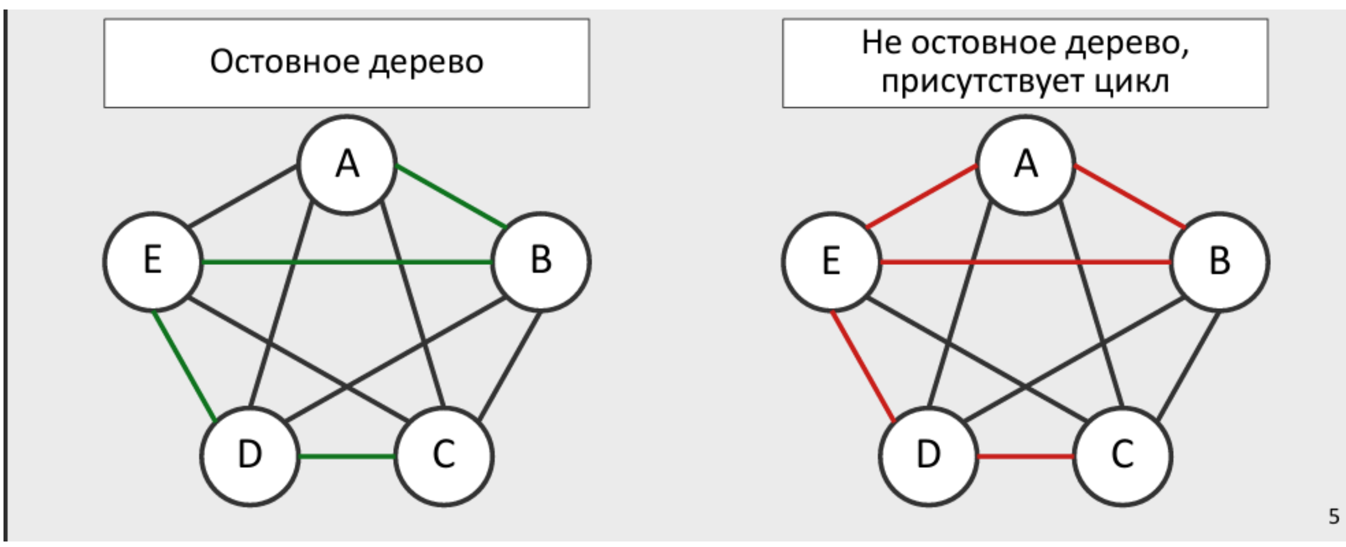
Остовные деревья минимальной стоимости (minimum spanning tree, MST). Алгоритмы построения MST. Система непересекающихся множеств. Алгоритм Крускала. Алгоритм Прима.

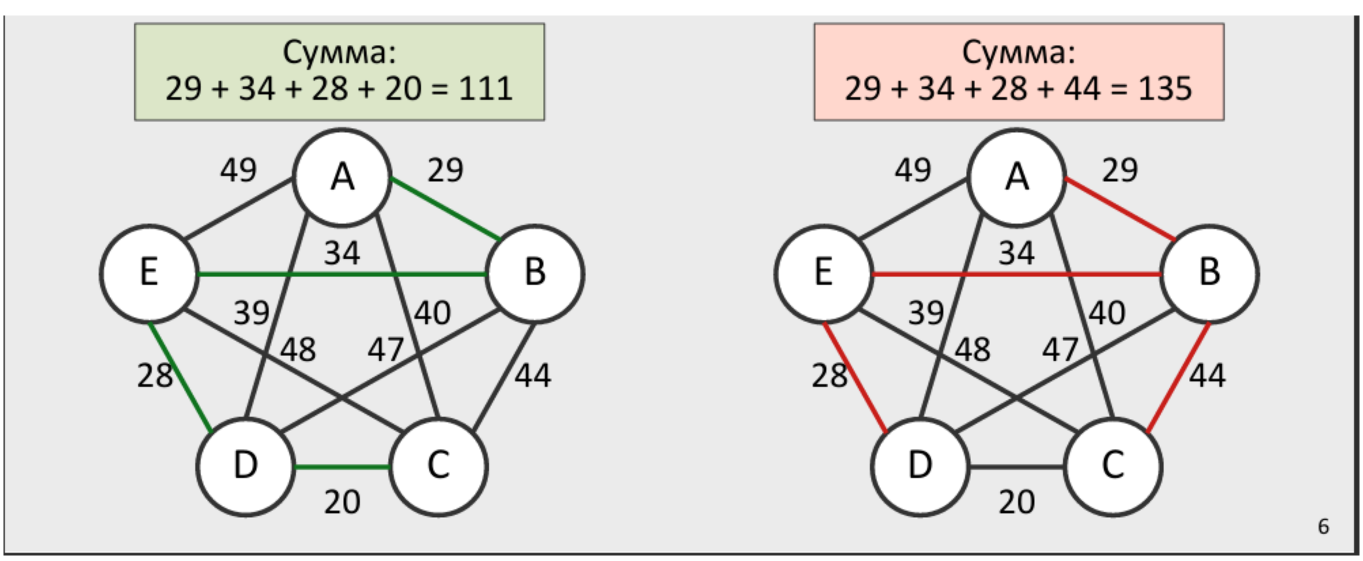
Остовные деревья

- **Остовное дерево** связного графа *(spanning tree)* – это ациклический связный подграф *(дерево)*, в который входят все вершины данного графа
- Синонимы: остов, покрывающее дерево, скелет графа



Остовные деревья минимальной стоимости *(MST)*

- Если граф взвешенный, рассматривается задача о нахождении остовного дерева с минимальной суммой весов входящих в него ребер



- **Остовное дерево минимальной стоимости** (minimum spanning tree, MST) – это остовное дерево с минимальной суммой весов своих ребер
- Практическое применение MST:
  – Формирование дерева для широковещательной рассылки информации в сети

(broadcasting)
– Прокладка кабеля между домами (вес ребер — стоимость прокладки кабеля между парой домов)
– Spanning Tree Protocol в телекоммуникационных сетях стандарта Ethernet для предотвращения образования циклов в сети

Алгоритмы построения MST

Алгоритм	Вычислительная сложность		
	Матрица смежности	Список смежности + бинарная куча	Список смежности + фибоначчиева куча
Крускала (J. Kruskal, 1956)	$O(\lvert E\rvert \cdot \log\lvert V\rvert)$		
Прима (V. Jarnik, 1930; R. Prim, 1957; E. Dijkstra, 1959)	$O(\lvert V\rvert^2)$	$O((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert)$	$O(\lvert E\rvert + \lvert V\rvert\log\lvert V\rvert)$
Борувки (O. Borůvka, 1926)	$O(\lvert E\rvert\log\lvert V\rvert)$		
Шазелля (B. Chazelle, 2000)	$O(\lvert E\rvert \cdot \alpha(\lvert E\rvert, \lvert V\rvert))$, $\alpha(m, n)$ – обратная функция Аккермана		

8

Система непересекающихся множеств

- **Система непересекающихся множеств** *(disjoint-set data structure)* – это структура данных для представления непересекающихся множеств
- Поддерживает следующие операции:
  – MakeSet(i) – создает множество из одного элемента i
  – FindSet(i) – возвращает номер множества, которому принадлежит i
  – UnionSets(i, j) – объединяет множества, содержащие элементы i и j

- MakeSet(1)
- MakeSet(2)
- MakeSet(5)
- MakeSet(6)

4 множества: {1}, {2}, {5}, {6}

- UnionSets(1, 2)

{1, 2}, {5}, {6}

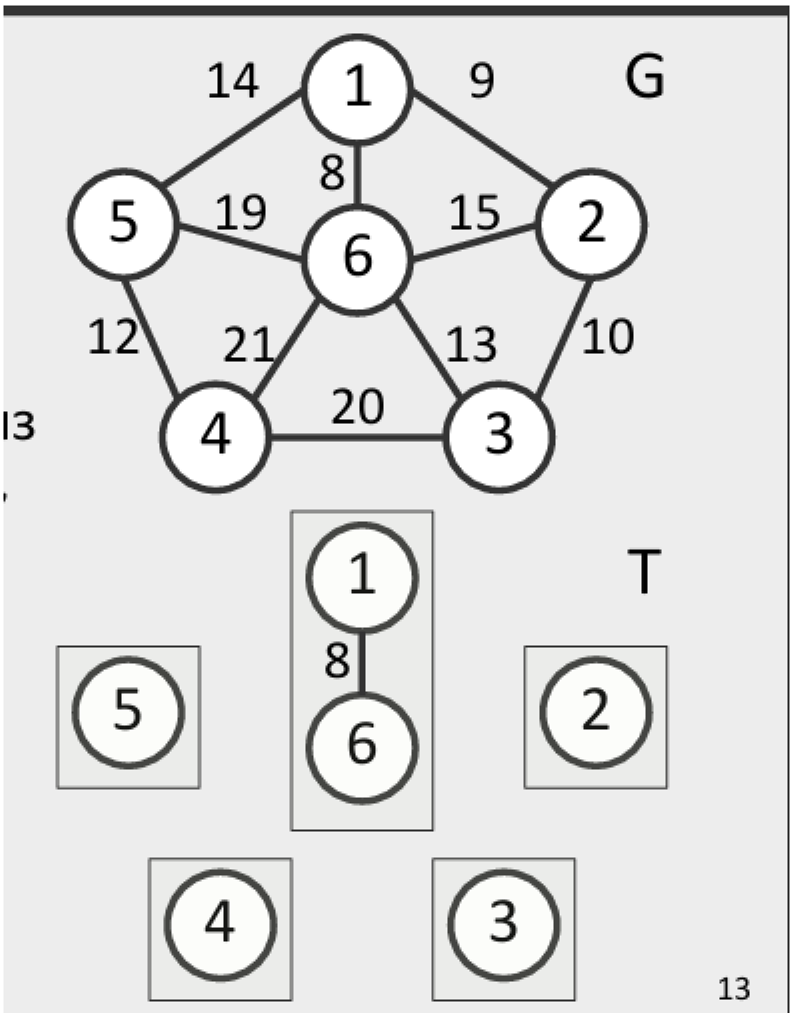
- UnionSets(2, 5)

{1, 2, 5}, {6}

10

Алгоритм Крускала

1. Создается пустой граф T из n вершин, не связанных ребрами



2. Все ребра исходного графа G помещаются в очередь с приоритетом Приоритет – вес ребра wij (ребра упорядочиваются по неубыванию весов – min-heap)

Q: {1, 6}, {1, 2}, {2, 3}, {4, 5}, {3, 6}, {1, 5}, …

3. Цикл из n – 1 итерации (по количеству ребер в MST)
   3.1. Из очереди извлекается ребро (i, j) с минимальным весом (**HeapExtractMin**)
   3.2. Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

Алгоритм Крускала на основе бинарной кучи и MFSET

$$T_{Kruskal} = \underbrace{O(|V|)}_{\text{Создание множеств}} + \underbrace{O(|E|\log|E|)}_{\text{Вставка ребер в очередь}} + \underbrace{O(|E|\log|E|)}_{\text{Извлечение ребер из очереди}} + \underbrace{O(|V|^2)}_{\text{Объединение множеств}}$$

$$\log|E| \le \log|V|^2 = 2\log|V|$$

$$\boxed{T_{Kruskal} = O(|E|\log|E|) + O(|V|^2) = O(|E|\log|V|) + O(|V|^2)}$$

- От чего зависит сложность алгоритма Крускала:
  – Реализация сортировки ребер по их весу
  – Реализация системы непересекающихся множеств

24

```c
// Структура для представления множества
struct set {
    int size;   // Количество элементов в множестве
    int first;  // Первый элемент множества (для связного списка)
};

// Структура элемента множества
struct elem {
    int set;    // Индекс множества, к которому принадлежит элемент
    int next;   // Следующий элемент в связном списке множества
};

// Структура системы непересекающихся множеств
struct mfset {
    struct set *sets;   // Массив множеств
    struct elem *elems; // Массив элементов
    int nelems;         // Количество элементов
    int nsets;          // Количество множеств
};

/**
 * Создает новую систему непересекающихся множеств
 * @param nelems - количество элементов
 * @return указатель на созданную структуру
 */
struct mfset *mfset_create(int nelems) {
    struct mfset *p = malloc(sizeof(*p));
    p->nelems = nelems;
    p->nsets = 0;

    // Выделяем память под массивы
    p->sets = malloc(sizeof(struct set) * nelems);
    p->elems = malloc(sizeof(struct elem) * nelems);

    // Инициализация всех элементов
    for (int i = 0; i < nelems; i++) {
        p->sets[i].size = 0;    // Множества изначально пусты
        p->sets[i].first = -1;  // Нет первого элемента
        p->elems[i].set = -1;   // Элемент не принадлежит ни одному множеству
        p->elems[i].next = -1;  // Нет следующего элемента
    }

    return p;
}

/**
 * Освобождает память, занятую системой множеств
 * @param set - указатель на систему множеств
 */
void mfset_free(struct mfset *set) {
    free(set->sets);
    free(set->elems);
    free(set);
}

/**
 * Создает новое множество, содержащее один элемент
 * @param set - система множеств
 * @param elem - индекс элемента
```

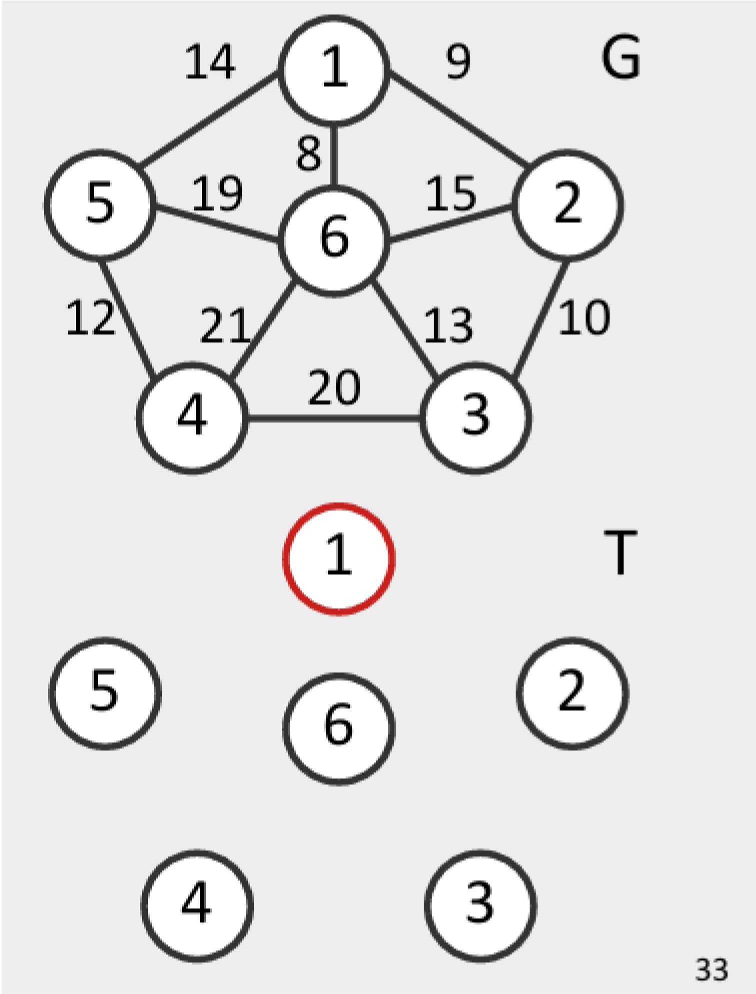
```
 */
void mfset_makeset(struct mfset *set, int elem) {
    set->sets[set->nsets].size = 1;      // Размер нового множества = 1
    set->sets[set->nsets].first = elem;  // Первый элемент - переданный
элемент
    set->elems[elem].set = set->nsets;   // Элемент ссылается на свое
множество
    set->elems[elem].next = -1;          // Нет следующего элемента
    set->nsets++;                        // Увеличиваем счетчик множеств
}

/**
 * Находит множество, содержащее указанный элемент
 * @param set - система множеств
 * @param elem - индекс элемента
 * @return индекс множества
 */
int mfset_findset(struct mfset *set, int elem) {
    return set->elems[elem].set;  // Просто возвращаем множество элемента
}

/**
 * Объединяет два множества, содержащие указанные элементы
 * @param set - система множеств
 * @param elem1 - первый элемент
 * @param elem2 - второй элемент
 */
void mfset_union(struct mfset *set, int elem1, int elem2) {
    int temp, i, set1, set2;

    // Находим множества для каждого элемента
    set1 = mfset_findset(set, elem1);
    set2 = mfset_findset(set, elem2);

    // Всегда объединяем меньшее множество с большим
    if (set->sets[set1].size < set->sets[set2].size) {
        temp = set1;
        set1 = set2;
        set2 = temp;
    }

    // Проходим по всем элементам меньшего множества (set2)
    i = set->sets[set2].first;
    while (set->elems[i].next != -1) {
        set->elems[i].set = set1;  // Переносим в большее множество (set1)
        i = set->elems[i].next;    // Переходим к следующему элементу
    }

    // Последний элемент меньшего множества
    set->elems[i].set = set1;
    // Присоединяем его список к началу большего множества
    set->elems[i].next = set->sets[set1].first;
    set->sets[set1].first = set->sets[set2].first;

    // Обновляем размер объединенного множества
    set->sets[set1].size += set->sets[set2].size;

    // Удаляем меньшее множество
    set->sets[set2].size = 0;
    set->sets[set2].first = -1;
```

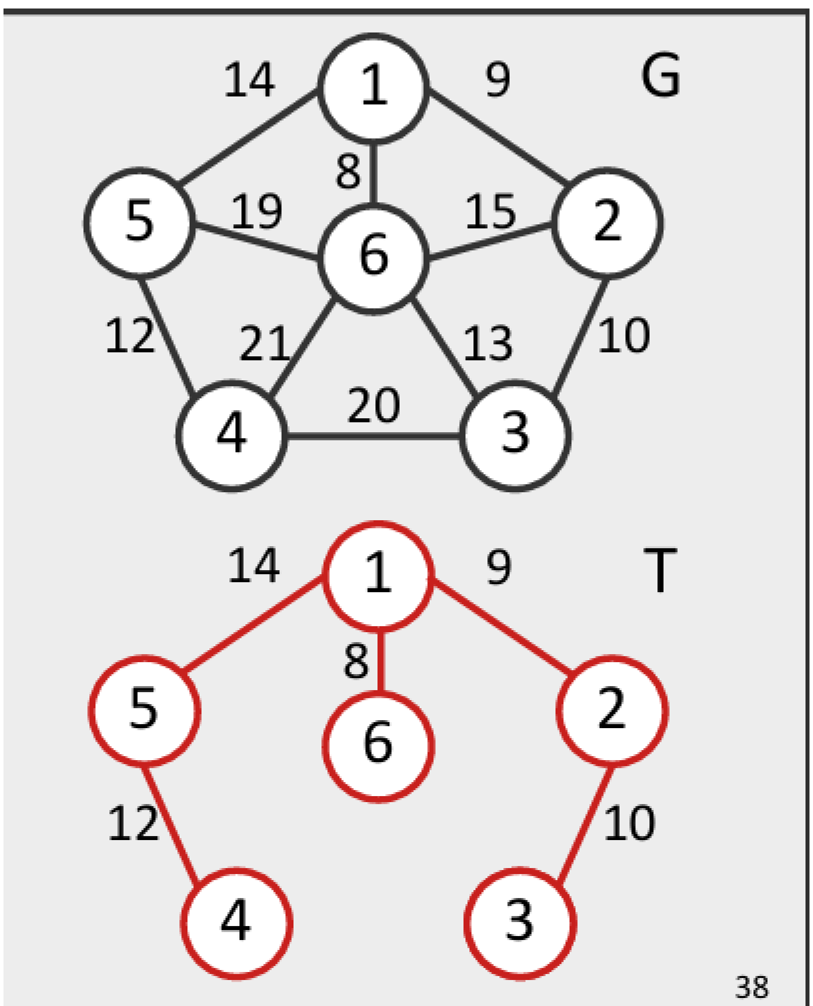
```
    set->nsets--;
}

/**
 * Поиск минимального остовного дерева алгоритмом Крускала
 * @param g - исходный граф
 * @param mst - граф для хранения MST
 * @return суммарный вес ребер MST
 */
int search_mst_kruskal(struct graph *g, struct graph *mst) {
    struct mfset *set;      // Система непересекающихся множеств
    struct heap *pq;        // Приоритетная очередь для ребер
    struct heapvalue edge;  // Ребро графа
    struct heapitem item;   // Элемент очереди
    int mstlen = 0;         // Суммарный вес MST
    int n = g->nvertices;   // Количество вершин

    // Инициализация системы множеств
    set = mfset_create(n);
    for (int i = 0; i < n; i++) {
        mfset_makeset(set, i);  // Каждая вершина - отдельное множество
    }

    // Инициализация приоритетной очереди
    pq = heap_create(n * n);

    // Добавляем все ребра графа в очередь с приоритетами
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int w = graph_get_edge(g, i + 1, j + 1);
            if (w > 0) {  // Если ребро существует
                edge.i = i;
                edge.j = j;
                heap_insert(pq, w, edge);  // Добавляем в очередь
            }
        }
    }

    // Основной цикл алгоритма Крускала
    for (int i = 0; i < n - 1; ) {
        item = heap_extract_min(pq);  // Извлекаем ребро с минимальным весом

        // Проверяем, принадлежат ли вершины разным множествам
        int s1 = mfset_findset(set, item.value.i);
        int s2 = mfset_findset(set, item.value.j);

        if (s1 != s2) {  // Если не создает цикл
            mfset_union(set, item.value.i, item.value.j);  // Объединяем
множества
            mstlen += item.priority;  // Добавляем вес ребра
            // Добавляем ребро в MST
            graph_set_edge(mst, item.value.i + 1, item.value.j + 1,
item.priority);
            i++;  // Увеличиваем счетчик добавленных ребер
        }
    }

    // Освобождение ресурсов
    heap_free(pq);
    mfset_free(set);
```

```
    return mstlen;
}
```

Алгоритм Прима

1. Создается пустой граф T
2. Во множество U помещается вершина 1, с которой начинается формирование остова
3. Цикл, пока U ≠ V
   3.1. Найти ребро (i, j) с наименьшим весом такое, что i ∈ U и j ∈ V
   3.2. Добавить ребро (i, j) в граф T
   3.3. Добавить вершину j во множество U


G
T
14
9
8
19
15
12
21
13
10
20
33


G
T
14
9
8
19
15
12
21
13
10
20
38