ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Департамент программной инженерии

Алгоритмы и структуры данных

Семинар №11. 2021-2022 учебный год

Нестеров Роман Александрович, *ДПИ ФКН и НУЛ ПОИС*

Бессмертный Александр Игоревич, *ДПИ ФКН*

Higher School of Economics

Faculty of Computer science

De grão em grão, a galinha enche o papo

Где мы?

Специальные структуры данных			
Вероятностные структуры			
Хеширование, хеш-таблицы ✓		**Disjoint sets** ✓	
Сбалансированные деревья ✓		**B-деревья** ✓	
Рекурсивные структуры данных			
Список ✓		**Дерево** ✓	
Элементарные (линейные) структуры данных			
Массив ✓	**Строка** ✓	**Стек** ✓	**Очередь** ✓

План

- Алгоритм Краскала: построение минимального остовного дерева с помощью СНМ
- Список с пропусками: аналог сбалансированного дерева поиска
- Фильтр Блума: кто принадлежит множеству?

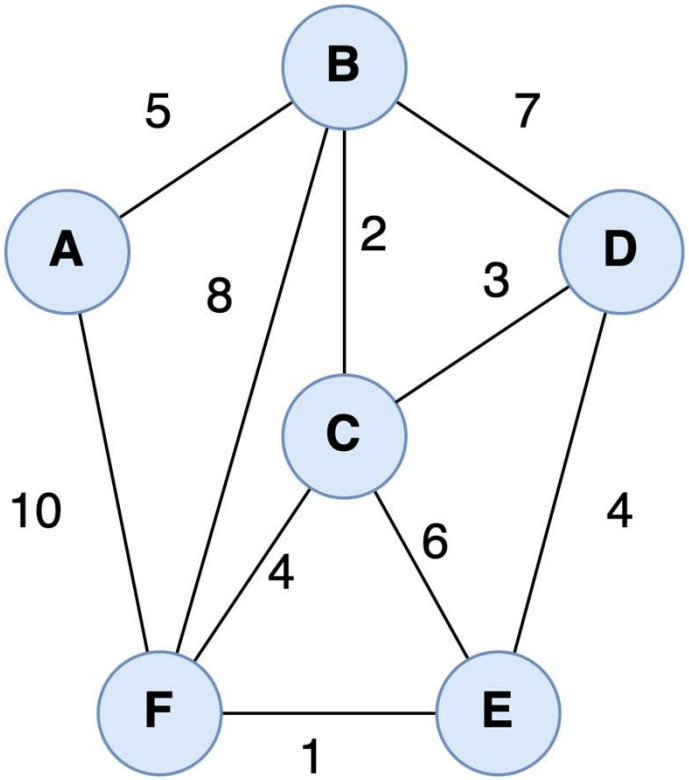
Алгоритм Краскала

Минимальное остовное дерево

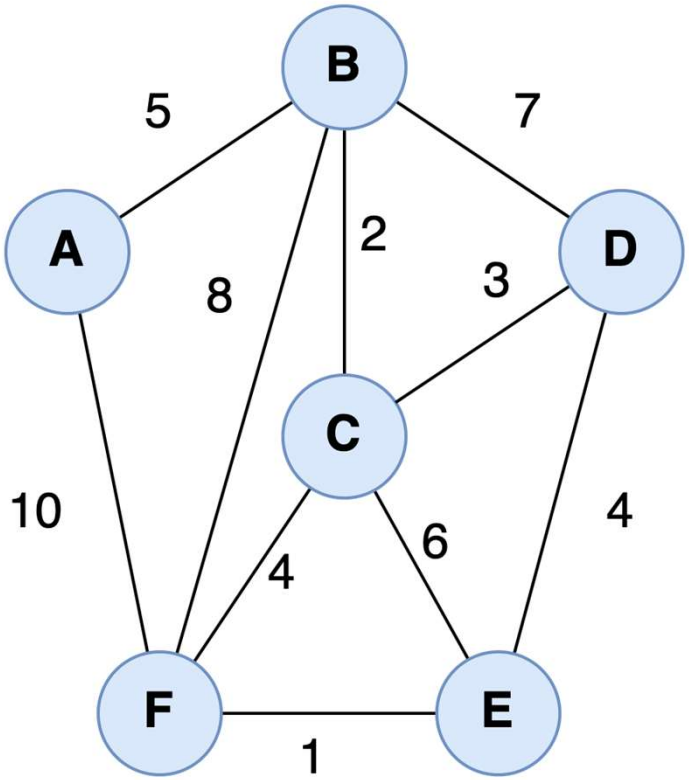
Дан взвешенный неориентированный граф.

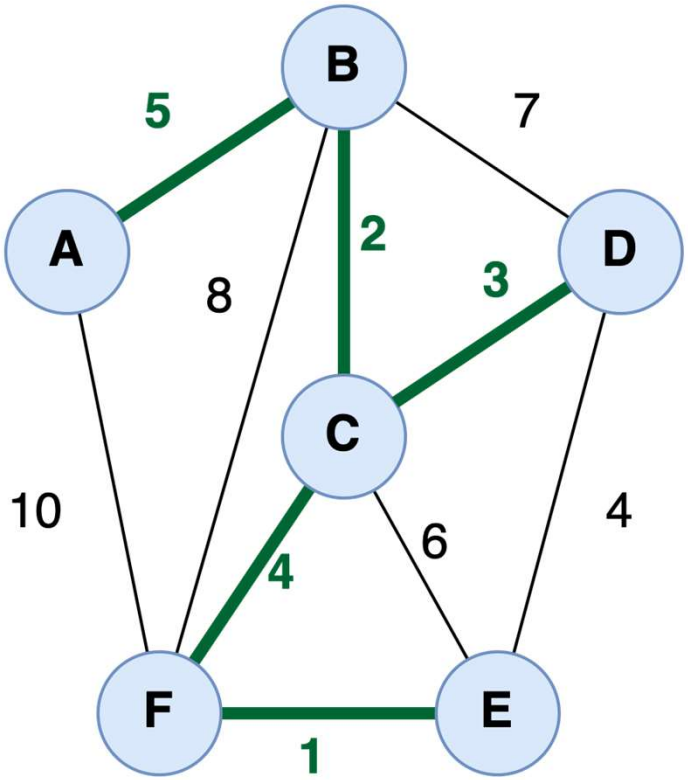
Требуется найти такое поддерево этого графа, которое соединяет все его вершины и имеет наименьший суммарный вес ребер.

Минимальное остовное дерево

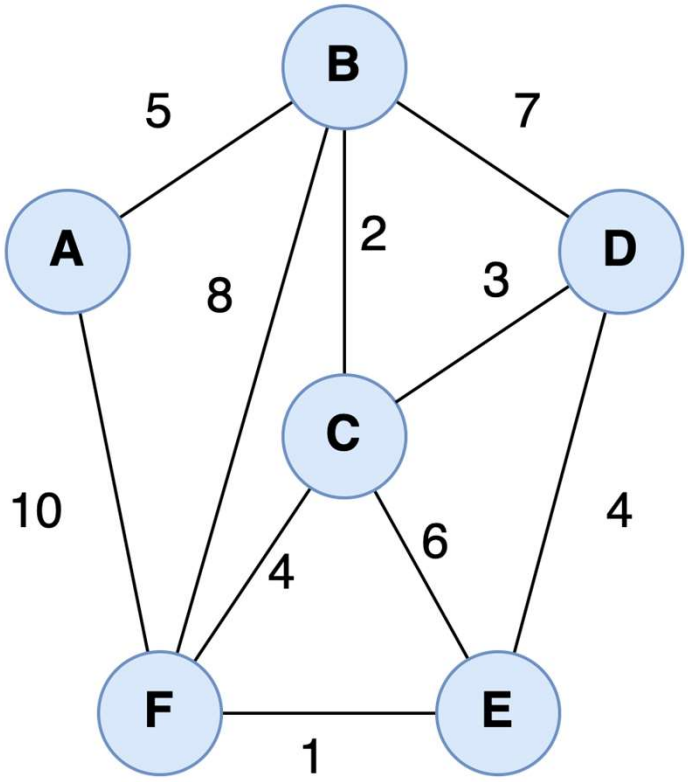


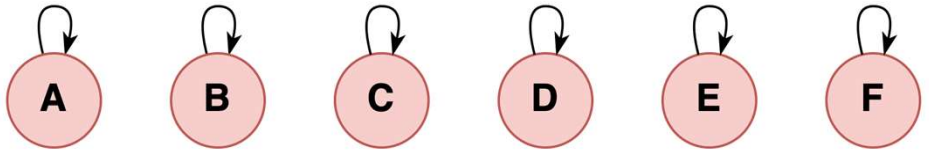
Минимальное остовное дерево



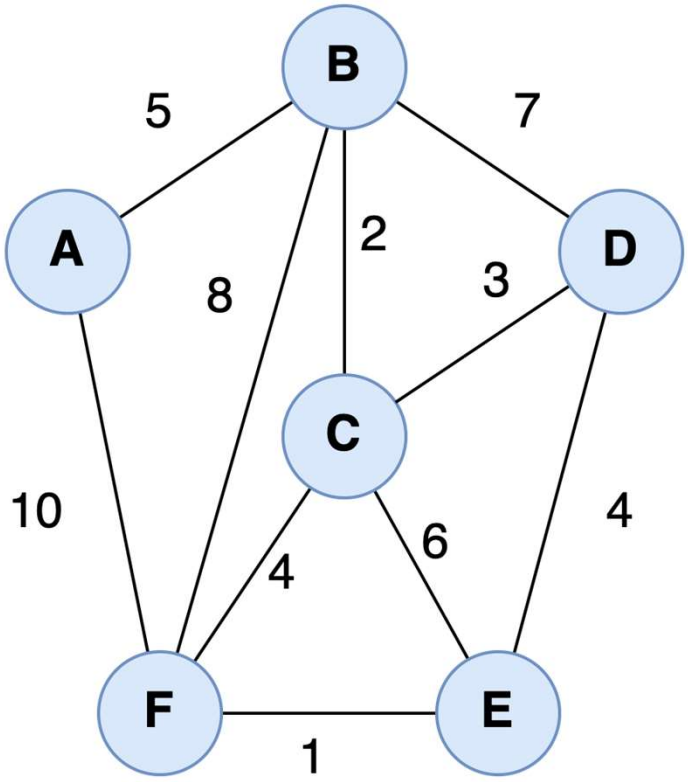


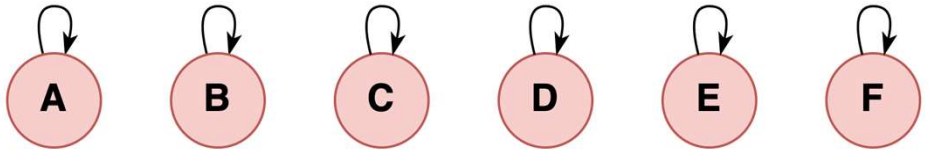
Минимальное остовное дерево





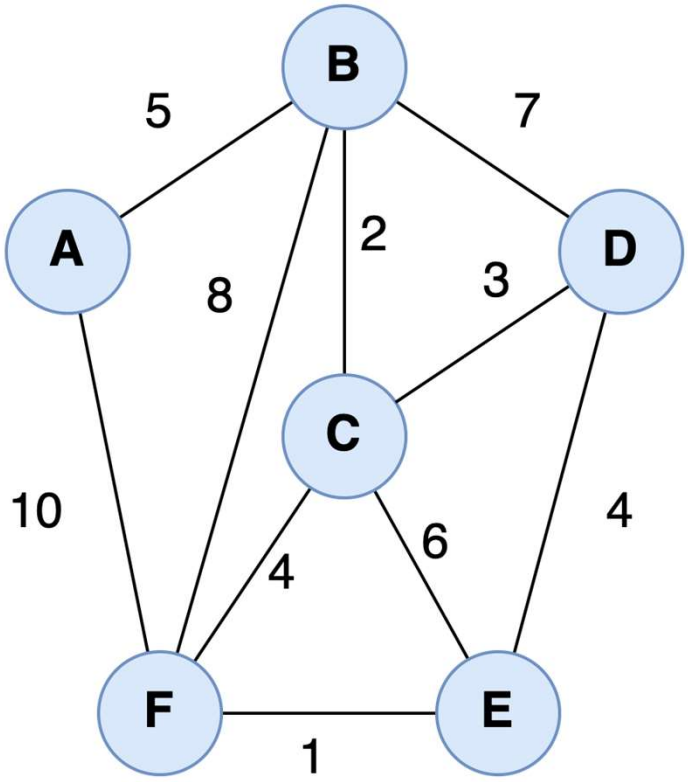
Минимальное остовное дерево

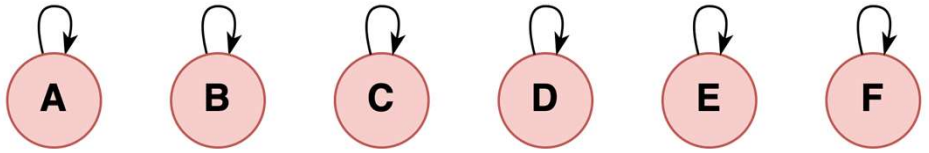





```
find(F) == find(E)
```

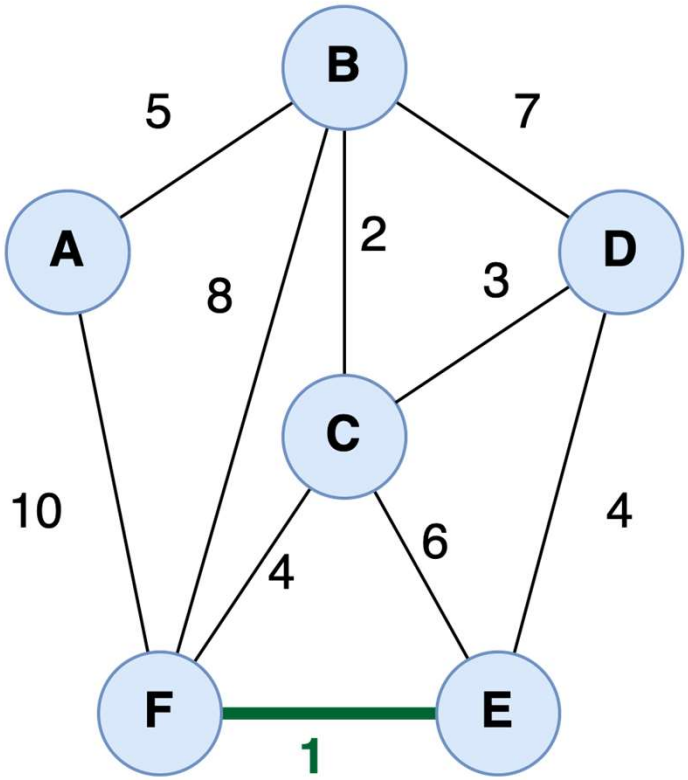
Минимальное остовное дерево

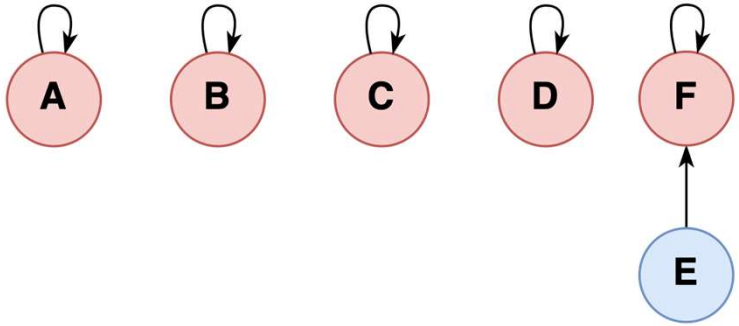





```
find(F) == find(E) -> NO, unite(F, E)
```

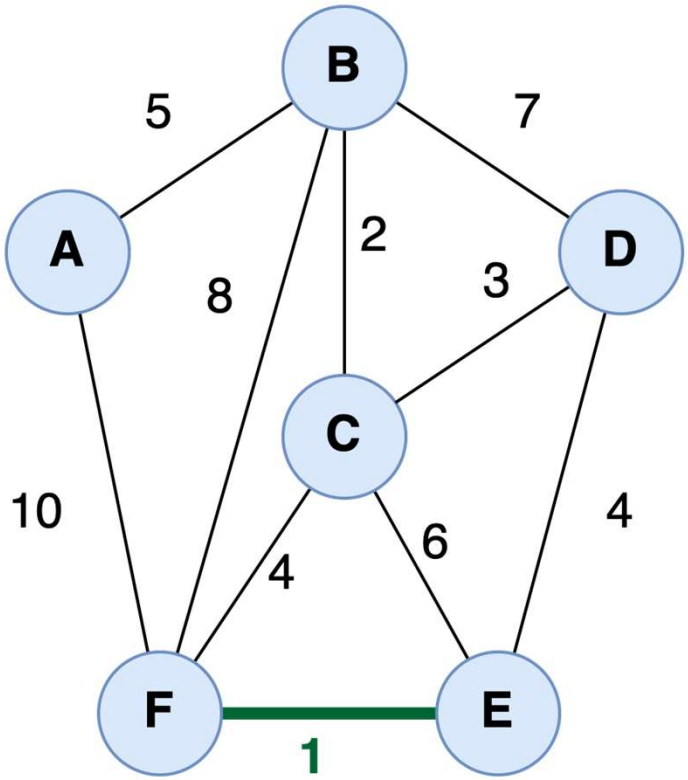
Минимальное остовное дерево

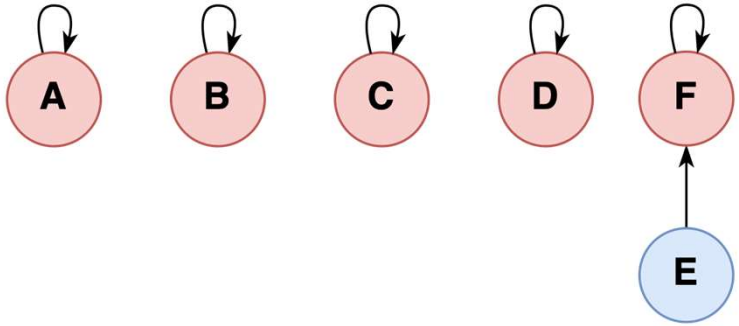





```
find(F) == find(E) -> NO, unite(F, E)
```

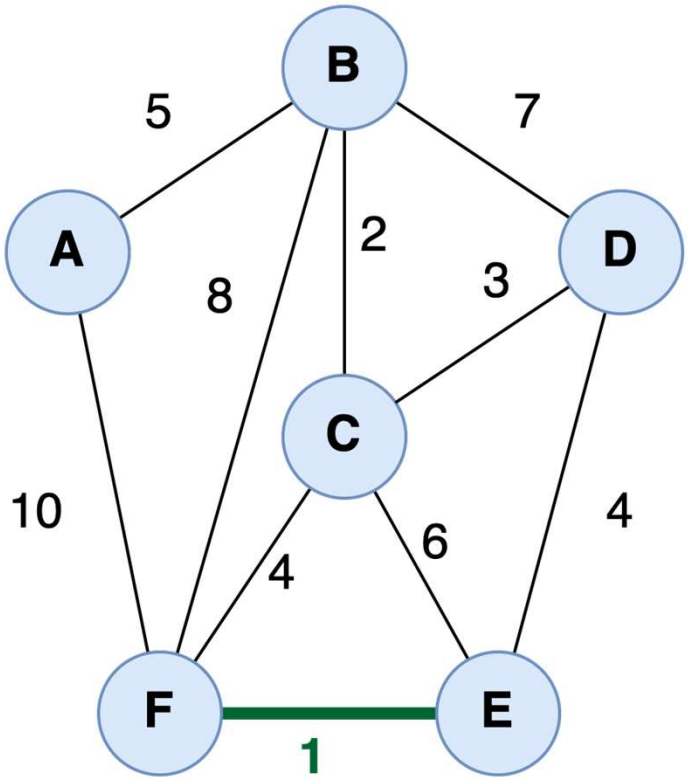
Минимальное остовное дерево

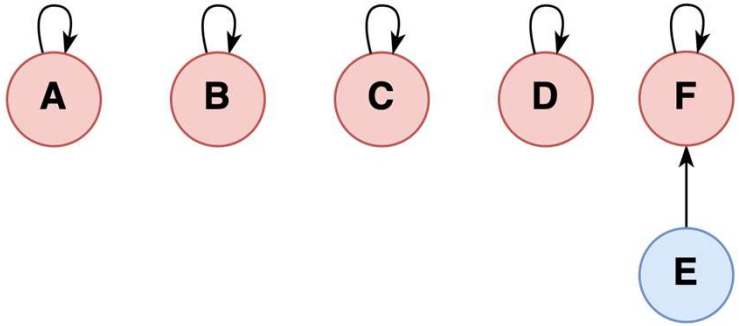





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C)
```

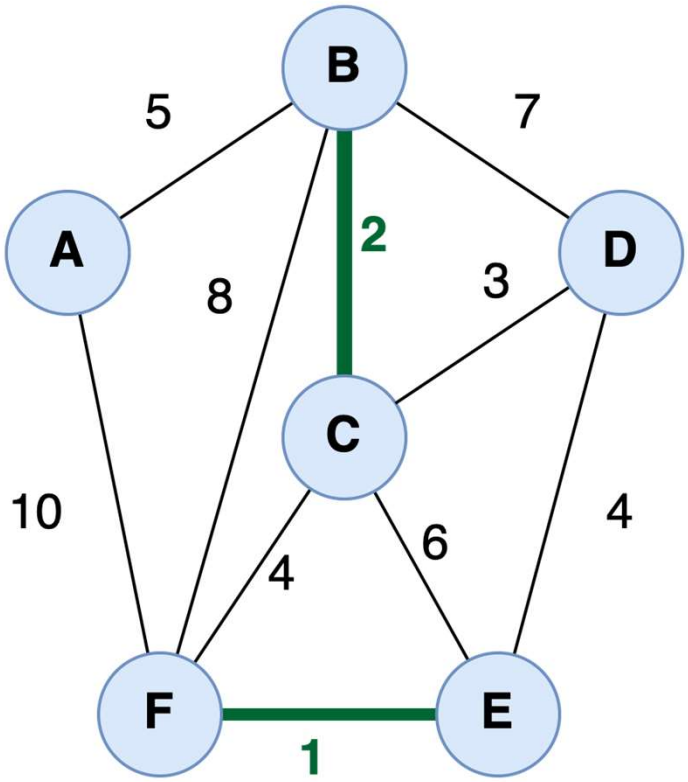
Минимальное остовное дерево

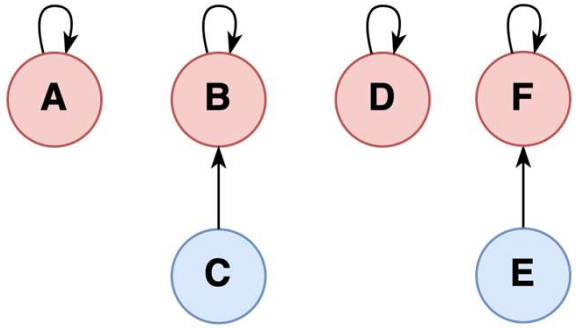





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
```

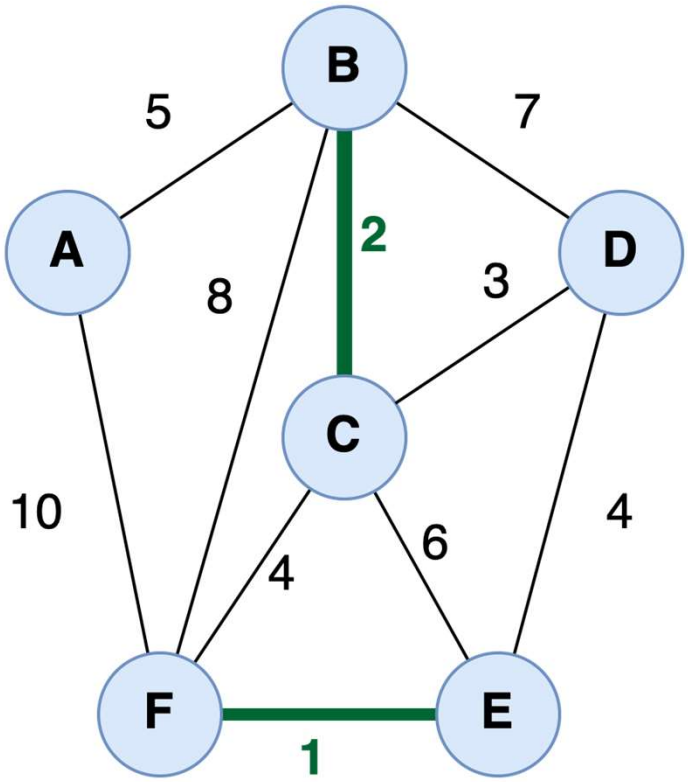
Минимальное остовное дерево

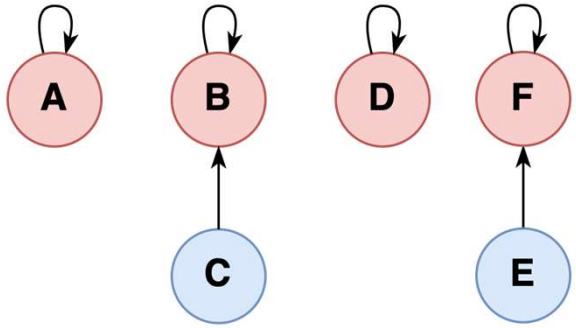





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
```

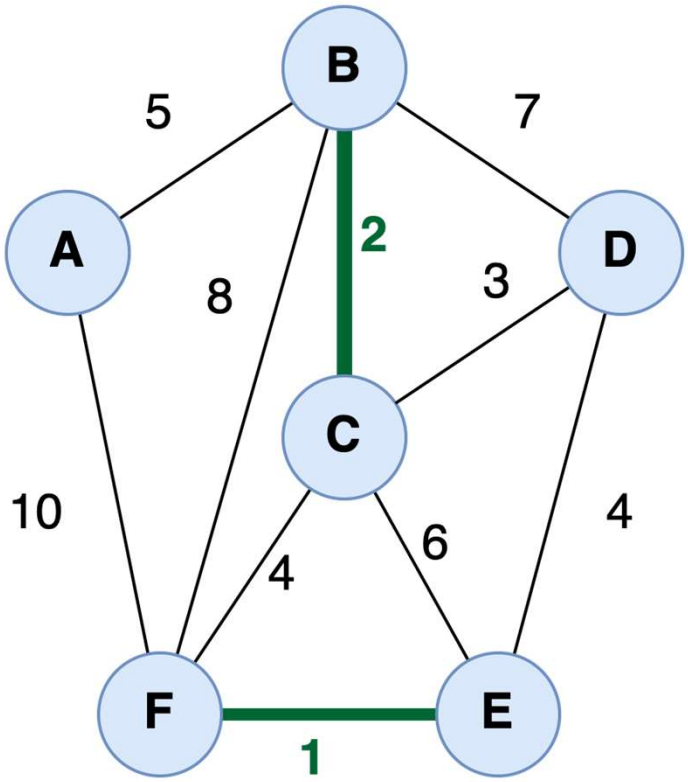
Минимальное остовное дерево

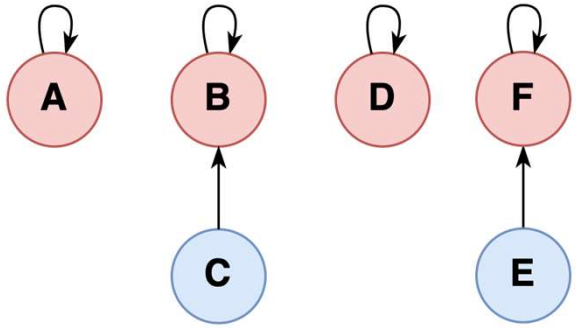





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D)
```

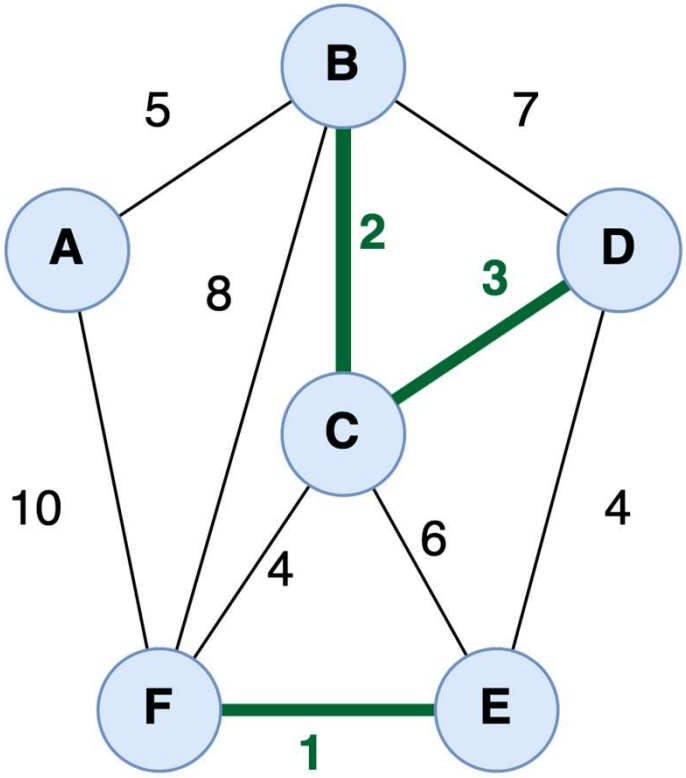
Минимальное остовное дерево

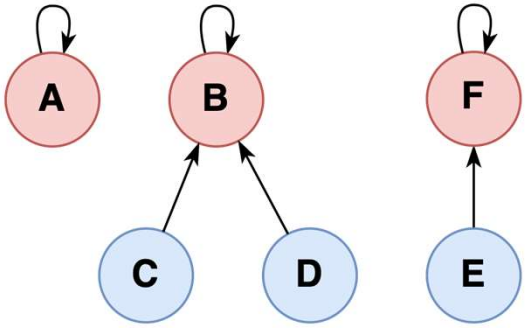





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
```

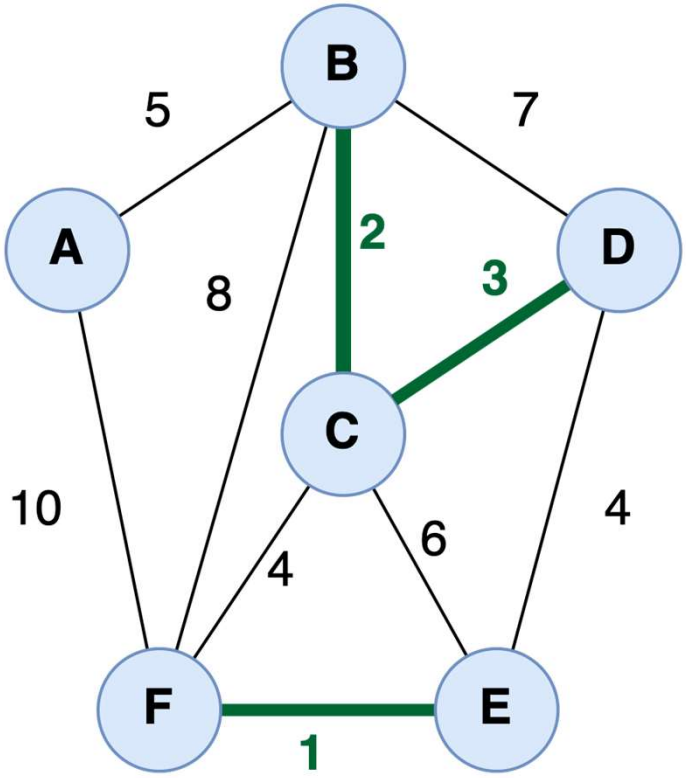
Минимальное остовное дерево

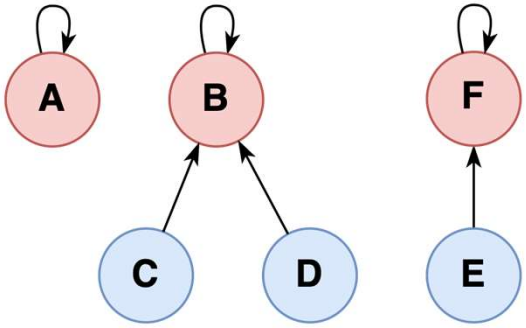





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
```

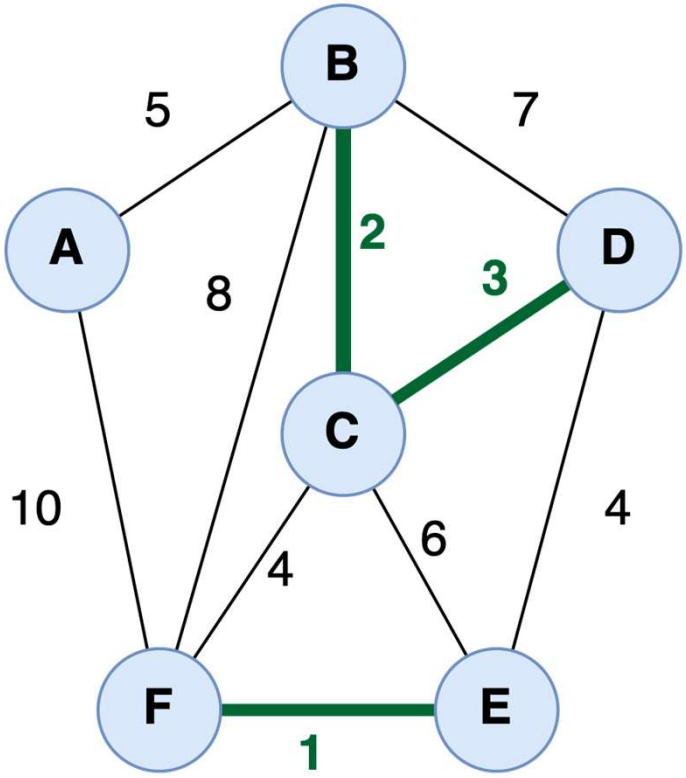
Минимальное остовное дерево

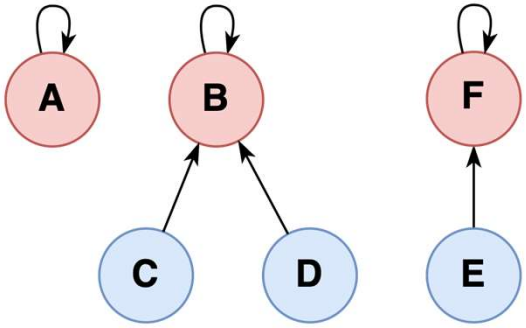





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F)
```

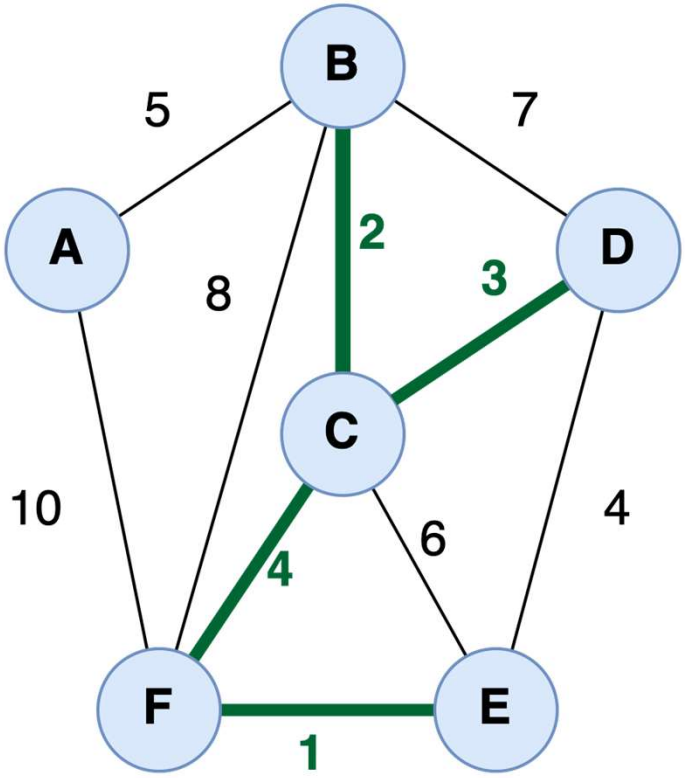
Минимальное остовное дерево

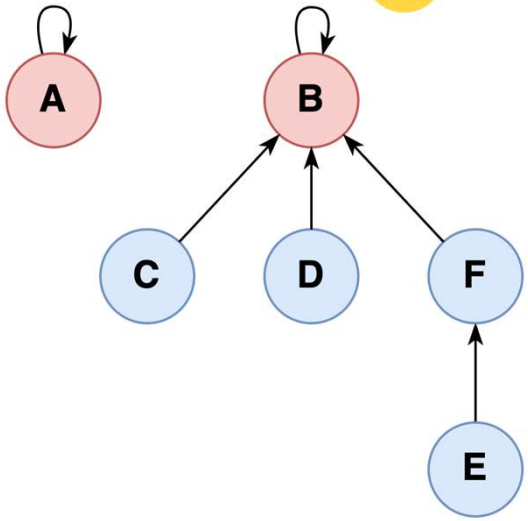





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
```

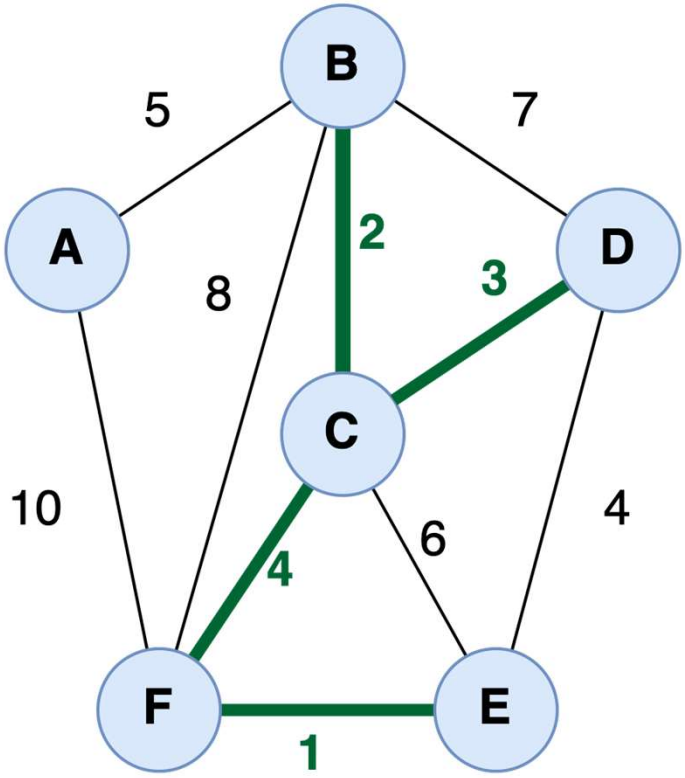
Минимальное остовное дерево

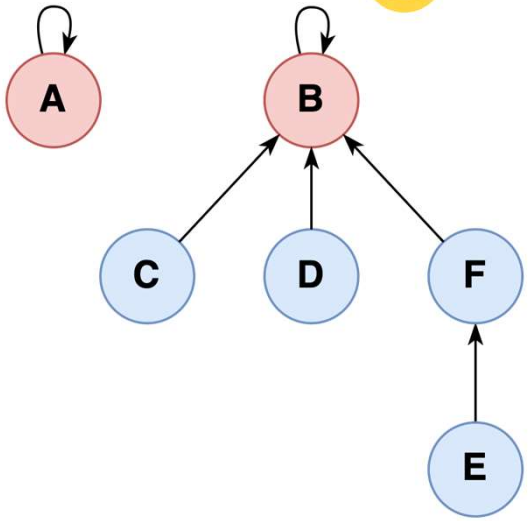





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
```

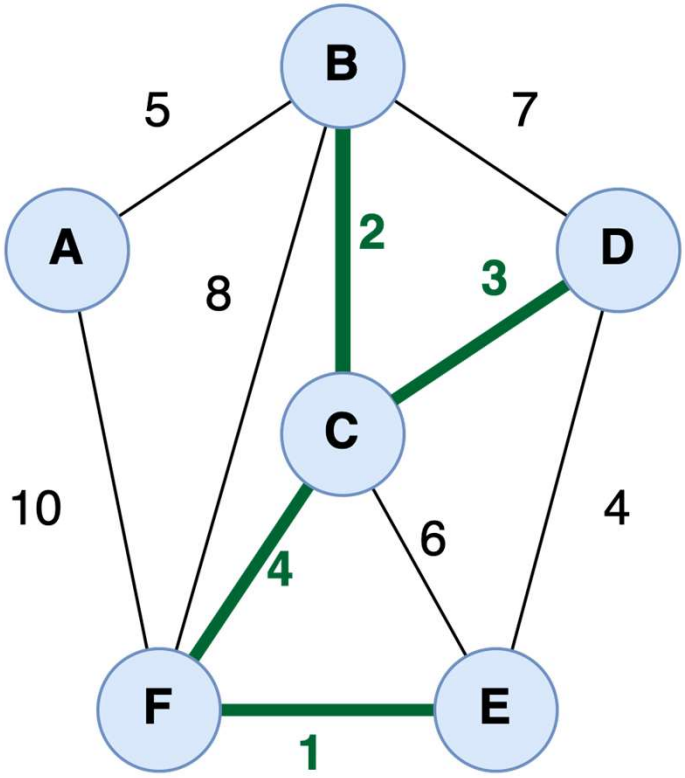
Минимальное остовное дерево

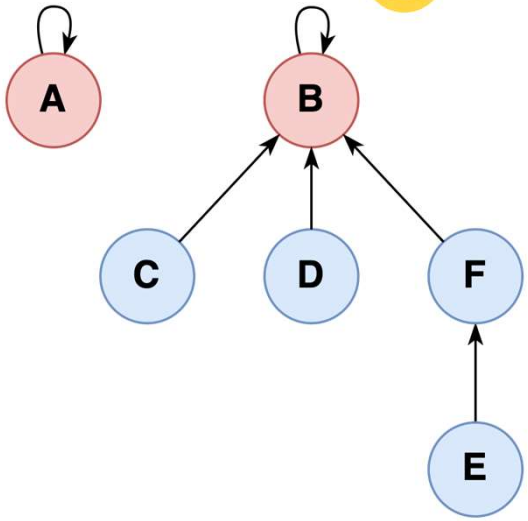





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
find(D) == find(E) -> YES
```

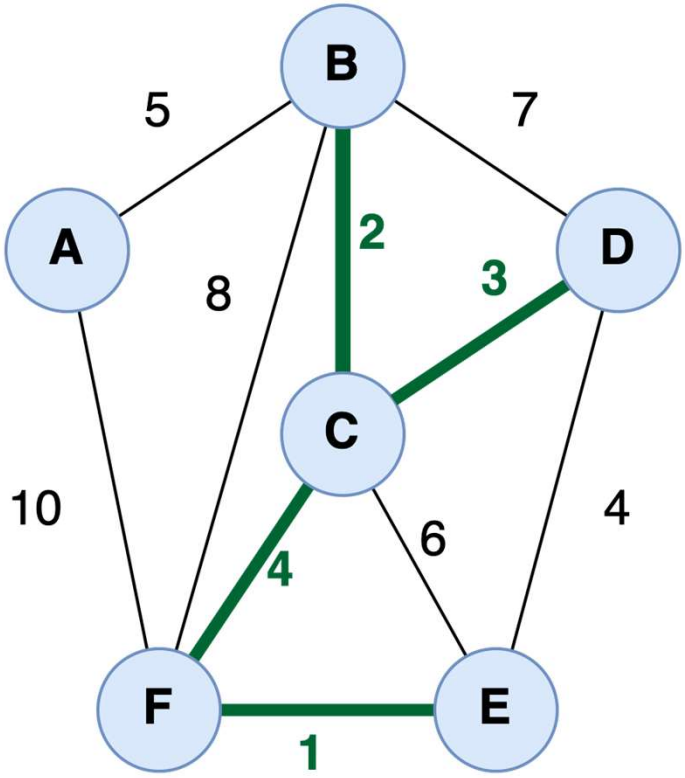
Минимальное остовное дерево

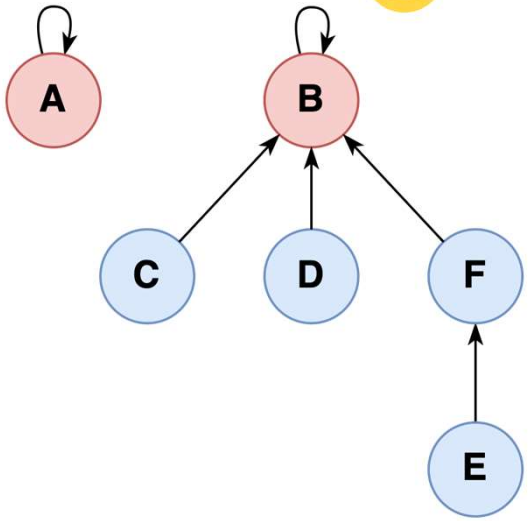





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
find(D) == find(E) -> YES, pass
```

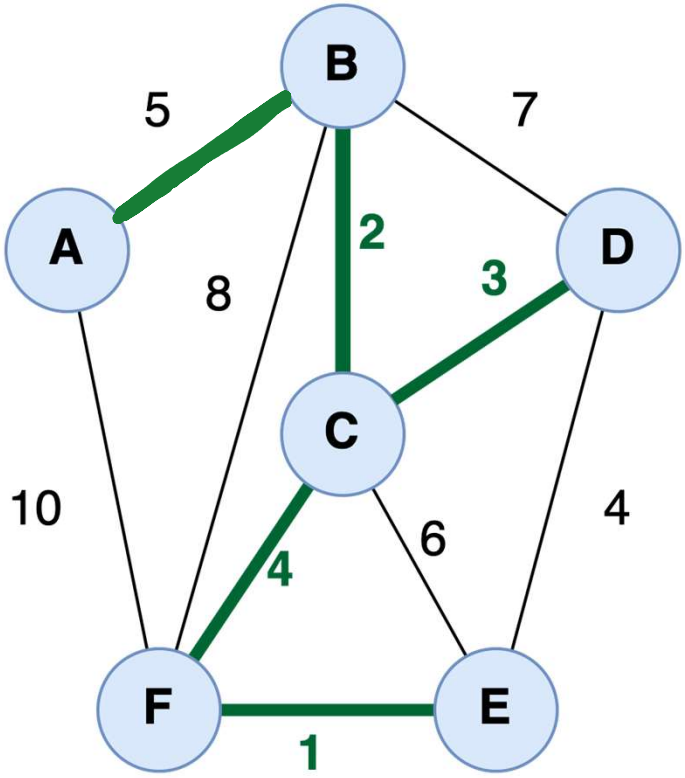
Минимальное остовное дерево

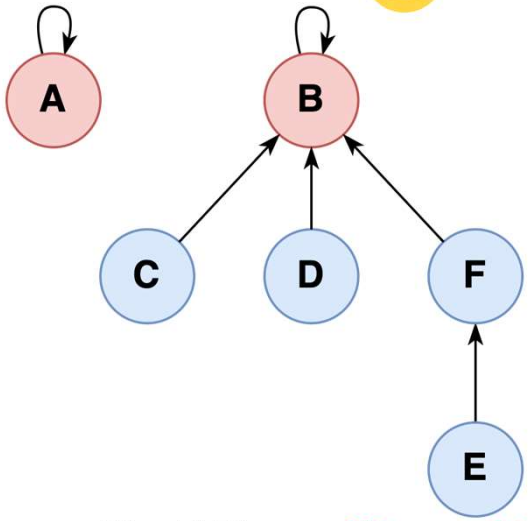





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
find(D) == find(E) -> YES, pass
find(A) == find(B)
```

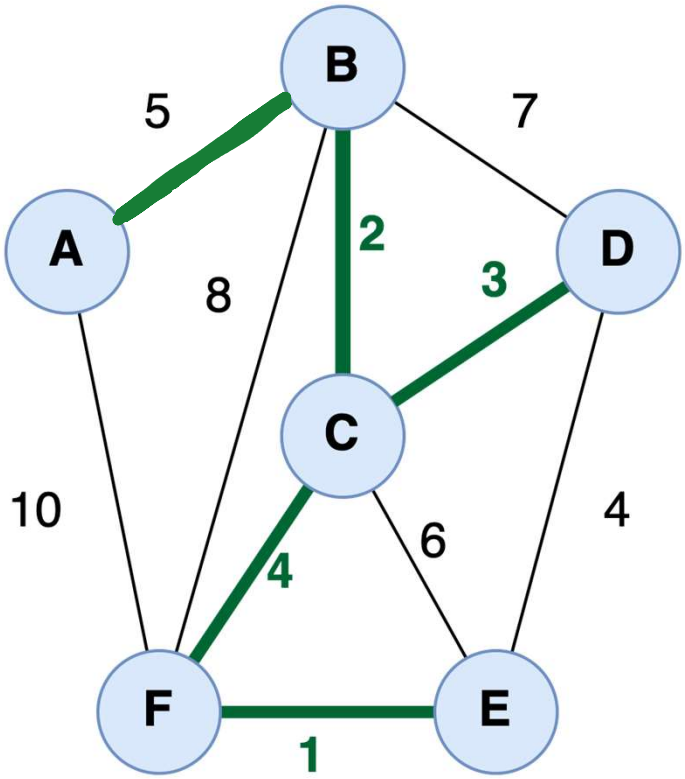
Минимальное остовное дерево

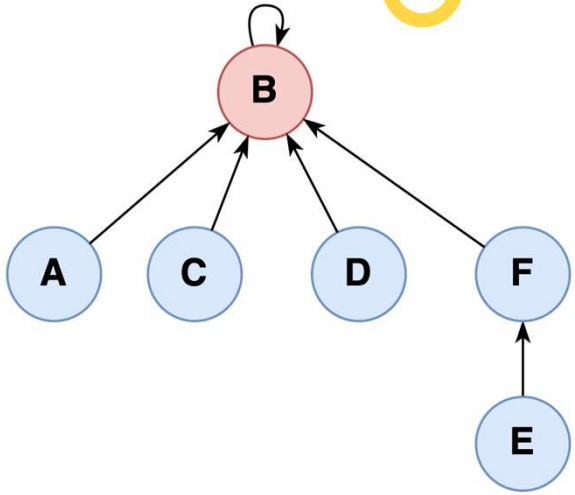





```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
find(D) == find(E) -> YES, pass
find(A) == find(B) -> NO, unite(A, B)
```

Минимальное остовное дерево






```
find(F) == find(E) -> NO, unite(F, E)
find(B) == find(C) -> NO, unite(B, C)
find(C) == find(D) -> NO, unite(C, D)
find(C) == find(F) -> NO, unite(C, F)
find(D) == find(E) -> YES, pass
find(A) == find(B) -> NO, unite(A, B)
```

Список с пропусками (Skip-list)

Список

- Просто вставлять и удалять элементы – перестановка нескольких указателей
- Не требуется знать объем необходимой памяти

- Долгий проход по списку и поиск – **O(n)**

Список с пропусками

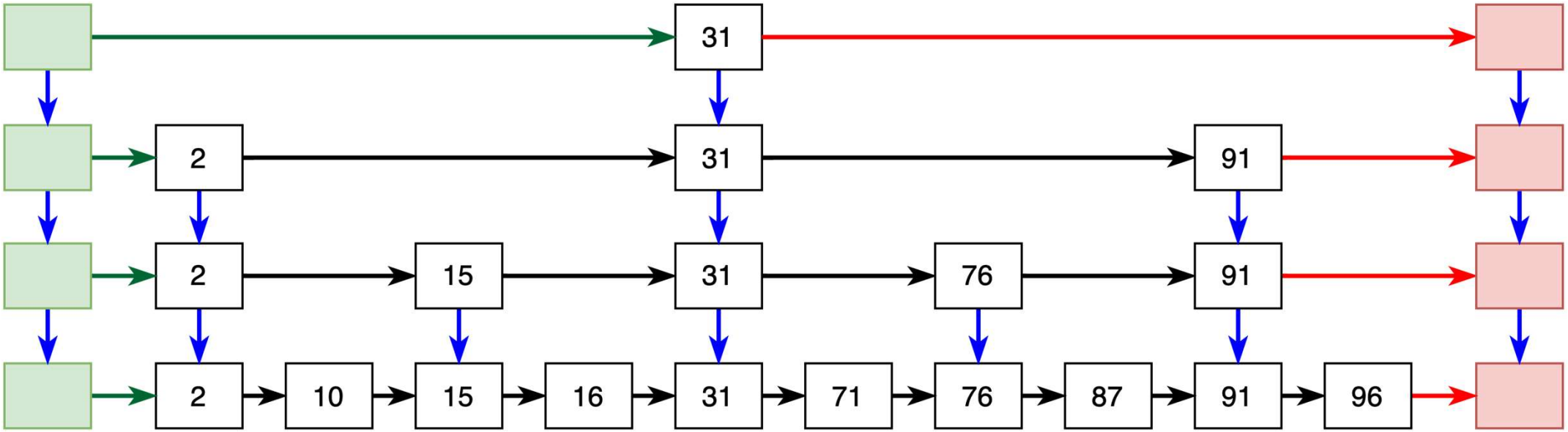
Уильям Пью, 1989 г.
Статья "**Skip Lists: Probabilistic Alternative to Balanced Trees**" в журнале *Communications of the ACM*.

- Обобщение отсортированных списков
- Ожидаемое время поиска - **`O(log n)`**
- *Вероятностная* структура данных

Идеальный список с пропусками

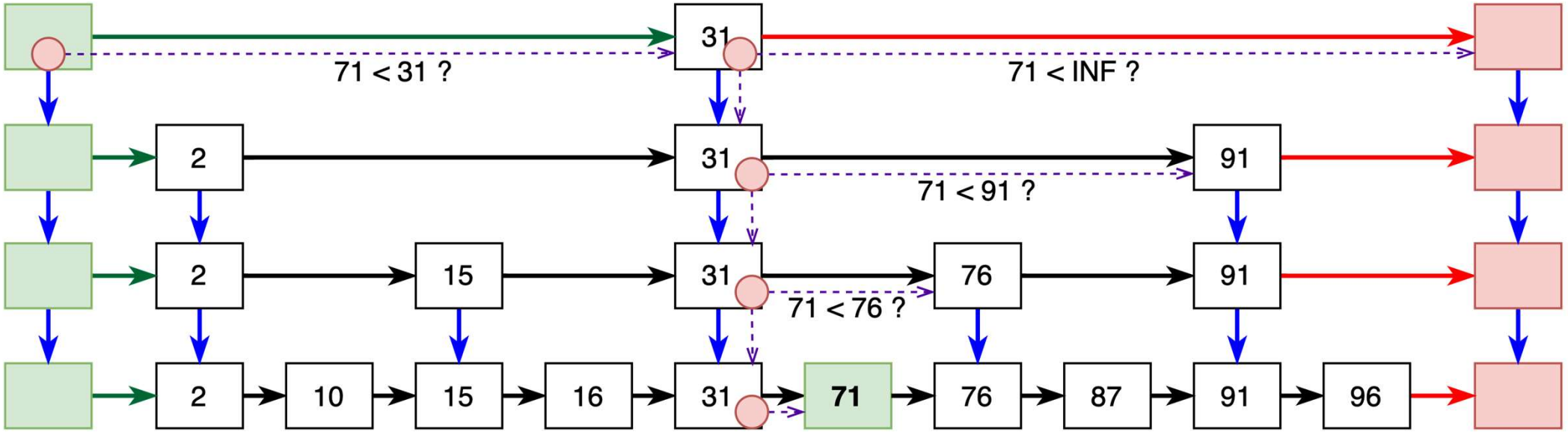
- Ключи хранятся в отсортированном виде.
- Содержит **`O(log n)`** уровней, каждый из которых также является списком.
- Все ключи хранятся только на последнем уровне, а каждый вышестоящий уровень содержит *половину* ключей, которые хранятся уровнем ниже.

Идеальный список с пропусками



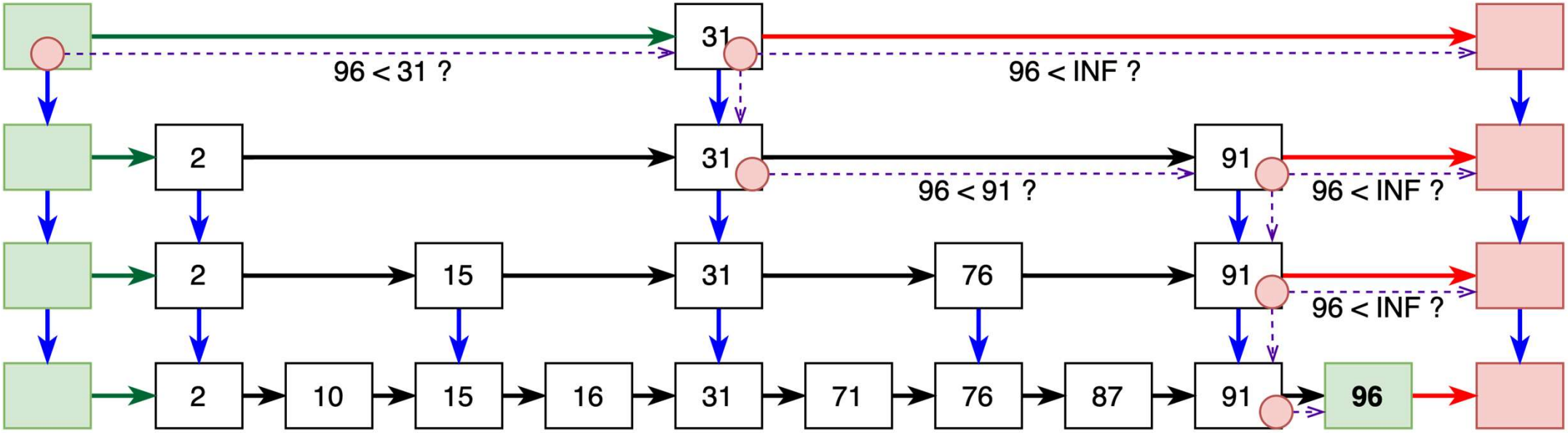
Идеальный список с пропусками. Поиск

`search(71)`



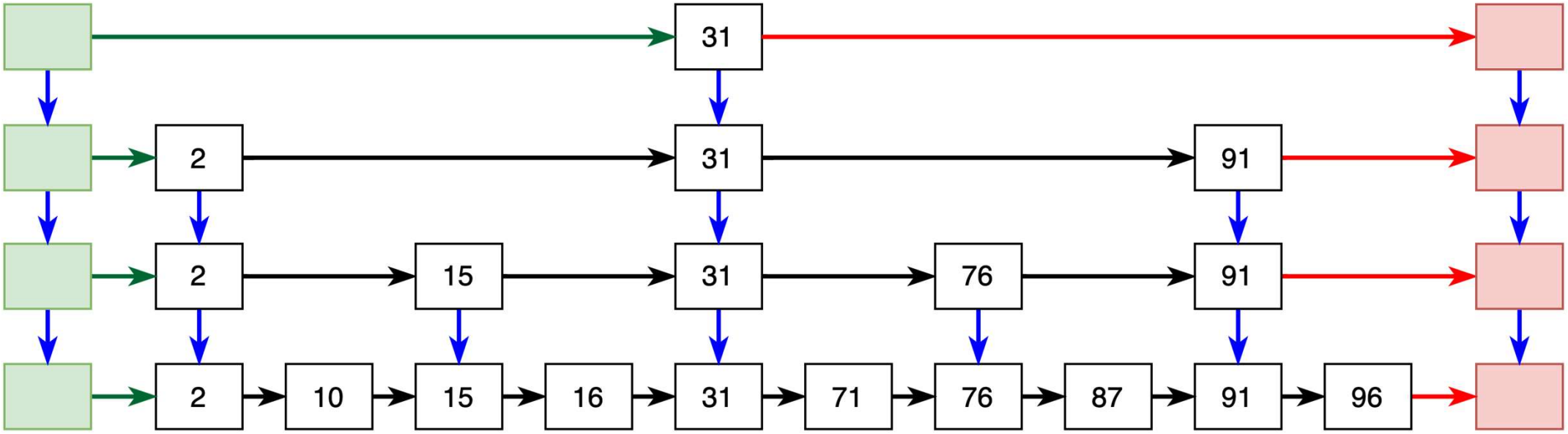
Идеальный список с пропусками. Поиск

`search(96)`



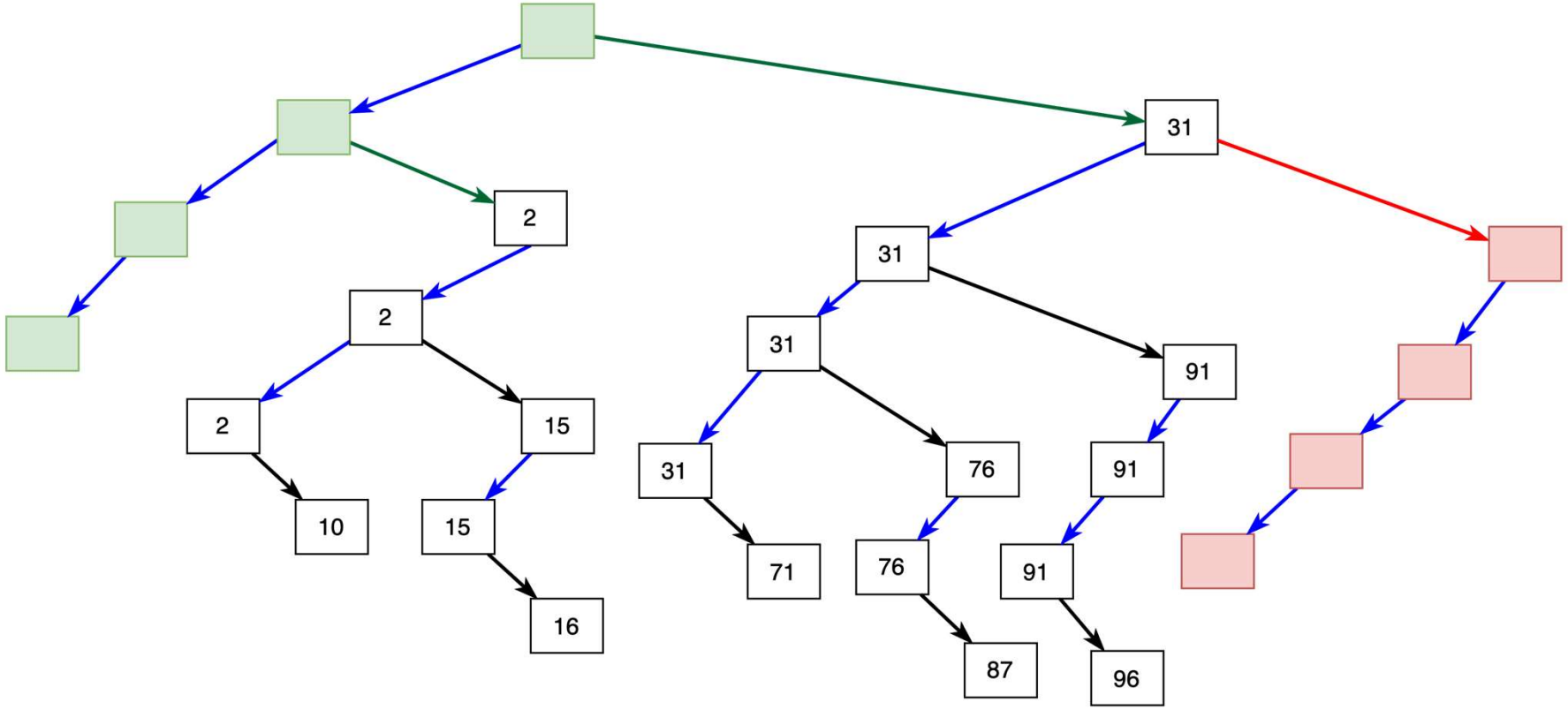
Идеальный список с пропусками

Все это уж очень похоже на дерево…



Идеальный список с пропусками

Все это уж очень похоже на дерево…



Идеальный список с пропусками. Поиск

```
Node *cur = head;

for (int i = MAX_LEVEL; i > 0; --i) {
    while (!cur->next[i] &&
            cur->next[i]->data < key) {
        cur = cur->next[i];
    }

cur = cur->next[0];

if (cur->data = key) {
    return cur;
} else {
    return nullptr;
}
```

```
class Node {
    T data;
    Node **next;
    ...
}

class SkipList {
    Node *head;
    ...
}
```

Идеальный список с пропусками. Поиск

- На каждом уровне посетим **не более двух вершин**
- Сложность – **`O(log n)`**

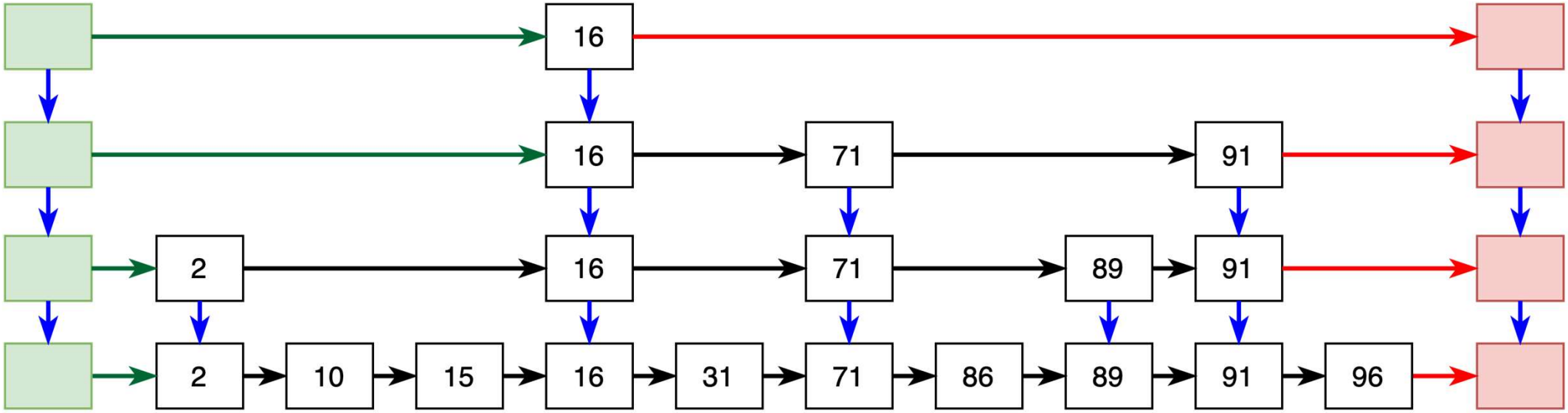
Идеальный список с пропусками. Вставка и удаление

- Из-за фиксированных требований в результате вставки и удаления потребуется перестройка всего списка.
- **Решение**: на каждом уровне *ожидается* половина ключей нижележащего уровня – используем рандомизацию

Рандомизированный список с пропусками

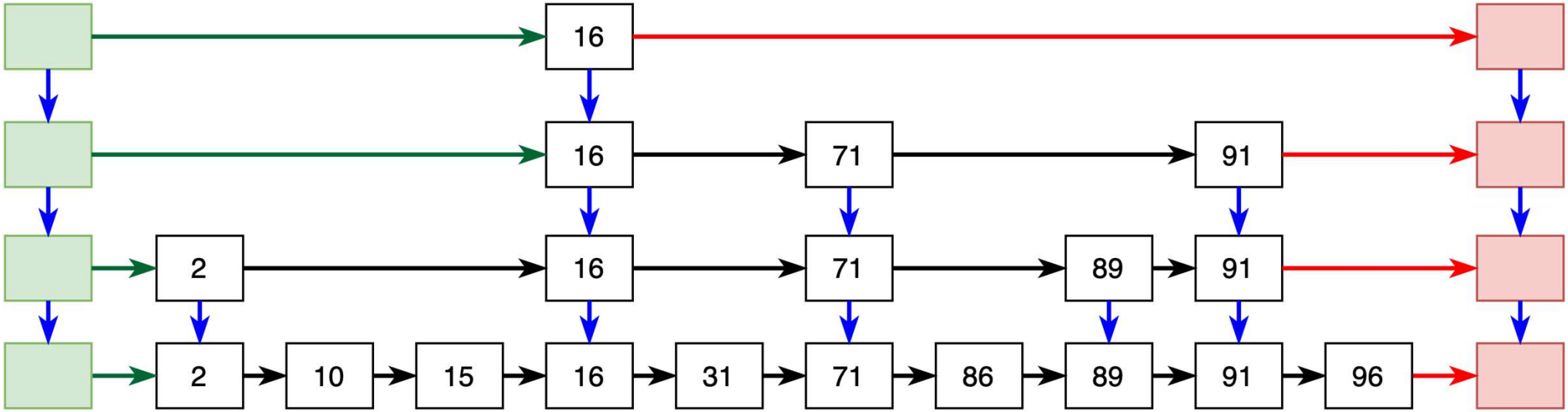
- Допускается несбалансированность
- Вероятность того, что ключ будет находиться на следующем (по высоте) уровне составляет **`0,5`**:
  - Ожидается половина всех ключей на уровне 1
  - Ожидается четверть всех ключей на уровне 2
  - …

Рандомизированный список с пропусками



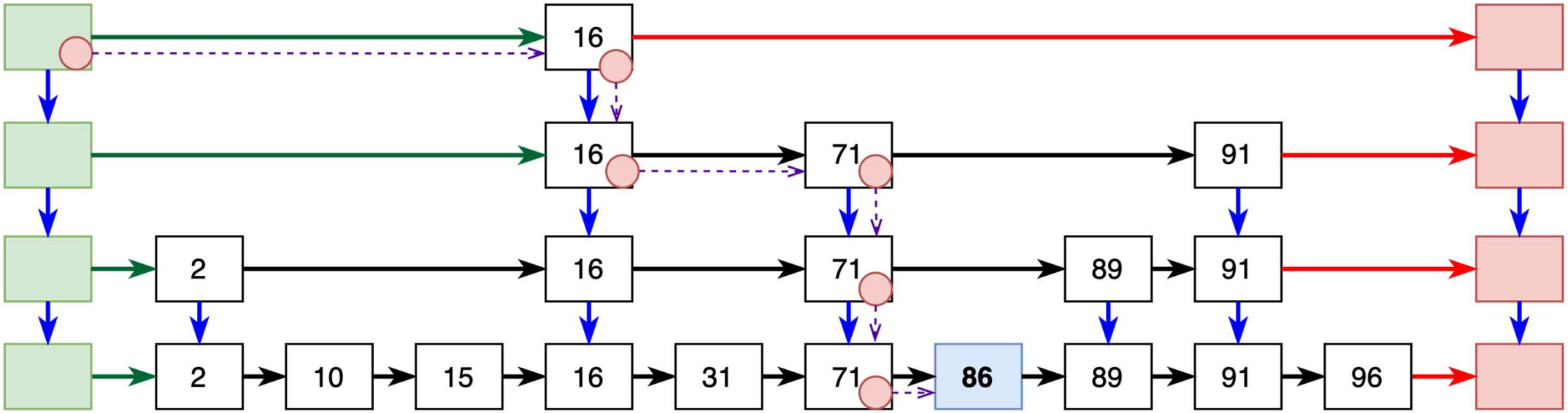
Рандомизированный список с пропусками

```
insert(87)
```



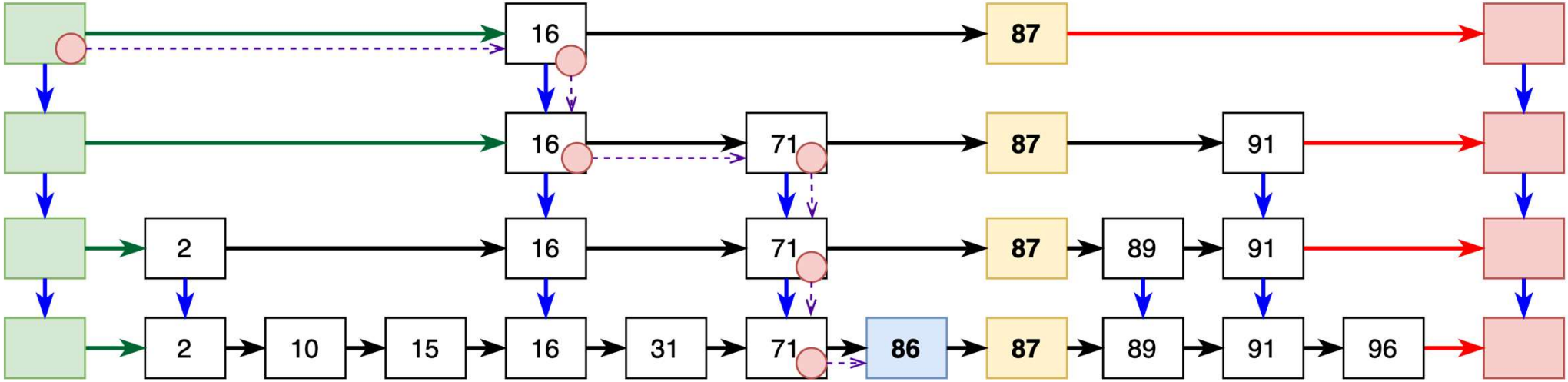
Рандомизированный список с пропусками

```
insert(87)
```



Рандомизированный список с пропусками

```
insert(87)
```



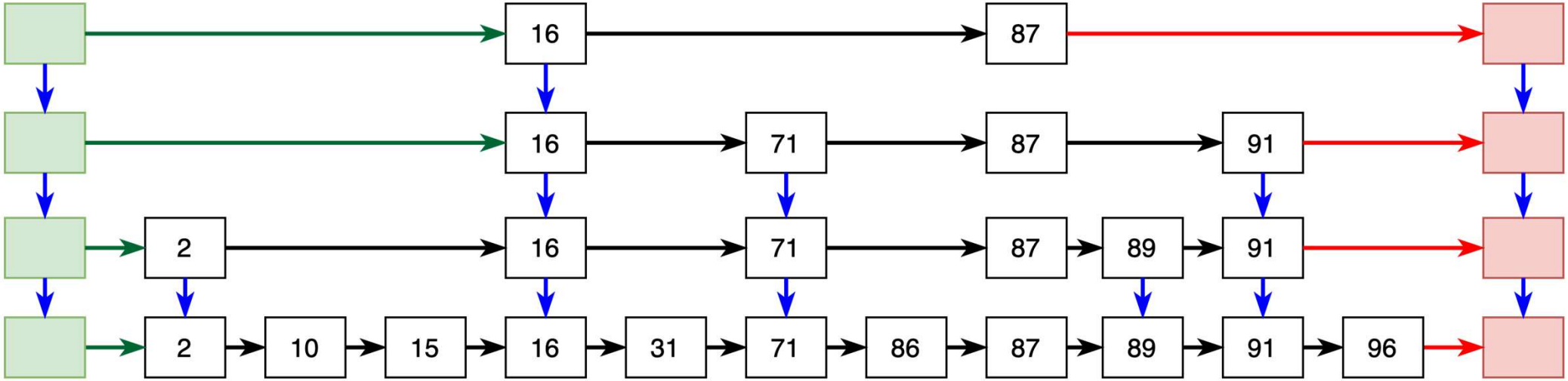
Рандомизированный список с пропусками

```
insert(87)
```

1. Вставляем ключ на самом нижнем уровне (уровень 0) стандартной вставкой в линейный список.
2. Если в результате подброса монеты получили «орла», то вставляем ключ на следующем уровне.

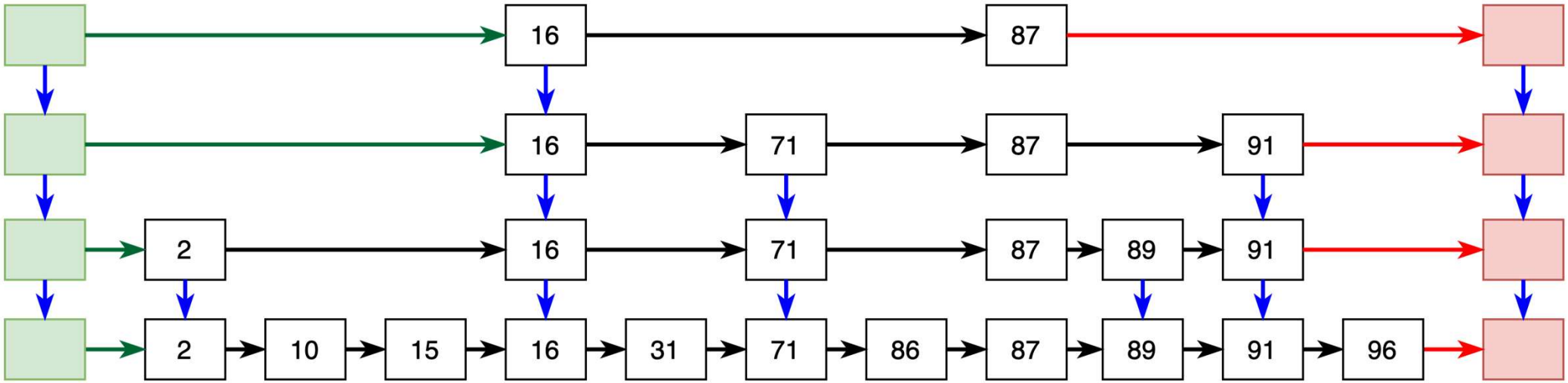
Рандомизированный список с пропусками

```
delete(91)
```



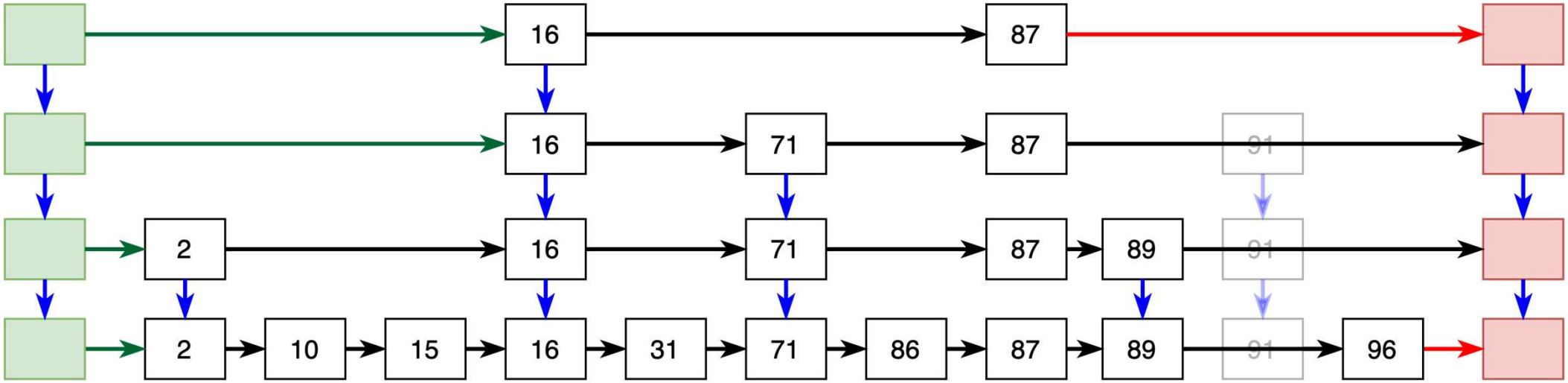
Рандомизированный список с пропусками

```
delete(91)
```



Рандомизированный список с пропусками

```
delete(91)
```



Рандомизированный список с пропусками

- Ожидаемая сложность выполнения основных операций совпадает с идеальным списком с пропусками

- Вероятность вырожденных ситуация крайне мала:
  1. Список с пропусками становится простым связным списком
  2. Каждый узел списка с пропусками будет находиться на каждом уровне

Фильтр Блума (Bloom filter)

Вероятностное множество

- Добавление элемента в множество **`add(x)`**
- Проверка принадлежности элемента множеству **`test(x)`**

- Принадлежит ли этот данный элемент множеству?
  Да/**Нет**

Вероятностное множество

- Добавление элемента в множество **`add(x)`**
- Проверка принадлежности элемента множеству **`test(x)`**

- Принадлежит ли этот данный элемент множеству?
  Возможно/Нет

Задача проверки принадлежности

- Последовательный перебор всех данных и сравнение с искомым ключом
- Построение индекса и поиск искомого ключа в индексе
- Хеш-таблица с открытой адресацией (линейное разрешение коллизий)
- Битовый массив с одной хеш-функцией
- Битовый массив несколькими хеш-функциями

Фильтр Блума

Бёртон Блум, **1970** г.

Битовый массив размерности **m** и **k** различных хеш-функций, которые равновероятно отображают исходные объекты в множество **{0, 1, …, m-1}**.

Фильтр Блума

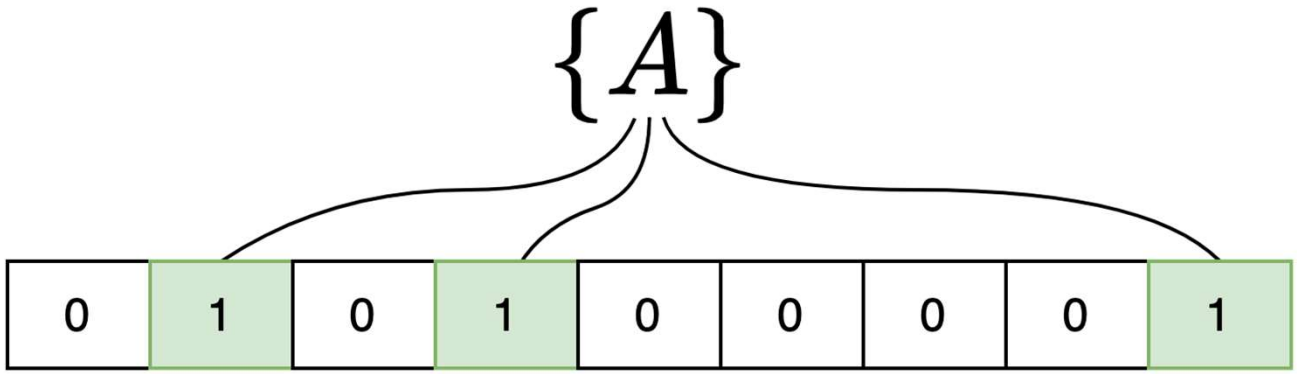
Исходное состояние – нулевой битовый массив, который соответствует пустому множеству.

$$\emptyset$$

0	0	0	0	0	0	0	0	0

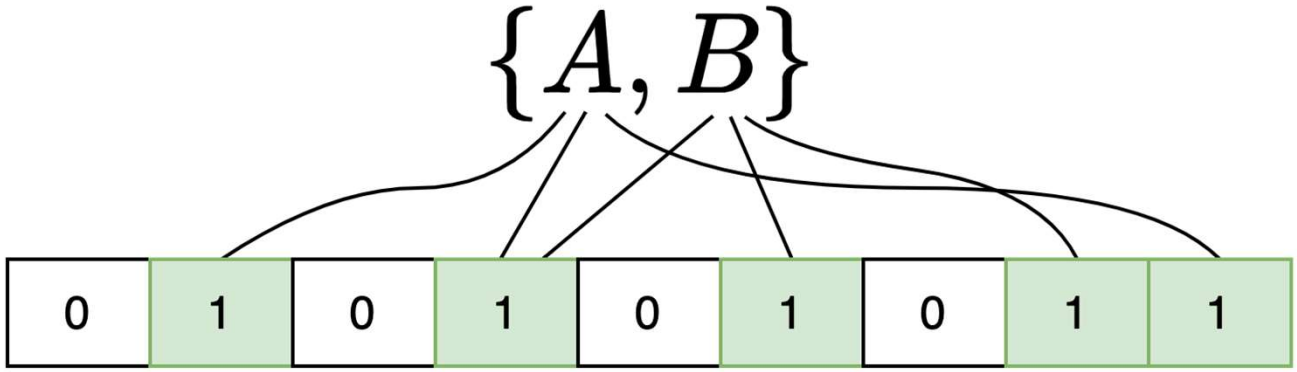
Фильтр Блума. Добавление элементов

При добавлении считаем значения хеш-функций и устанавливаем в соответствующих ячейках массива единицы.



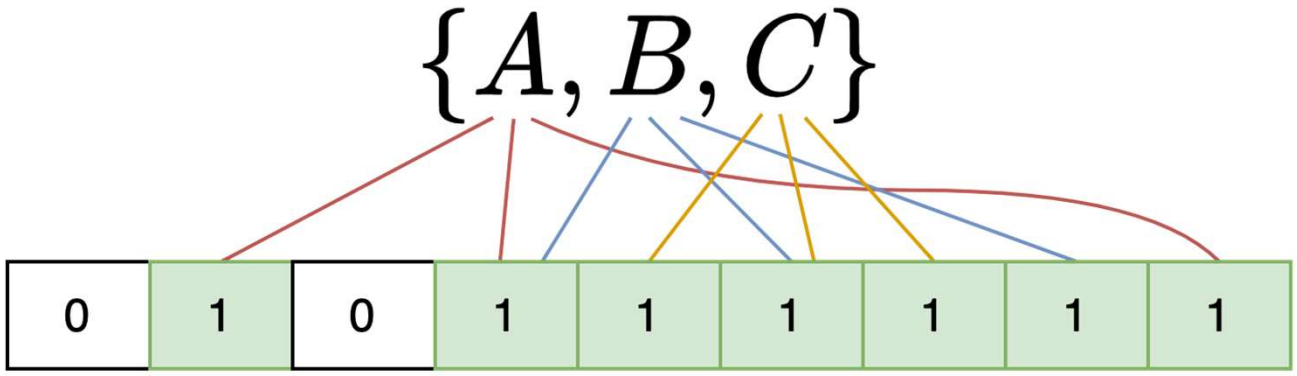
Фильтр Блума. Добавление элементов

При добавлении считаем значения хеш-функций и устанавливаем в соответствующих ячейках массива единицы.



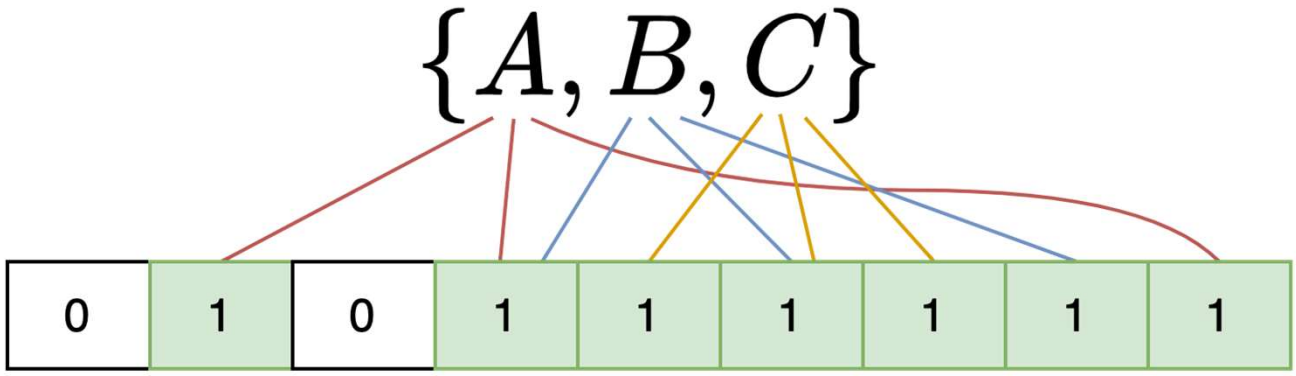
Фильтр Блума. Добавление элементов

При добавлении считаем значения хеш-функций и устанавливаем в соответствующих ячейках массива единицы.



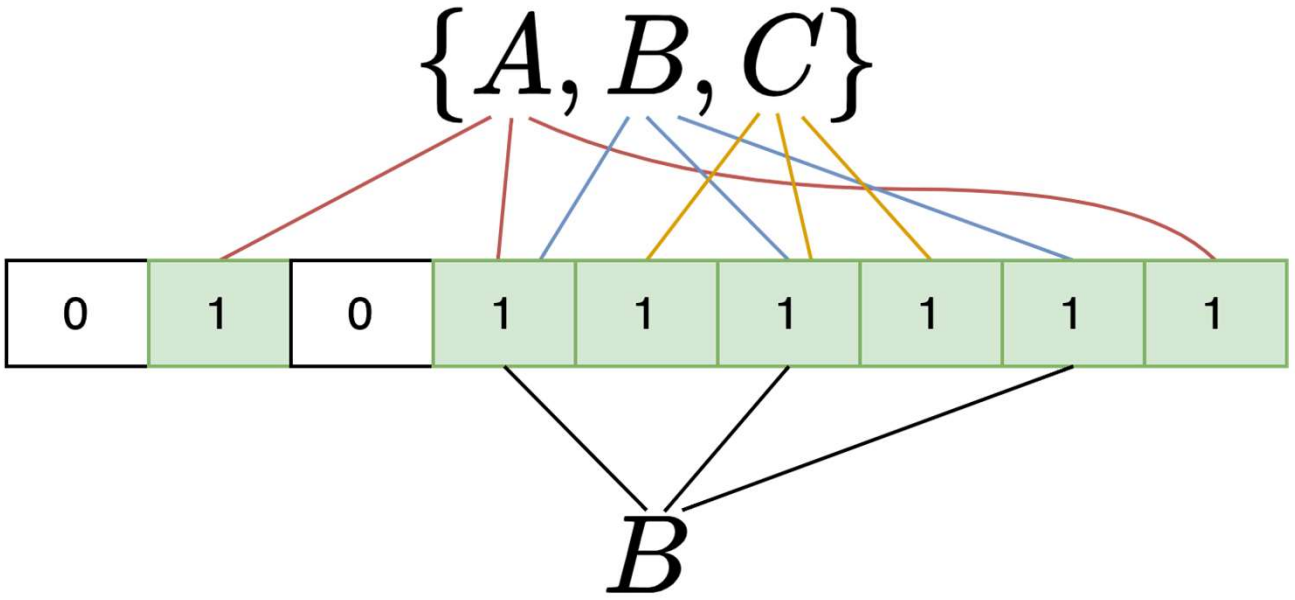
Фильтр Блума. Проверка принадлежности

Если хотя бы один бит равен нулю, то элемент точно не принадлежит множеству, иначе – **возможно** принадлежит.



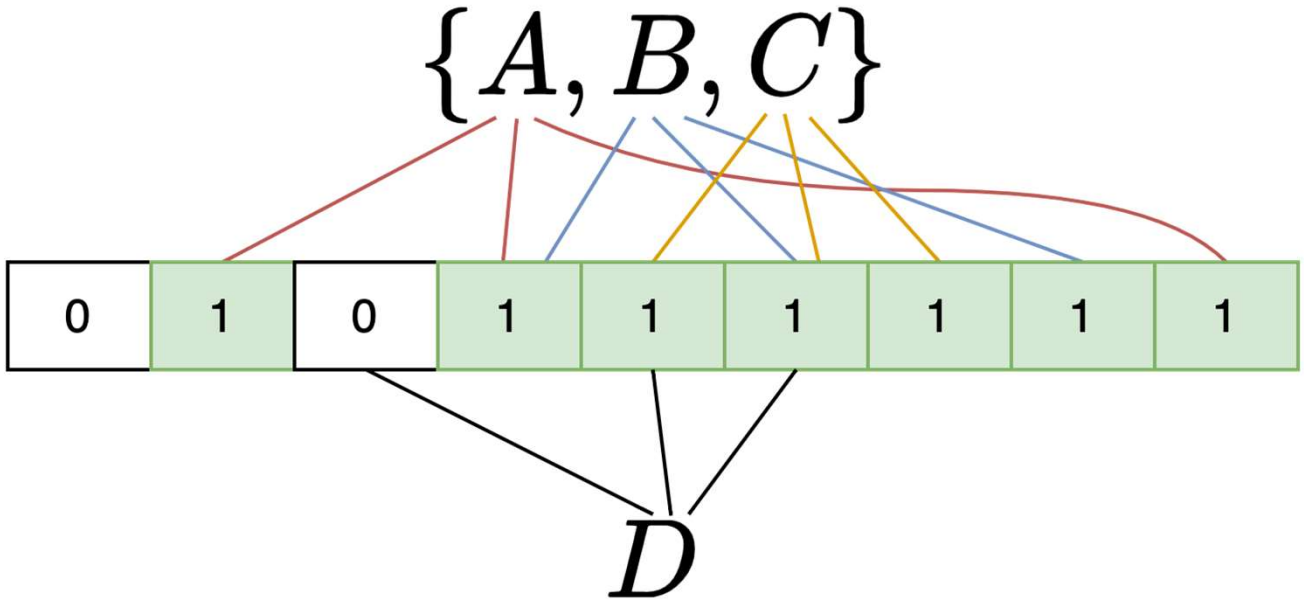
Фильтр Блума. Проверка принадлежности

Если хотя бы один бит равен нулю, то элемент точно не принадлежит множеству, иначе – **возможно** принадлежит.



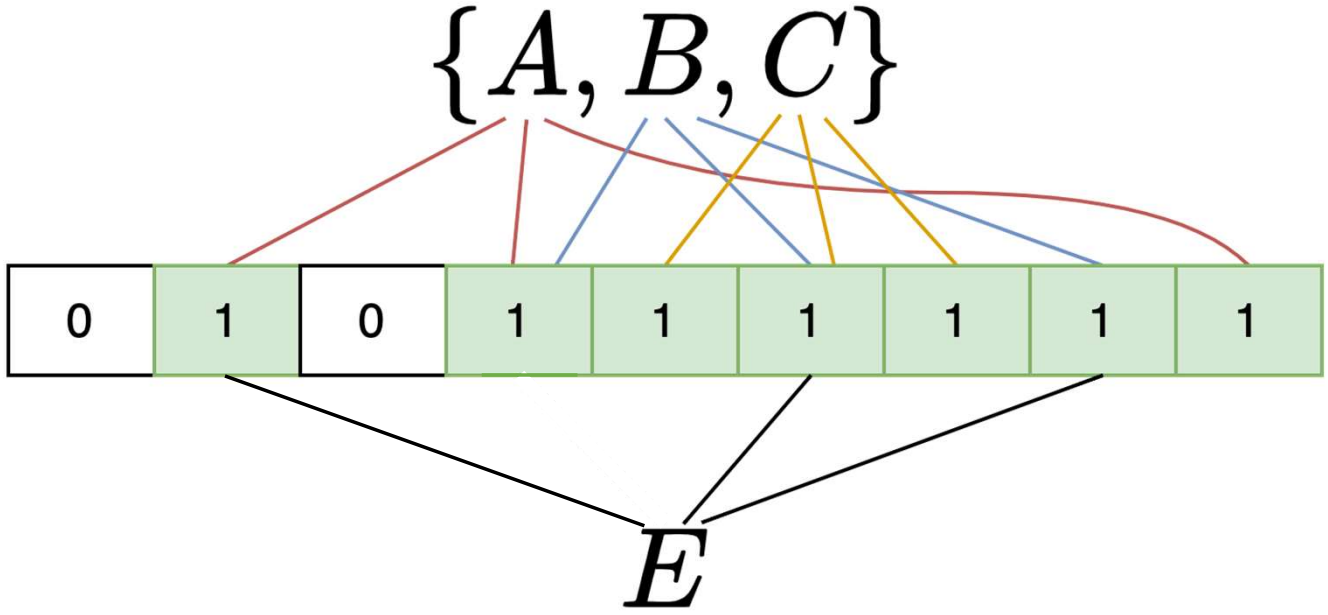
Фильтр Блума. Проверка принадлежности

Если хотя бы один бит равен нулю, то элемент точно не принадлежит множеству, иначе – **возможно** принадлежит.



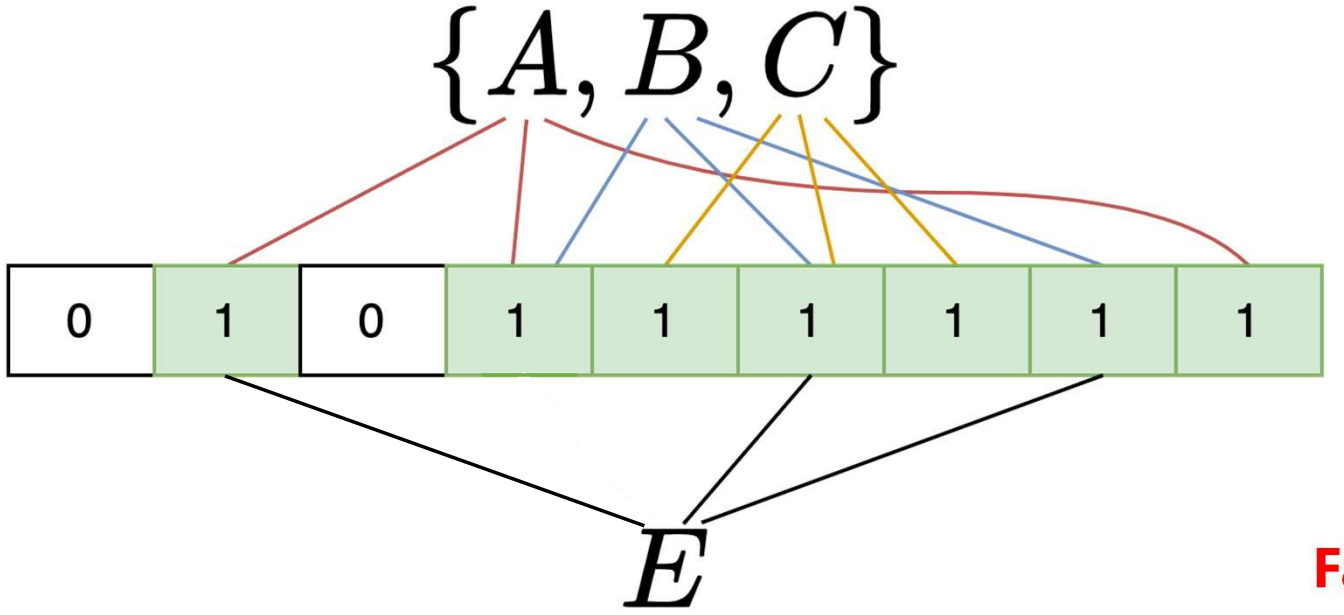
Фильтр Блума. Проверка принадлежности

Если хотя бы один бит равен нулю, то элемент точно не принадлежит множеству, иначе – **возможно** принадлежит.



Фильтр Блума. Проверка принадлежности

Если хотя бы один бит равен нулю, то элемент точно не принадлежит множеству, иначе – **возможно** принадлежит.



False positive!

Фильтр Блума. Минимизация ложный срабатываний

- Чем больше размер, тем меньше вероятность получения ложноположительного ответа о принадлежности.

- Оптимальное количество хеш-функций, которые минимизируют ложноположительные срабатывания (при известной мощности **n** исходного множества объектов):

$$\ln 2 \frac{m}{n} = 0.6931 \frac{m}{n}$$

Фильтр Блума

Что же он фильтрует?

Фильтр Блума

Что же он фильтрует?

Заведомо отсутствующие в исходном множеству объекты

- База данных Google BigTable
- Протокол интернет-кэширования
- Синхронизация кошельков в системе Bitcoin
- …

Фильтр Блума. Реализация

Хеш-функции

```
int hash1(string s, int arrSize) {
    long long hash = 1;
    for (size_t i = 0; i < s.size(); ++i) {
        hash += pow(19, i) * s[i];
        hash %= arrSize;
    return hash;
}
```

...

Проверка принадлежности

```
bool test(bool *a, int size, string s) {
    int h1 = hash1(s, size);
    ...;
    if (a[h1] && ...) return true;
    else return false;
}
```

Добавление элемента

```
void add(bool *a, int size, string s) {
    if (test(a, size, s)) return;
    else {
        int h1 = hash1(s, size);
        ...;
        a[h1] = true;
        ...;
    }
}
```

Вместо заключения

Элементарные структуры данных

1. **Массивы и матрицы**
   - Фиксированные и динамические
   - Итерация по указателям, цикл по диапазону
   - Интервальные запросы и корневая декомпозиция
2. **Стеки и очереди**
   - Дисциплины доступа к данным: LIFO и FIFO
   - Реализация очереди на массиве: кольцевой буфер
   - Очередь на двух стеках: ассоциативные операции
3. **Строки**: массивы **`char[]`** и объекты **`std::string`**

Рекурсивные структуры данных 1

1. **Список**
   - Последовательный доступ к данным
   - Односвязный, двусвязный, циклический
   - Алгоритм Флойда для поиска цикла
2. **Бинарное дерево**
   - Идеальное, строгое и полное дерево
   - Синтаксический разбор
   - Обход: в ширину и в глубину (три варианта)
   - Дерево поиска и повороты

Рекурсивные структуры данных 2

3. **Сбалансированные деревья поиска**
   - АВЛ-дерево: балансировка по высотам
   - Красно-черное дерево: балансировка по цветам
   - Splay-дерево: непостоянная сбалансированность
4. **B/B$^+$-деревья: блоки данных**
   - Плотное хранение упорядоченных данных
   - Многоуровневая индексация
   - Оптимизация доступа к медленной памяти

Специальные структуры 1

1. **Система непересекающихся множеств**
   - Хранение информации об объектах, помещенных в одно или несколько множеств
   - Проверка принадлежности и объединение множеств
2. **Хеширование и хеш-таблицы**
   - Отображение совокупности объектов на фиксированное множество хеш-кодов
   - Хеш-функция: коллизии
   - Метод цепочек, линейные сдвиги, кукушкино хеширование

Специальные структуры 2

3. **Список с пропусками**
   - Оптимизация последовательного доступа: многоуровневое хранение ключей
   - Вероятностный аналог сбалансированного дерева поиска
4. **Фильтр Блума**
   - Вероятностное множество: добавление и проверка принадлежности
   - Битовый массив и набор хеш-функций
   - Отсеивание заведомо отсутствующих объектов

Ну а дальше?...

Ну а дальше?...

А дальше STL!!!1!1!!