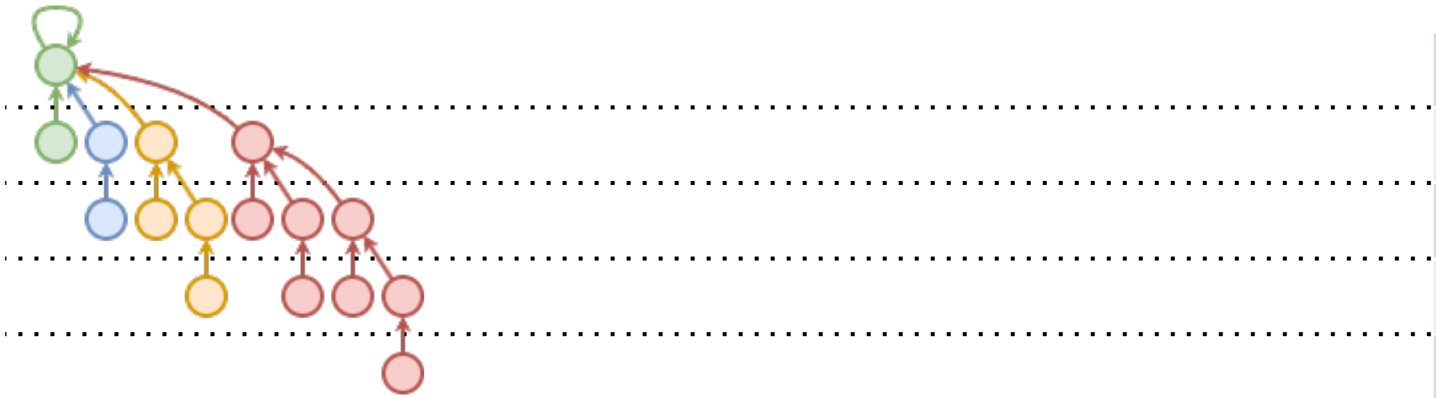
0	1
1	3
2	3
3	1

Шаг 3 Объединение двух деревьев-множеств высоты 2

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



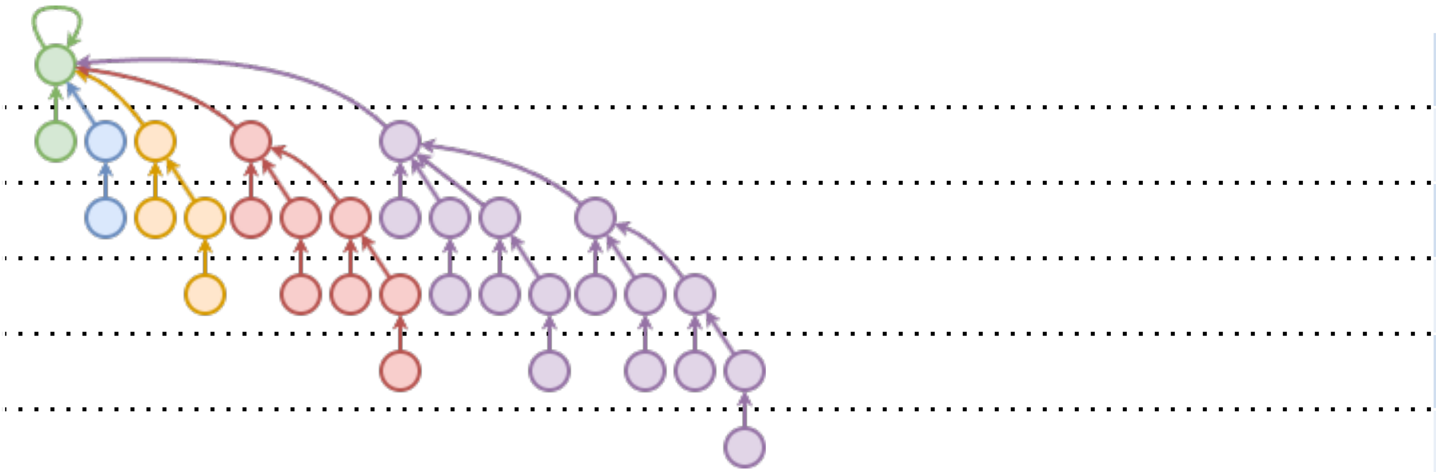
0	1
1	4
2	6
3	4
4	1

Шаг 4 Объединение двух деревьев-множеств высоты 3

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



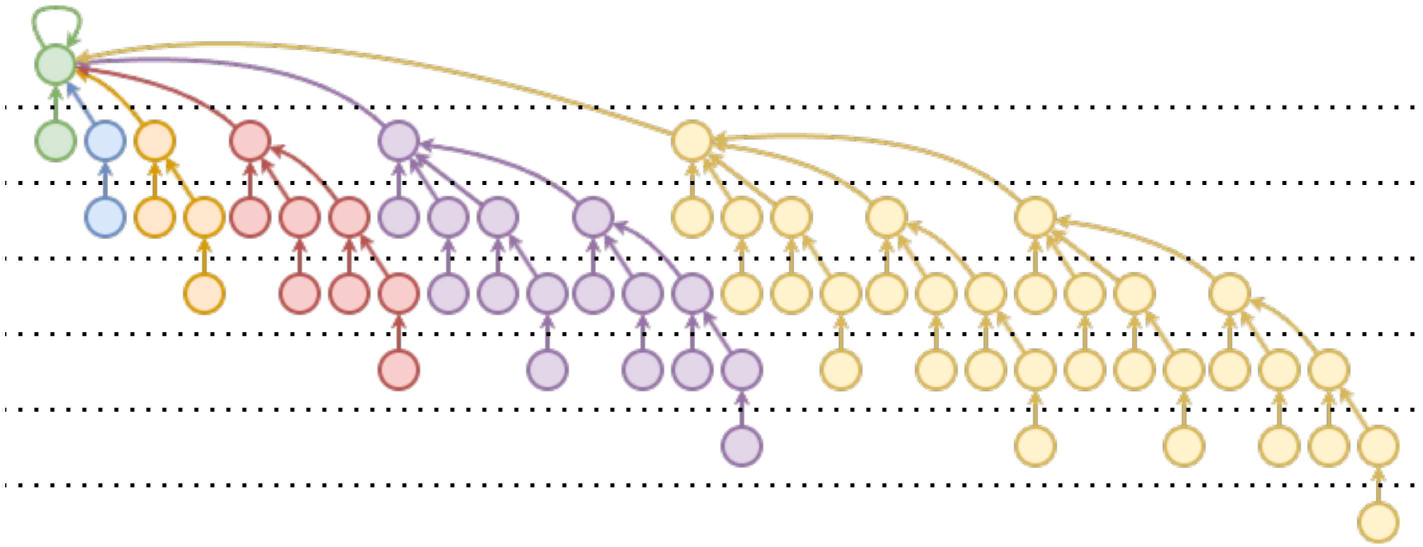
0	1
1	5
2	10
3	10
4	5
5	1

Шаг 5 Объединение двух деревьев-множеств высоты 4

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



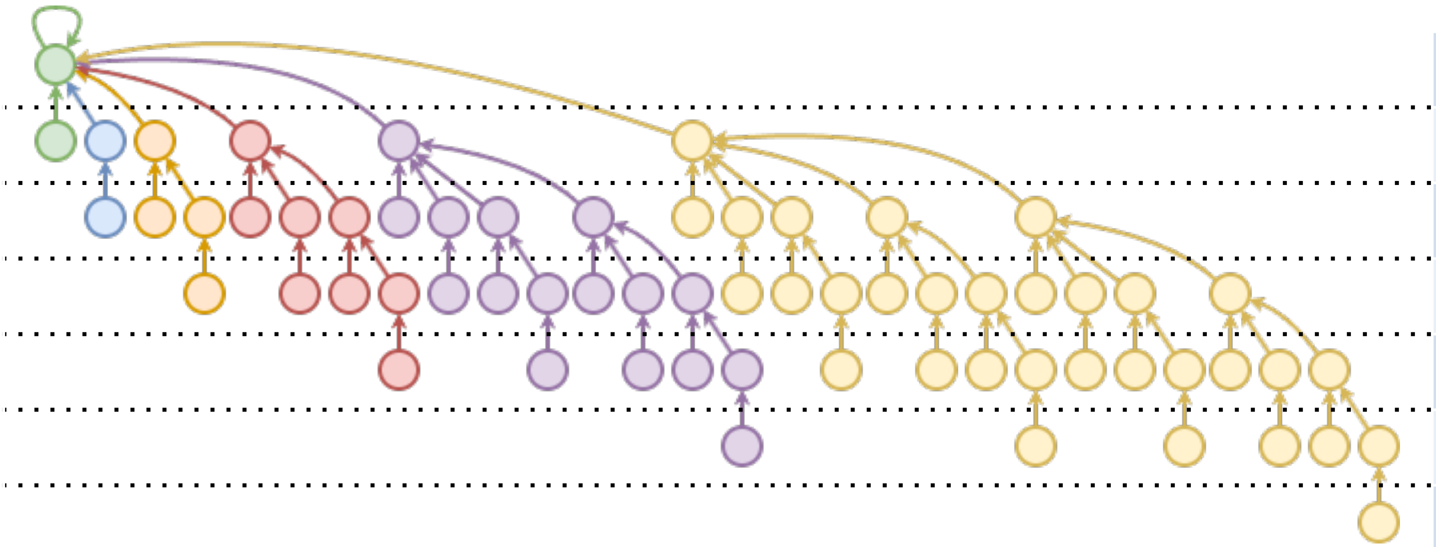
0	1
1	6
2	15
3	20
4	15
5	6
6	1

Шаг 6 Объединение двух деревьев-множеств высоты 5, и так далее…

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



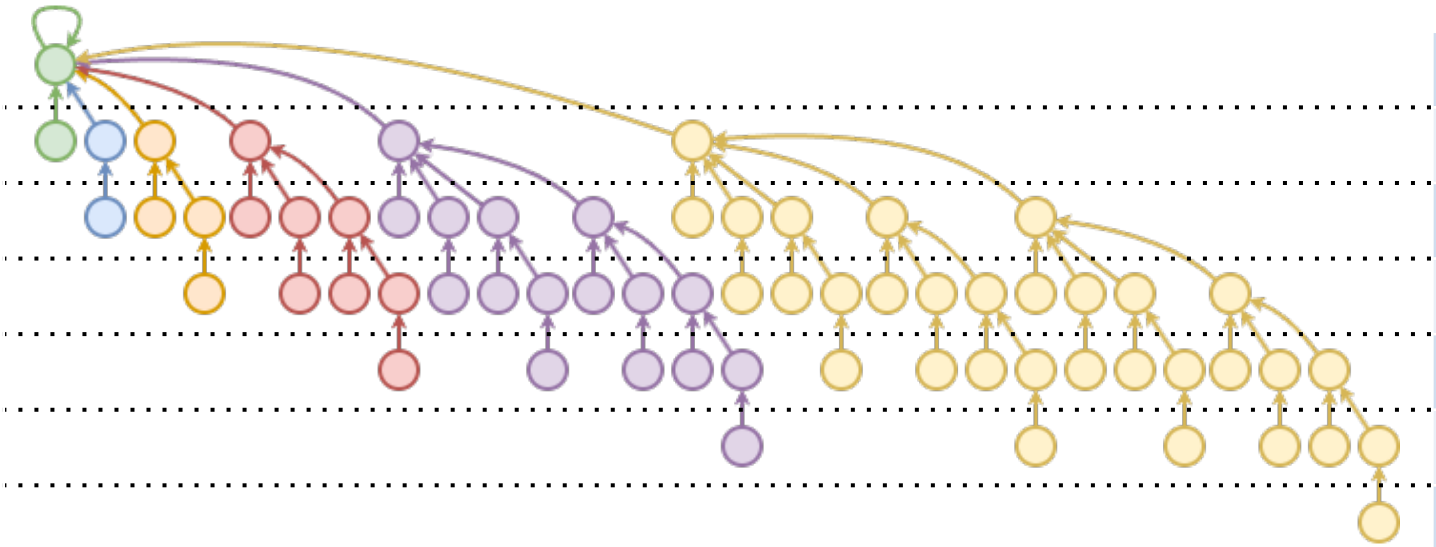
0	1
1	6
2	15
3	20
4	15
5	6
6	1

Получившиеся деревья называются **биномиальными**, т.к. на каждом уровне находится C_h^k, где h — высота дерева, а k — номер уровня

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ХУДШИЙ СЛУЧАЙ

Поскольку всегда «легкое» дерево привешивается к «тяжелому», то худший случай возникает, когда **деревья-множества имеют одинаковую высоту**



0	1
1	6
2	15
3	20
4	15
5	6
6	1

Оценим высоту дерева (n — общее количество объектов):

$$\sum_{k=0}^{h} C_h^k = 2^h = n \Rightarrow h = \log n$$

НА ОСНОВЕ «ДЕРЕВЬЕВ»

ЛУЧШИЙ СЛУЧАЙ

Лучший случай возникает, когда родитель всех объектов один и тот же, — **единственное дерево высоты $\Theta(1)$**

	0	1	2	3	4	5	6	7	8	9
`int *p`	0	0	0	0	0	0	0	0	0	0

Сложности основных операций `find(x)` и `union(x, y)` также ограничиваются высотой получаемого дерева-множества

НА ОСНОВЕ «ДЕРЕВЬЕВ»

СРЕДНИЙ СЛУЧАЙ

Асимптотическое поведение `UNION-FIND` в среднем случае ограничивается $O(\alpha(n))$, где $\alpha(n)$ — функция, **обратная к функции Аккермана $A(n,m)$**:

$$A(n,m) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \text{ и } n = 0 \\ A(m-1, A(m, n-1), & m > 0 \text{ и } n > 0 \end{cases}$$

`A(0,0) =` **`1`**`, A(1,1) =` **`3`**`, A(2,2) =` **`7`**`, A(3,3) =` **`61`**`, A(4,4) =` **$2^{A(3,4)} - 3$**

НА ОСНОВЕ «ДЕРЕВЬЕВ»

СРЕДНИЙ СЛУЧАЙ

Асимптотическое поведение UNION-FIND в среднем случае ограничивается $O(\alpha(n))$, где $\alpha(n)$ — функция, **обратная к функции Аккермана $A(n,m)$**:

$$A(n,m) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \text{ и } n = 0 \\ A(m-1, A(m,n-1)), & m > 0 \text{ и } n > 0 \end{cases}$$

A(0,0) = **1**, A(1,1) = **3**, A(2,2) = **7**, A(3,3) = **61**, A(4,4) = **$2^{A(3,4)}$ - 3**

A(3,4) = **2003529930406846464979072351...**

НА ОСНОВЕ «ДЕРЕВЬЕВ»

СРЕДНИЙ СЛУЧАЙ

Асимптотическое поведение UNION-FIND в среднем случае ограничивается $O(\alpha(n))$, где $\alpha(n)$ — функция, **обратная к функции Аккермана $A(n,m)$**:

$$A(n,m) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \text{ и } n = 0 \\ A(m-1, A(m, n-1)), & m > 0 \text{ и } n > 0 \end{cases}$$

`A(0,0) =` **`1`**`, A(1,1) =` **`3`**`, A(2,2) =` **`7`**`, A(3,3) =` **`61`**`, A(4,4) =` **$2^{A(3,4)} - 3$**

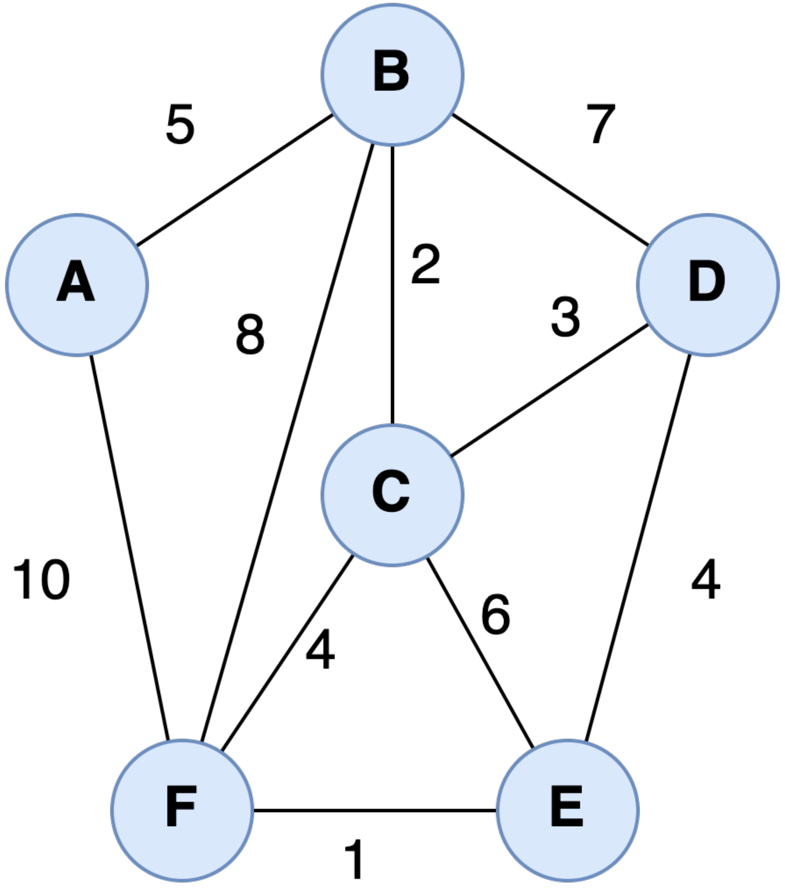
Из *инженерных* соображений, абсолютно правомерно предположить, что средняя высота дерева-множества **вряд ли может быть более четырех**

Применения UNION-FIND

- Анализ **компонент связности** графов
- Поиск **циклов** в графах
- **Сегментация** изображений
- Генерация **лабиринтов**
- Поиск **минимального** остовного дерева
- ...

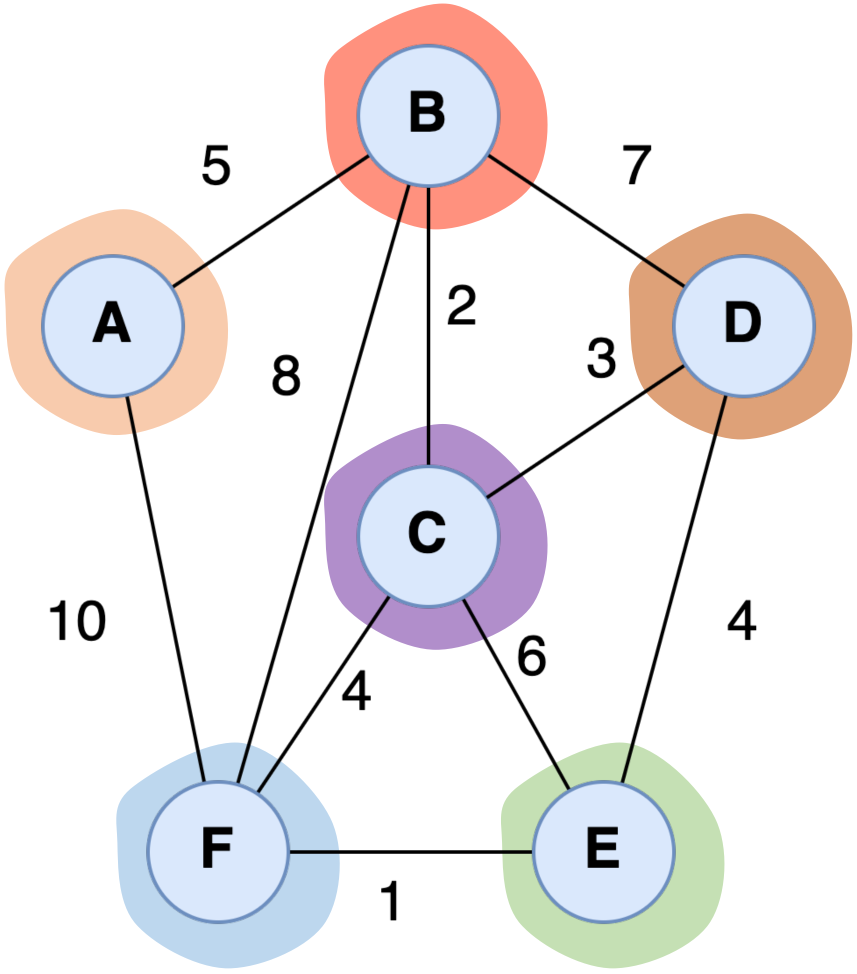
Минимальное остовное дерево **MST**

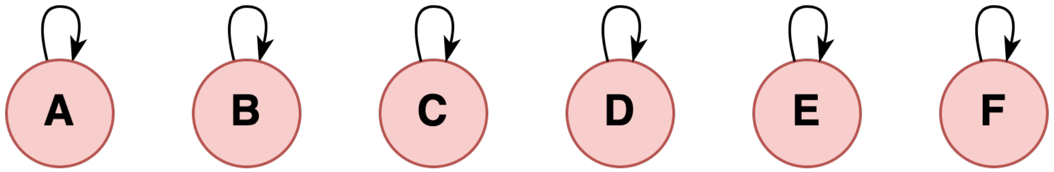
алгоритм Краскала с UNION-FIND



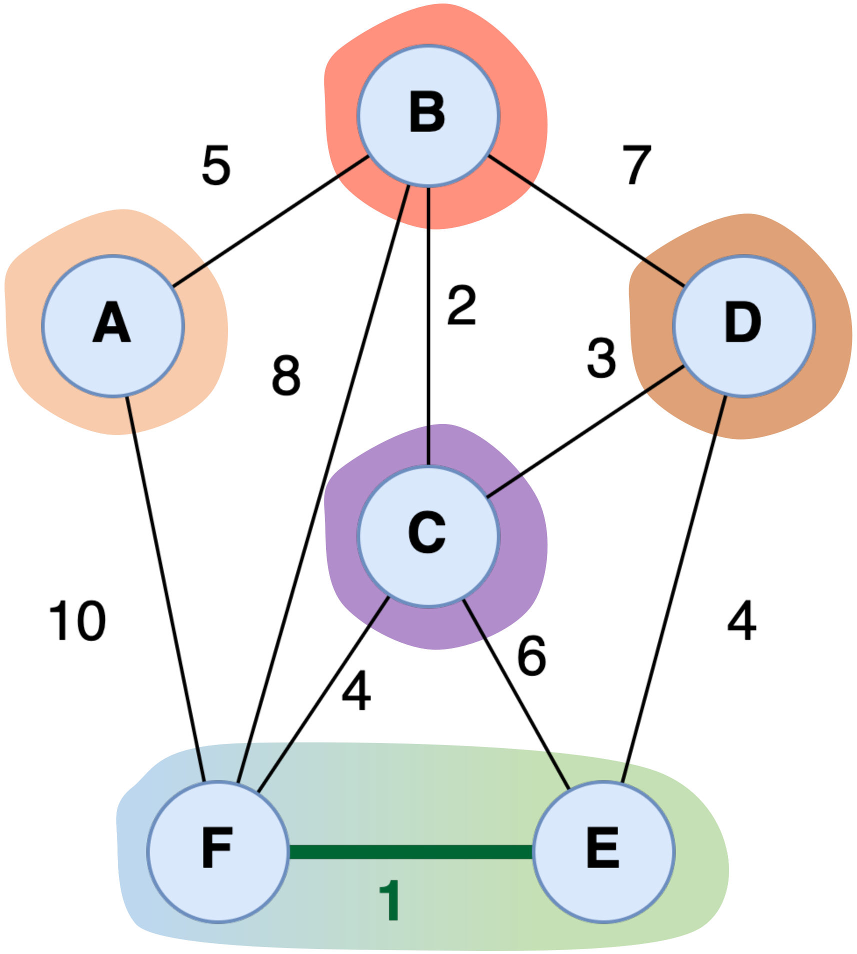
UnionFindKruskal

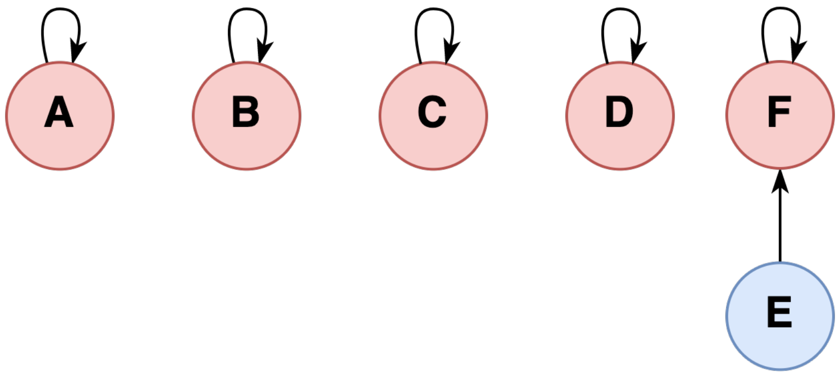
```
G = (V, E) - граф

отсортировать ребра E в порядке неубывания

MST = {}

for (vertex : V)
    makeSet(vertex)

for ((u, v) : sorted_E)
    if (find(u) != find(v))
        добавить (u, v) в MST

    union(u, v)

return MST
```



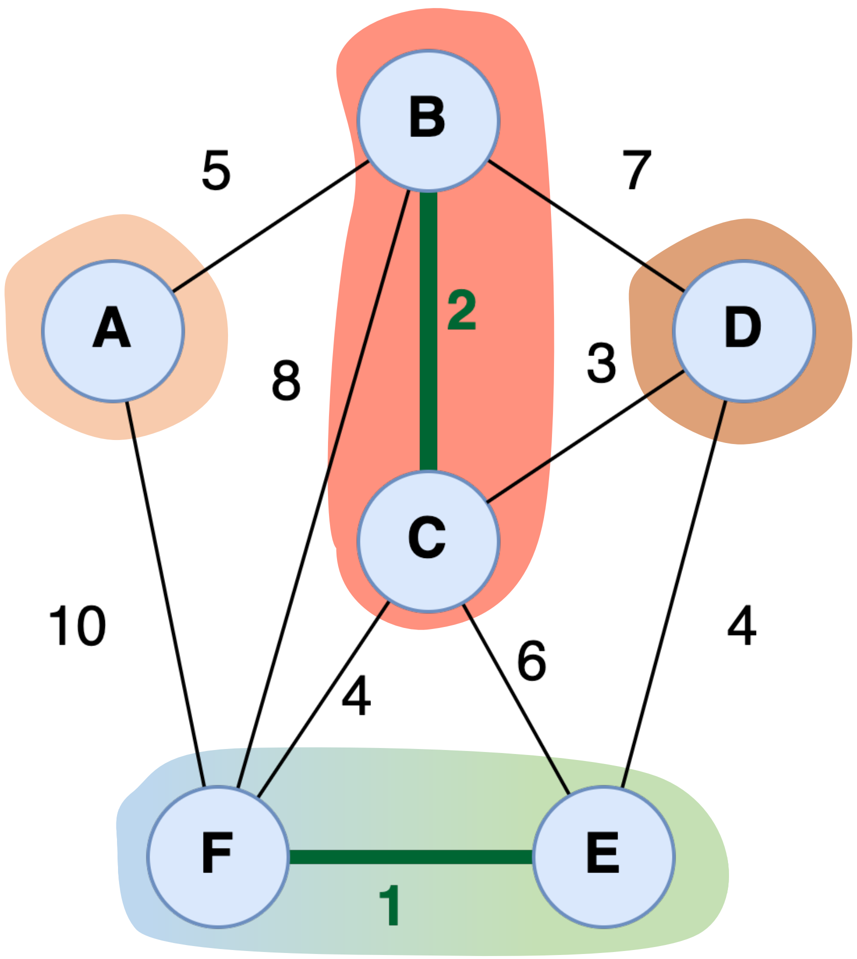


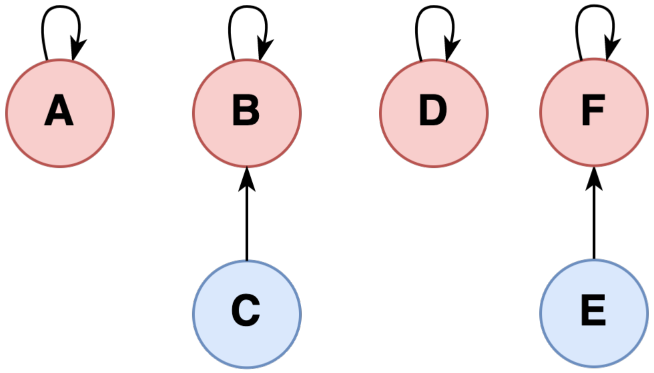

```
find(F) == find(E)
```



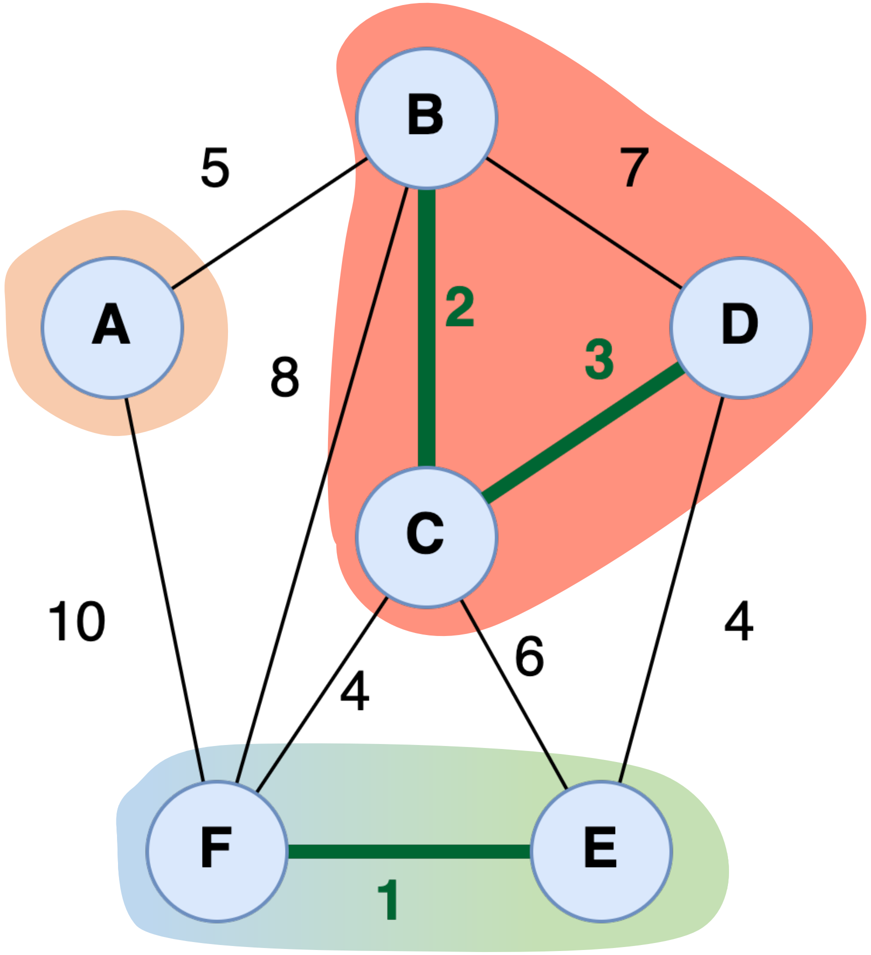


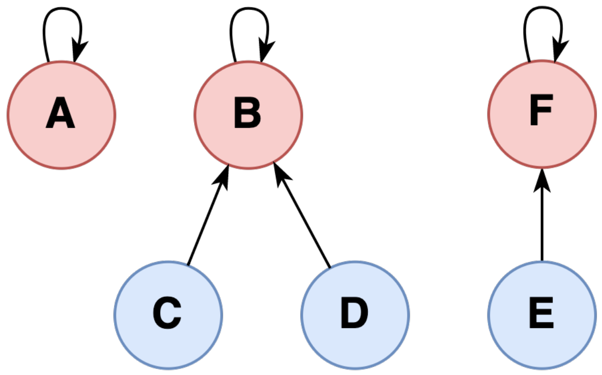
`find(F) == find(E) -> NO, union(F,E)`



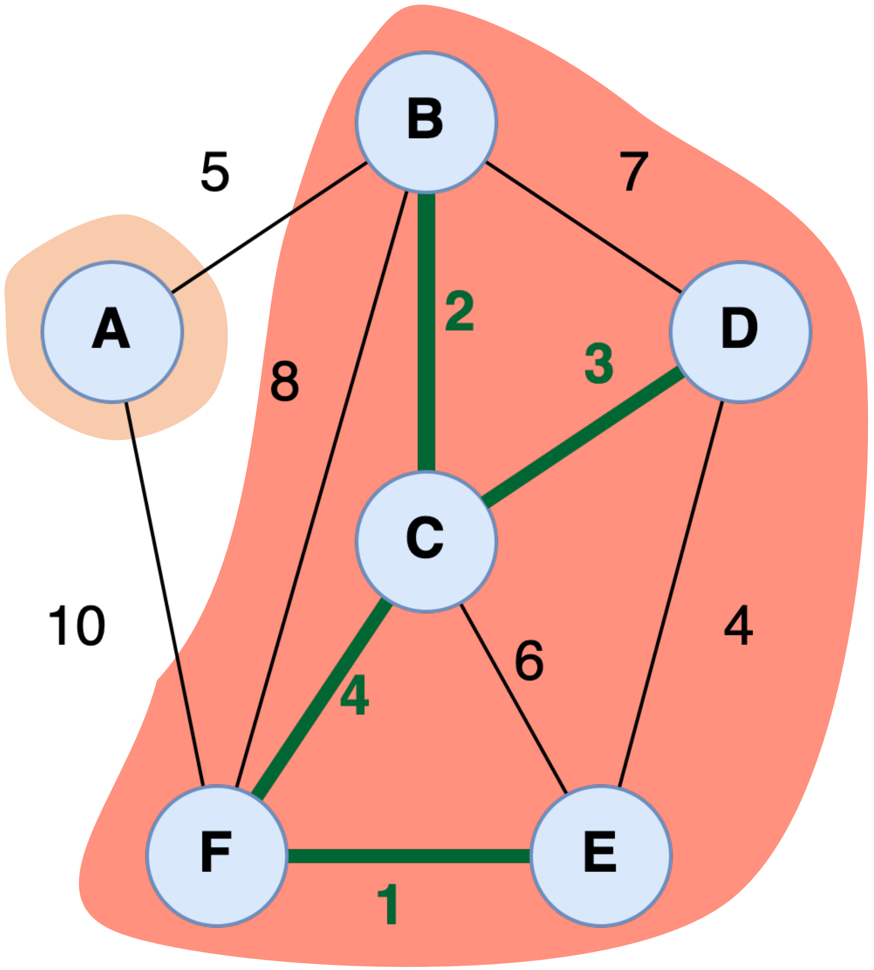


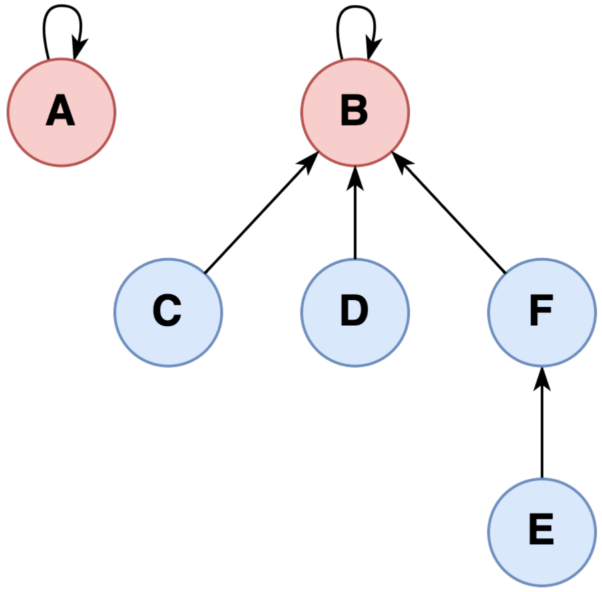

```
find(F) == find(E) -> NO, union(F, E)

find(B) == find(C) -> NO, union(B, C)
```



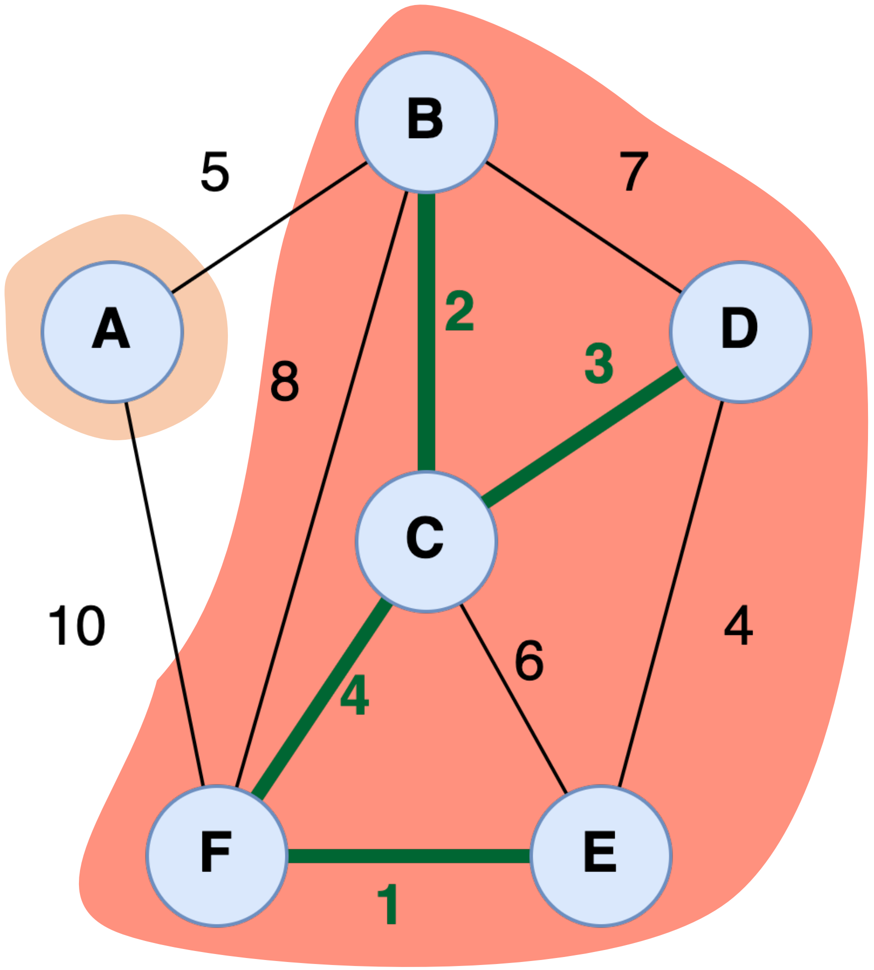


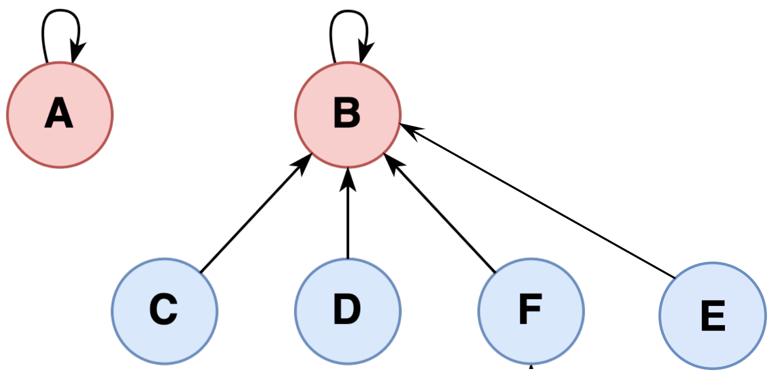

```
find(F) == find(E) -> NO, union(F,E)
find(B) == find(C) -> NO, union(B,C)
find(C) == find(D) -> NO, union(C,D)
```



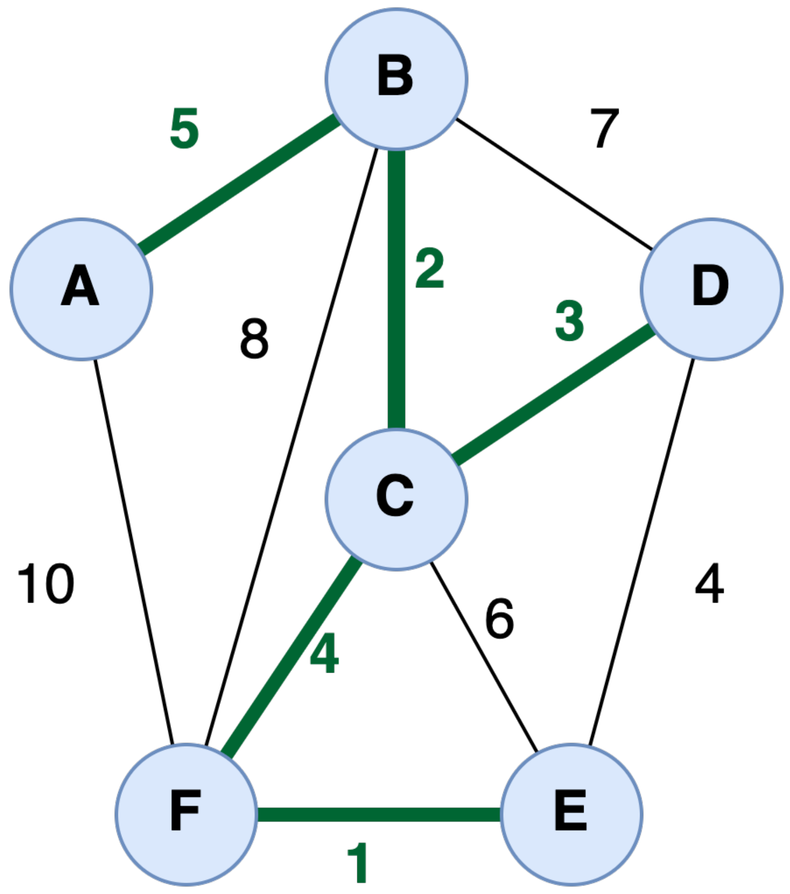



```
find(F) == find(E) -> NO, union(F,E)

find(B) == find(C) -> NO, union(B,C)

find(C) == find(D) -> NO, union(C,D)

find(C) == find(F) -> NO, union(C,F)
```






```
find(F) == find(E) -> NO, union(F,E)

find(B) == find(C) -> NO, union(B,C)

find(C) == find(D) -> NO, union(C,D)

find(C) == find(F) -> NO, union(C,F)

find(D) == find(E) -> YES, SKIP

...
```



Сложность изначальной сортировки ребер доминирует над другими компонентами алгоритма

UnionFindKruskal

```
G = (V, E) - граф

отсортировать ребра E в порядке неубывания
                                        O(E·log E)
MST = {}

for (vertex : V)
    makeSet(vertex)

for ((u, v) : sorted_E)
    if (find(u) != find(v))           O(α(V))
        добавить (u, v) в MST

    union(u, v)                       O(α(V))

return MST
```