ТЕХНОСФЕРА

Обобщённый быстрый поиск. Хеш-функции.

Лекция 6

План лекции

1. Ещё раз об абстракции *Отображение*.
2. Обобщённый быстрый поиск.
3. Хеш-функции.
4. Применение хеш-функций.
5. Алгоритм Карпа-Рабина
6. Хеш-таблицы
7. Сочетание хеш-таблиц и деревьев.
8. Хеш-таблицы во внешней памяти.
9. Пример использования алгоритмов и структур данных.

Ещё раз об абстракции

Отображение

Абстракция *отображение*

Интерфейс абстракции *отображение* как ассоциативного массива.

- ***m[key] = value*** — добавить элемент с ключом *key* и значением *value*
- ***value = m[key]*** — найти элемент с ключом *key* и вернуть его.
- ***m[key] = nil*** — удалить элемент с ключом *key*
- ***for (auto x: m)*** — получить все ключи (или все пары ключ/значение) в каком-либо порядке.

Абстракция *отображение*

Алгоритм операция	Худшее время	Среднее время
BST: Вставка	$O(N \log N)$	$O(N \log N)$
BST: Поиск	$O(N \log N)$	$O(N \log N)$

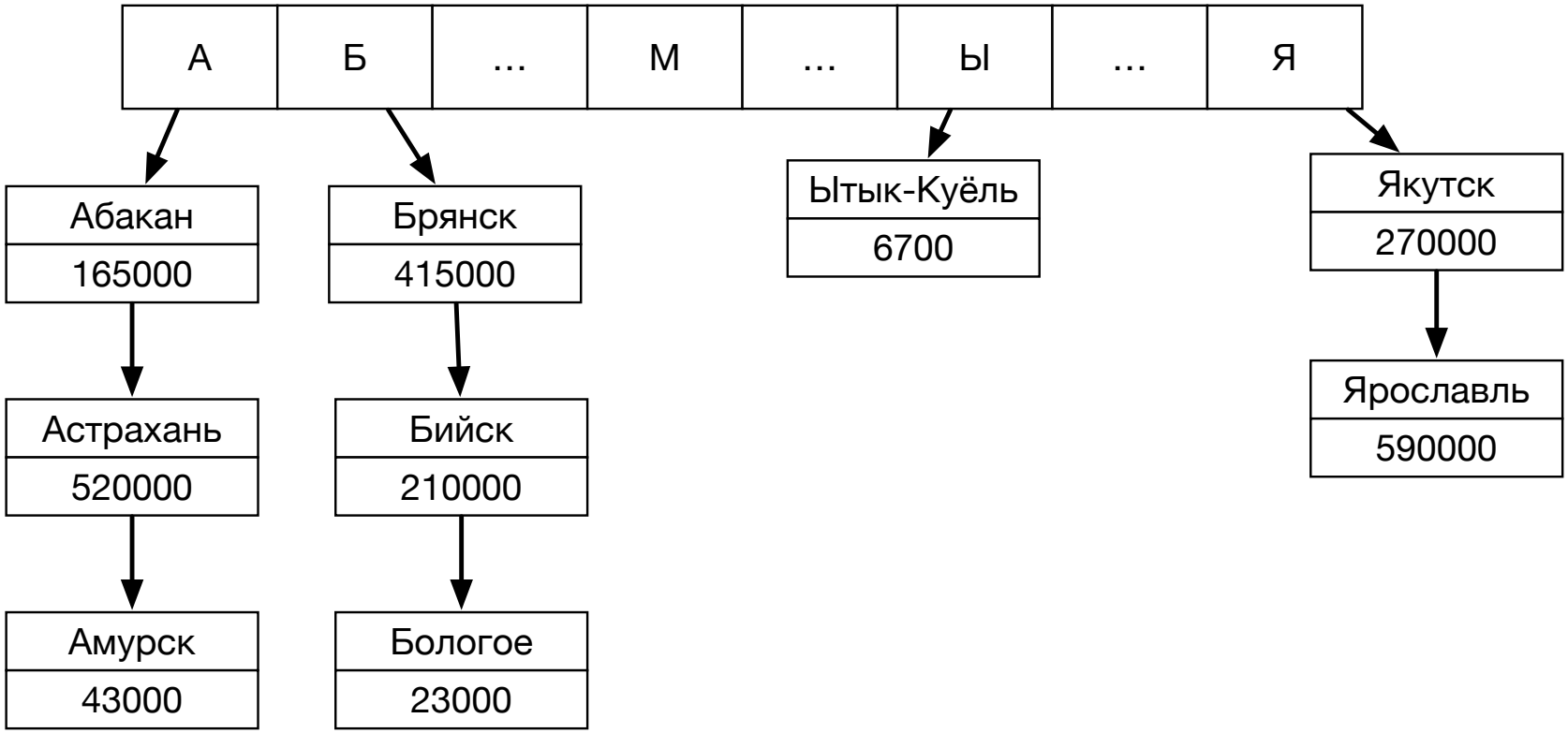
Обобщённый быстрый поиск

Обобщённый быстрый поиск

- Требуется:
  - Уменьшить амортизационную стоимость поиска
  - Уменьшить сложность функции (например, $O(\log N) \to O(\log \log N)$)

Обобщённый быстрый поиск

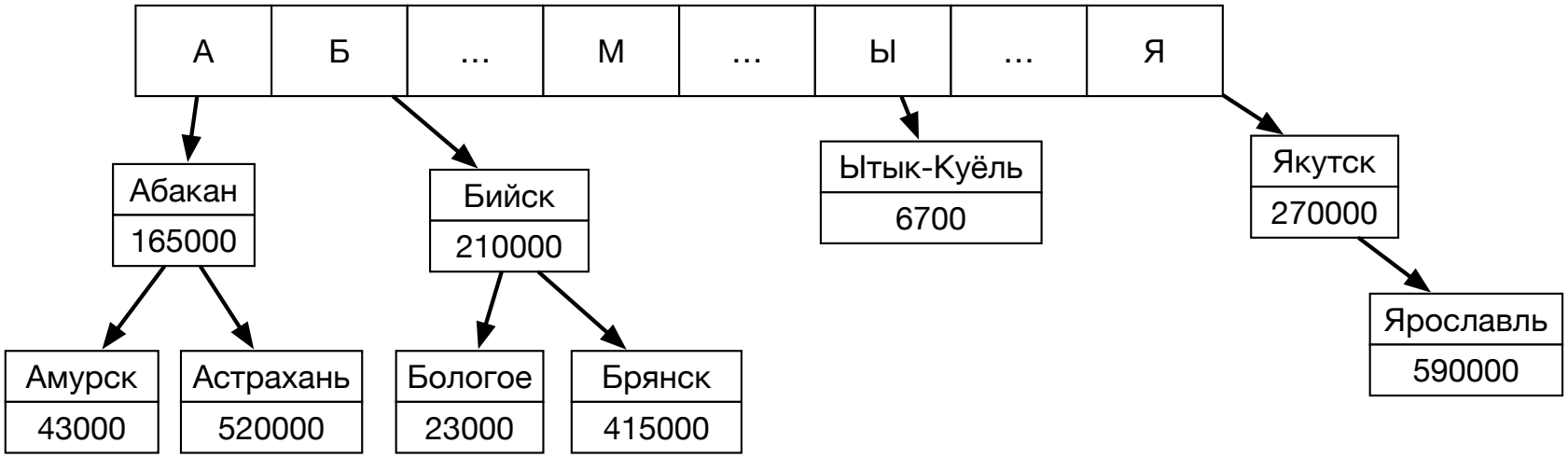
- База данных названий городов и их численности.



33 связных списка.

Обобщённый быстрый поиск

- База данных названий городов и численности их населения.



33 сбалансированных дерева.

Обобщённый быстрый поиск

- Основная идея — разбиение пространства ключей на независимые подпространства (*partitioning*).
- При независимом разбиении на M подпространств сложность уменьшается.

Для разбиения множества N ключей на примерно равные M подмножеств сложность вычисляется по главной теореме о рекурсии при числе подзадач M, коэффициенте размножения 1 и консолидации $O(1)$.

$$C \cdot O(N) \to \frac{C}{M} O(N)$$

$$C \cdot O(N \log N) \to \frac{C}{M} O(N \log N)$$

Обобщённый быстрый поиск

- При увеличении M

$$\lim_{K \to \infty} T(N, M) = O(1)$$

$$\lim_{K \to \infty} Mem(N, M) = \infty$$

- Имеется зона оптимальности при $M \approx N$

Обобщённый быстрый поиск

- Требуется иметь детерминированный способ разбиения пространства ключей на M независимых подпространств.
- Условия разбиения:

$$|K_1| \approx |K_2| \approx \cdots \approx |K_M|$$

$$\sum_{i=1}^{M} |K_i| = |K|$$

- Эврика! Создаём функцию $H(K)$, удовлетворяющую некоторым условиям.

Хеш-функции

Хеш-функции

- Функция преобразования:

$$H(K) \to V$$

$$|D(V)| = M$$

- Отображение пространства ключей K на пространство значений V.
- M — мощность множества пространства значений.

Хеш-функции

- Введём понятие *соперника*, то есть того, кто предоставляет нам ключи.
- Цель *соперника* — предоставлять ключи таким образом, чтобы значения функции оказались не равновероятными.
- *Соперник* знает хеш-функцию и может выбирать ключи.

Хеш-функции

Хотелось бы обеспечить свойства:

- **Эффективность.**
$$T(H(K)) \leqslant O(L(K)),$$
где $L(K)$ — мера длины ключа K.
- **Равномерность.** Каждое выходное значение равновероятно.
$$p_{H(K_1)} = p_{H(K_2)} = \cdots = p_{H(K_M)}$$
- **Лавинность.** При изменении одного бита во входной последовательности изменяется значительное число выходных битов.
- Для борьбы с *соперником* — **необратимость**, то есть невозможность восстановления ключа по значению его функции.

Хеш-функции

Следствия их требуемых свойств.

- Функция не должна быть непрерывной. Для близких значений аргумента должны получаться сильно различающиеся результаты.
- В значениях функции не должно образовываться *кластеров*, множеств близко стоящих точек.

Определение непрерывности для дискретных функций может быть дано неформально.

Хеш-функции

Примеры плохих функций:

- $H = K^2 \mod 10000$ для $K < 100$
  Функция монотонно возрастает. Пространство значений ключа слишком велико и часть значений недостижима.
- $H = \sum_{i=0}^{s.size()-1} s[i]$ для строки s.
  Функция даёт одинаковые значения для строк `abcd` и `abdc` и отличающиеся на единицу для строк `abcd` и `abde`. Сопернику легко найти ключи, которые дают равные значения функции.

Универсальная хеш-функция

- Совпадение значений функции для разных значений ключа называется **коллизией**
- Введём H^* — множество хеш-функций, которые отображают пространство ключей в $m = |D(M)|$ различных значений.
- Это множество **универсально**, если для каждой пары ключей $K_i, K_j, i \neq j$ количество хеш-функций, для которых $H^*(K_i) = H^*(K_j)$ не более $\frac{|H^*|}{m}$

Универсальная хеш-функция

- Если случайным образом выбирается функция из множества H^*, то для случайной пары ключей $K_i, K_j, i \neq j$ вероятность коллизии не должна превышать $\frac{1}{m}$

Теорема об универсальном множестве хеш-функций

Теорема.

- Пусть множество $Z_p = \{0, 1, \ldots, p-1\}$, множество $Z_p^* = \{1, 2, \ldots, p-1\}$, p — простое число, $a \in Z_p^*$, $b \in Z_p$.
- Тогда множество

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \mod p) \mod m\}$$

есть универсальное множество хеш-функций.

Теорема об универсальном множестве хеш-функций

Теорема.

- Пусть множество $Z_p = \{0, 1, \ldots, p-1\}$, множество $Z_p^* = \{1, 2, \ldots, p-1\}$, p — простое число, $a \in Z_p^*$, $b \in Z_p$.
- Тогда множество

$$H^*(p, m) = \{H(a, b, K) = ((aK + b) \mod p) \mod m\}$$

есть универсальное множество хеш-функций.

Доказательство. См. книгу Кормена.

Хеш-функции

- Не универсальная, не не столь уж и отвратительная функция

$$h = \sum_{i=0}^{n} s_i \times 8^i \mod HASHSIZE$$

Схема Горнера:

```
unsigned
hash_sum(string s, unsigned HASHSIZE)
{
    unsigned sum = 0;
    for (size_t i = 0; i < s.size(); i++) {
        sum <<= 3;
        sum += s[i];
    }
    return sum % HASHSIZE;
}
```

Хеш-функции

- Хеш-функция получше

```
unsigned
hash_sedgwick(string s, unsigned HASHSIZE)
{
    unsigned h, i, a = 31415, b = 27183;
    for (h = 0, i = 0; i < s.size();
      i++, a = a * b % (HASHSIZE-1)) {
        h = (a * h + *v) % HASHSIZE;
    }
    return h;
}
```

Хеш-функции

- Лучшие по статистическим показателям функции — криптографические.
- Недостатки:
  - длинный код
  - медленные

Хеш-функции

- Очень хорошие хеш-функции.
- Применяется полиномиальная арифметика или арифметика полей Галуа.
- В полях Галуа определены операции сложения и умножения.
- Пример: операции в поле $GF(2^3)$, оно состоит из чисел $0 \ldots 7$
- Операция сложение есть побитовое исключающее или XOR.

- Для умножения требуется число представить как полином.

$$5 = 101_2 = 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 = x^2 + 1$$

- Умножение чисел есть умножение полиномов.

$$5 * 5 = (x^2 + 1) \cdot (x^2 + 1) = x^4 + x^2 + x^2 + 1 = x^4 + 1 = 17$$

- Но ведь 17 не входит в поле $GF(2^3)$?
- Вводится понятие *производящий полином*, который должен быть *неприводимым*, то есть, не должен иметь полиномов-делителей, отличных от него и единицы.
- Один из таких полиномов для $GF(2^3)$ есть $x^3 + x + 1$.
- Результатом умножения будет остаток от деления $x^4 + 1$ на производящий полином $x^3 + x + 1$, что будет равно $x^2 + x + 1 = 7$

Хеш-функции

- $P(x)$ - исходное сообщение длины M битов
- $G(x)$ - производящий полином длины N битов
- $R(x)$ есть остаток от деления $P(x)$ на $G(x)$ в $GF(2^N)$
- Длина $R(x)$ ровно N битов.
- Если производящий полином $G(x)$ неприводим, то множество $R(x)$ имеет мощность 2^N.
  Для $GF(2^{32})$ один из неприводимых полиномов

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^6+x^4+x^2+x+1$$

Хеш-функции

- Примитивный член поля $GF(2^N)$ есть тот, степени которого содержат все ненулевые элементы поля.
- Алгоритм умножения чисел становится элементарным.
- Составляется таблица степеней примитивного члена.
- Например, для $GF(2^3)$ $2^6 = 5$. Тогда
  $5 \cdot 5 = 2^6 \cdot 2^6 = 2^{12} = 2^{12 \mod 7} = 2^5 = 7$
- Достаточно составить таблицу размером 2^3, содержащую требуемые значения.

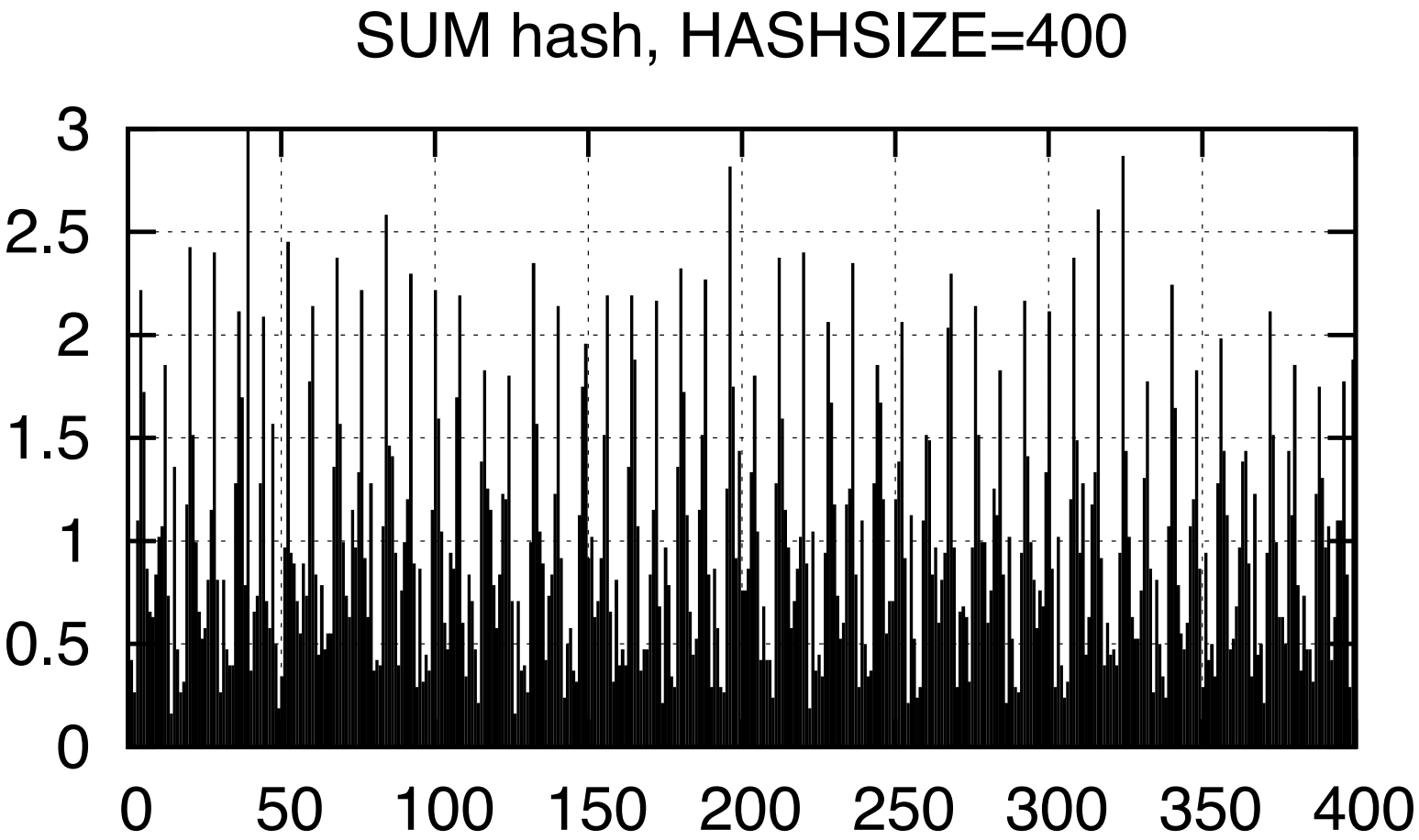
Хорошая хеш-функция

Сама функция для $GF(2^{32})$:

```
uint32 hash(uchar *ptr, unsigned length) {
    uint32 c = 0xFFFFFFFF;
    while (length) {
        c ^= (uint32) (ptr[0]);
        c = (c >> 8) ^ _table[c & 0xFF];
        ptr++;
        length--;
    }
    return c ^ 0xFFFFFFFF;
}
```

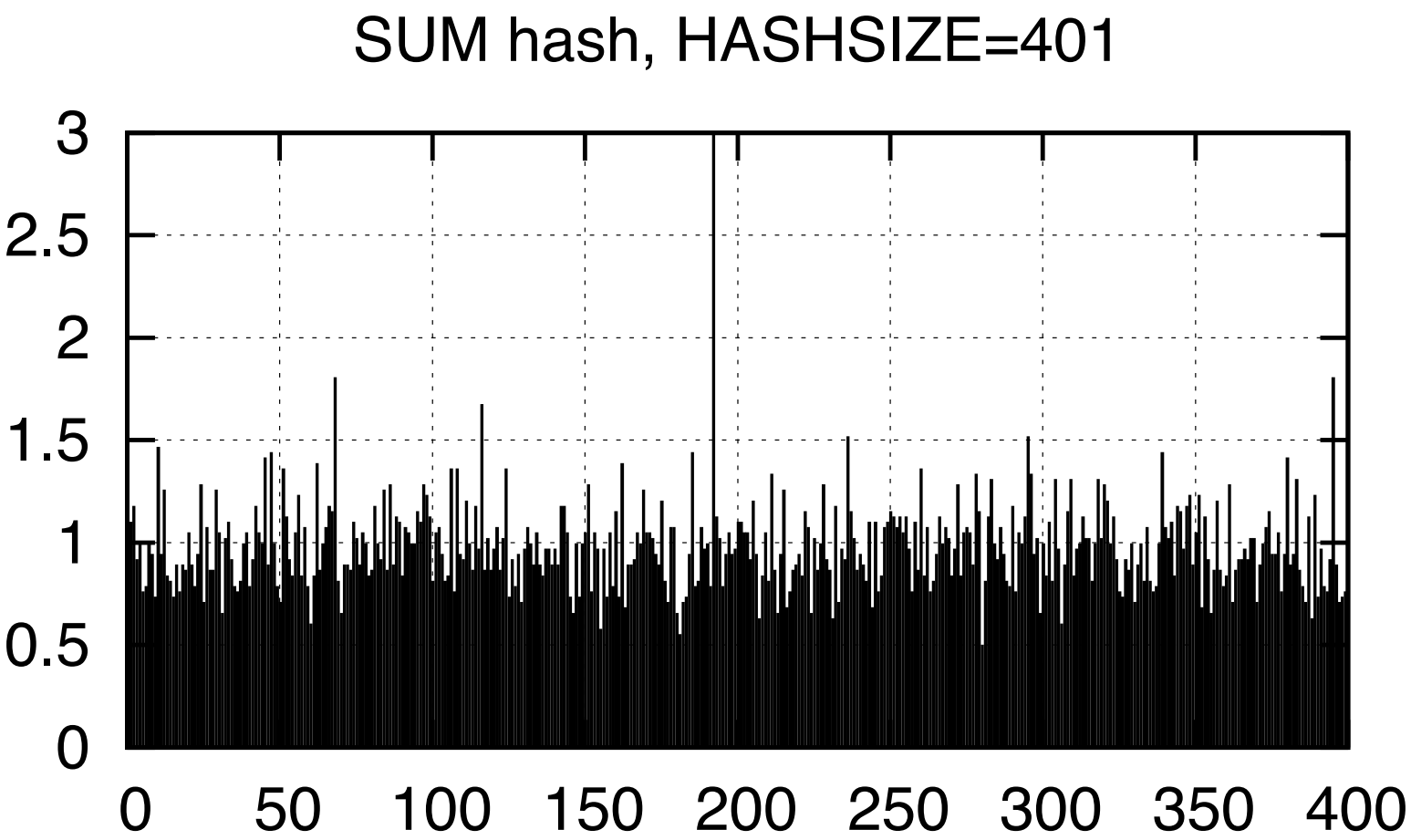
Хеш-функции: исследование свойств

Распределение значений для случайных идентификаторов. Плохая функция.



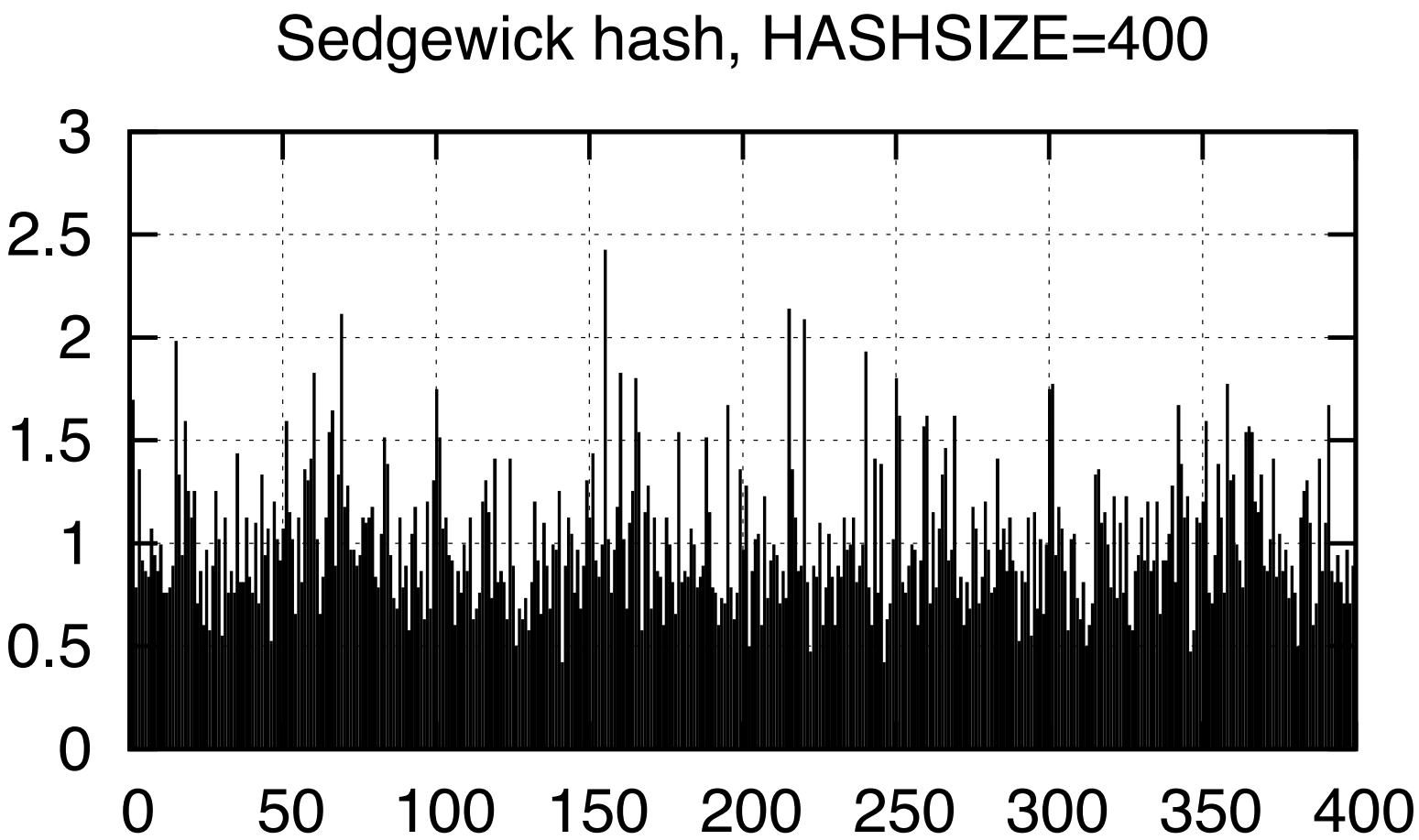
Хеш-функции

Распределение значений для случайных идентификаторов. Плохая функция.



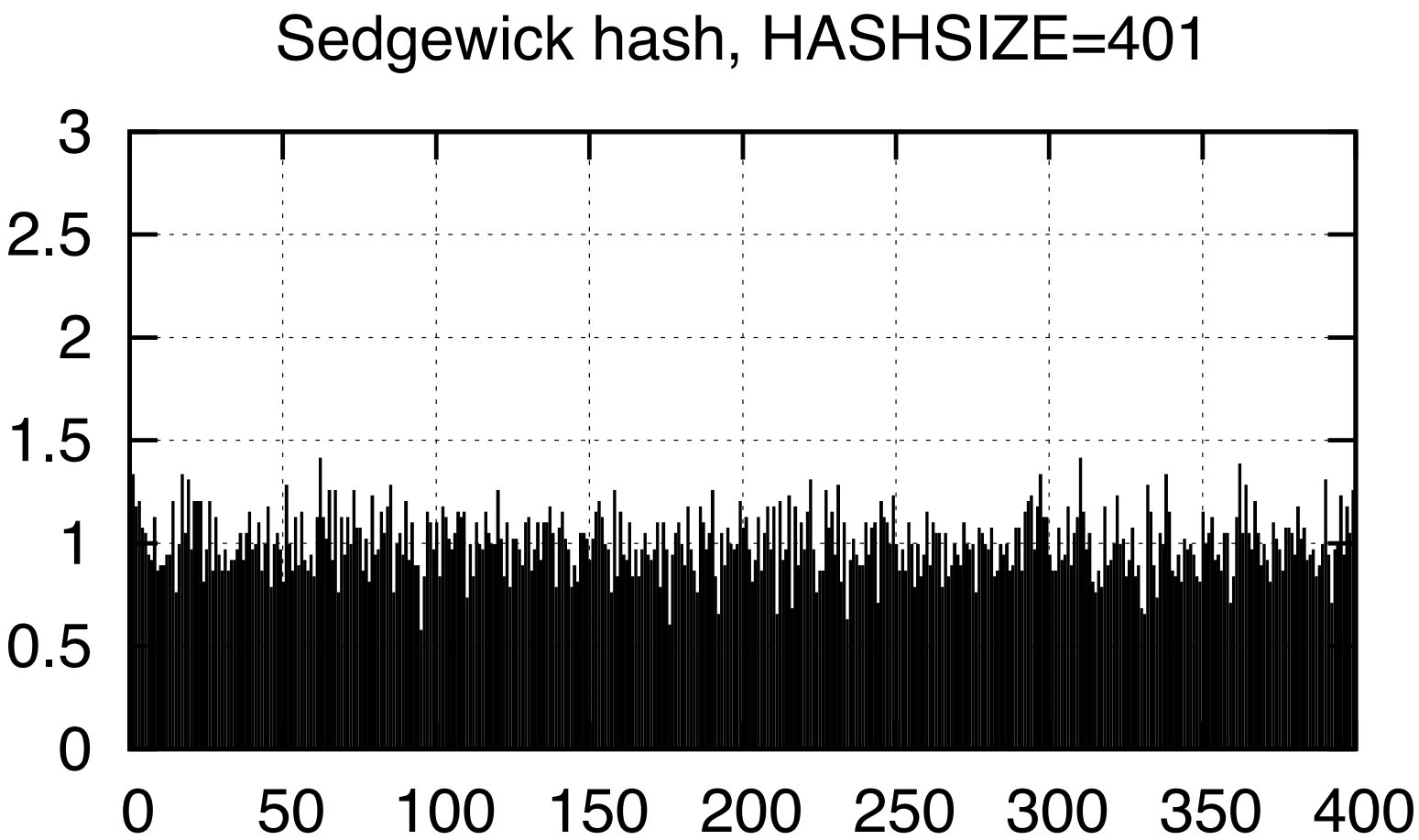
Хеш-функции

Хорошая функция.



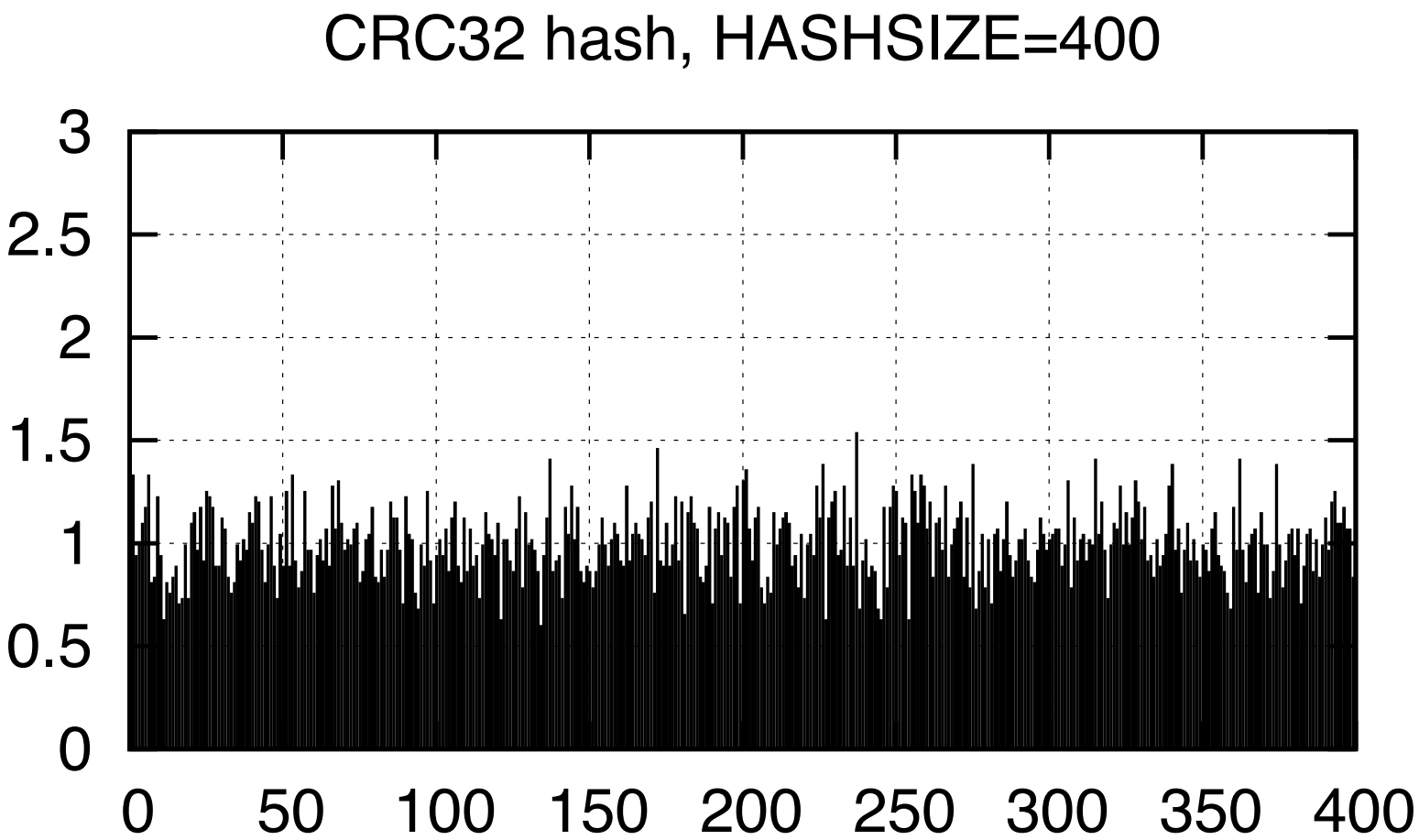
Хеш-функции

Хорошая функция.



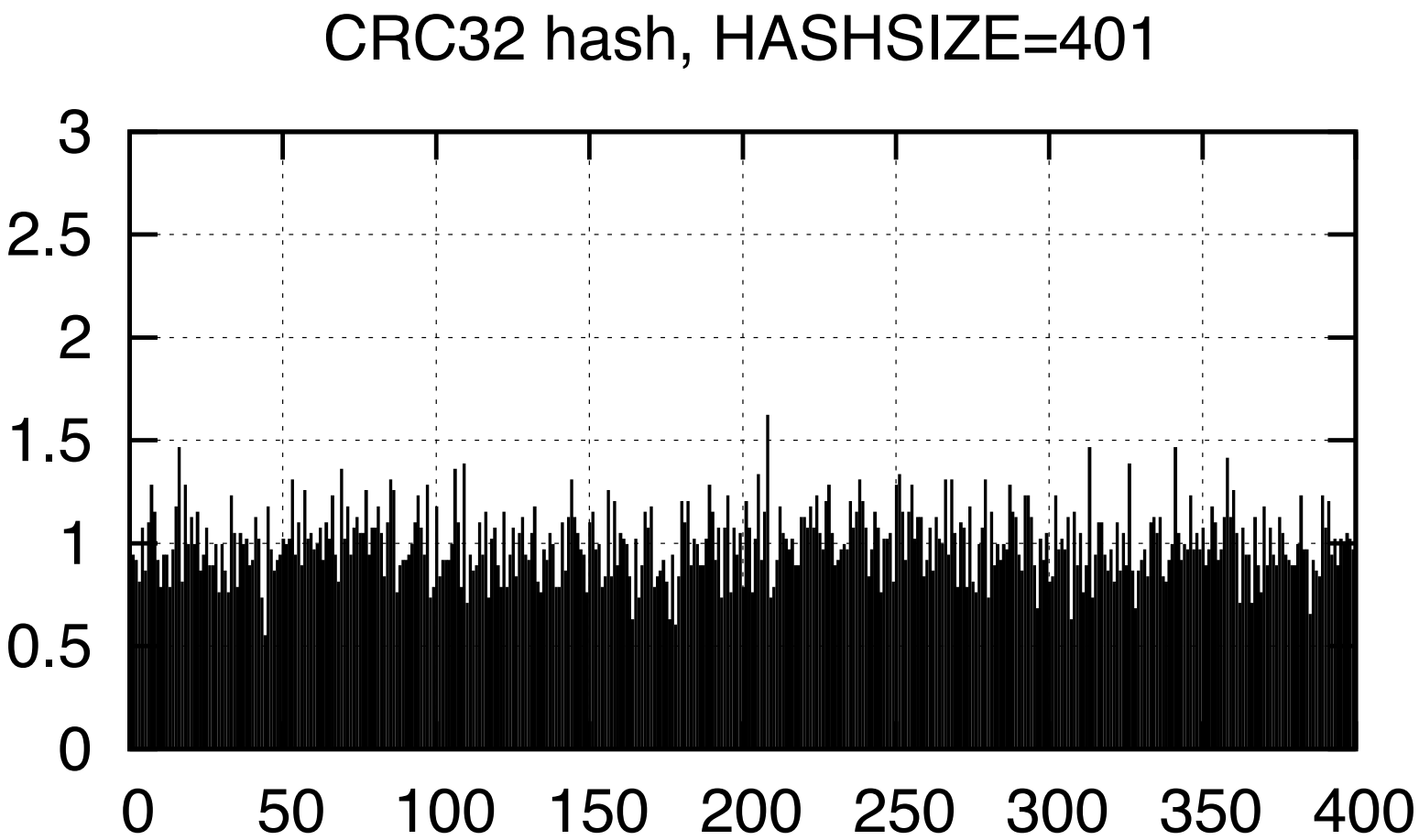
Хеш-функции

Отличная функция.



Хеш-функции

Отличная функция.



Затраты времени на исполнение хеш-функций

Алгоритм/набор	include.txt	source.txt
hash_sum	890	786
hash_sedgewick	2873	2312
hash_crc	912	801

Применение хеш-функций

Вероятностный подход к надёжности

Надёжны ли современные вычислительные системы?

- Производитель серверной памяти с коррекцией ошибок IBM измерил, что произошло 6 отказов на 10000 серверов за три года с 4ГБ памяти.
- Один отказ на 10^{20} обработанных байт.
- Сравним два блока памяти во 4096 байт. Вероятность получения неверного ответа при их равенстве есть $\frac{4096}{10^{20}} \approx 2.5 \cdot 10^{-16}$.
- Вероятность совпадения значений хорошей 64-битной хеш-функции для двух блоков данных размером в 4096 байт есть $\frac{4096}{2^{64}} = 2^{-52} \approx 10^{-17.1}$, то есть меньше!

Синхронизация больших объектов

Условия применения:

- Синхронизируемый объект имеет значительный размер
- Объект регулярно изменяет своё содержимое
- Размер изменяемой зоны относительно невелик

Обычное копирование расходует ресурс: пропускную способность.

Синхронизация больших объектов

Два паттерна использования:

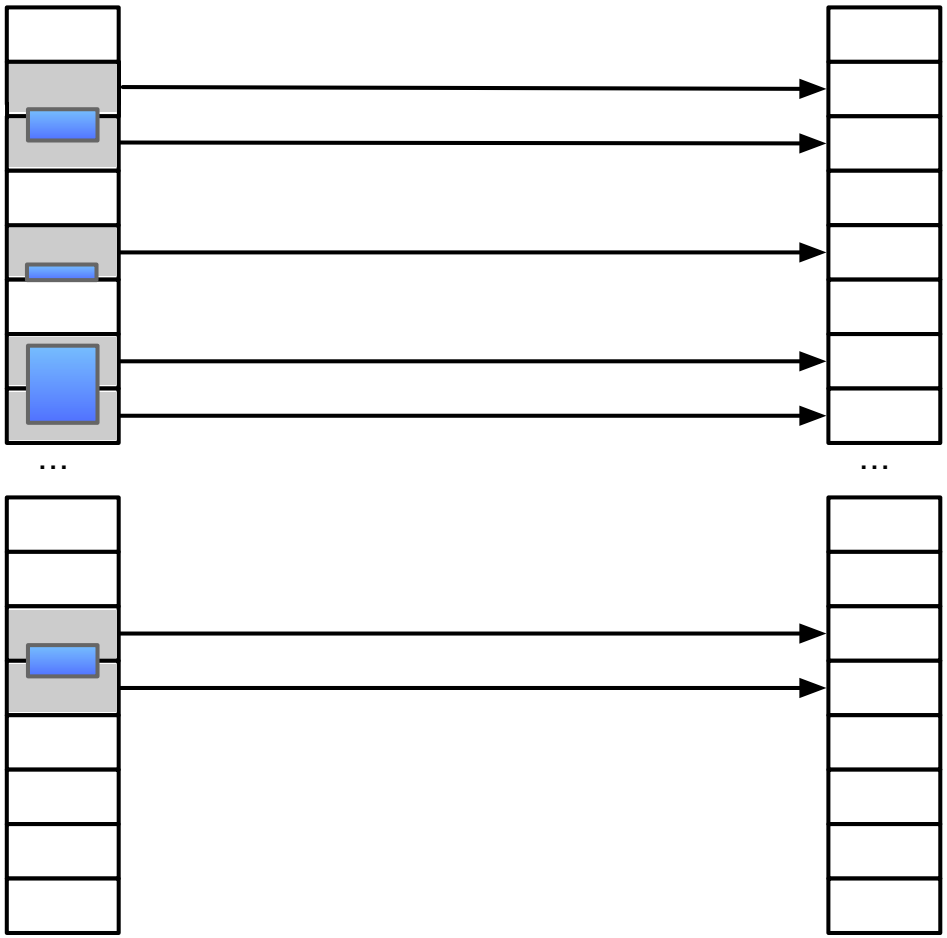
1. первичная пересылка объекта. Может потребовать передачи полного объёма.
2. пересылка изменённых фрагментов.

Синхронизация больших объектов: алгоритм

Задача: клиент синхронизирует большой объект с сервера.
Условия: на клиенте и сервере имеются реплики большого объекта, возможно, уже изменившегося на сервере. Используется одна и та же хеш-функция.

1. клиент и сервер разбивают объект на (виртуальные) блоки. Для каждого блока подсчитывается хеш.
2. клиент передаёт серверу номера блоков, для которых нужно вычислить хеш
3. сервер передаёт хеш запрошенных блоков
4. клиент сравнивает хеш и обнаруживает блоки с несовпадающем хешем
5. клиент запрашивает блоки с несовпадающем хешем

Синхронизация больших объектов



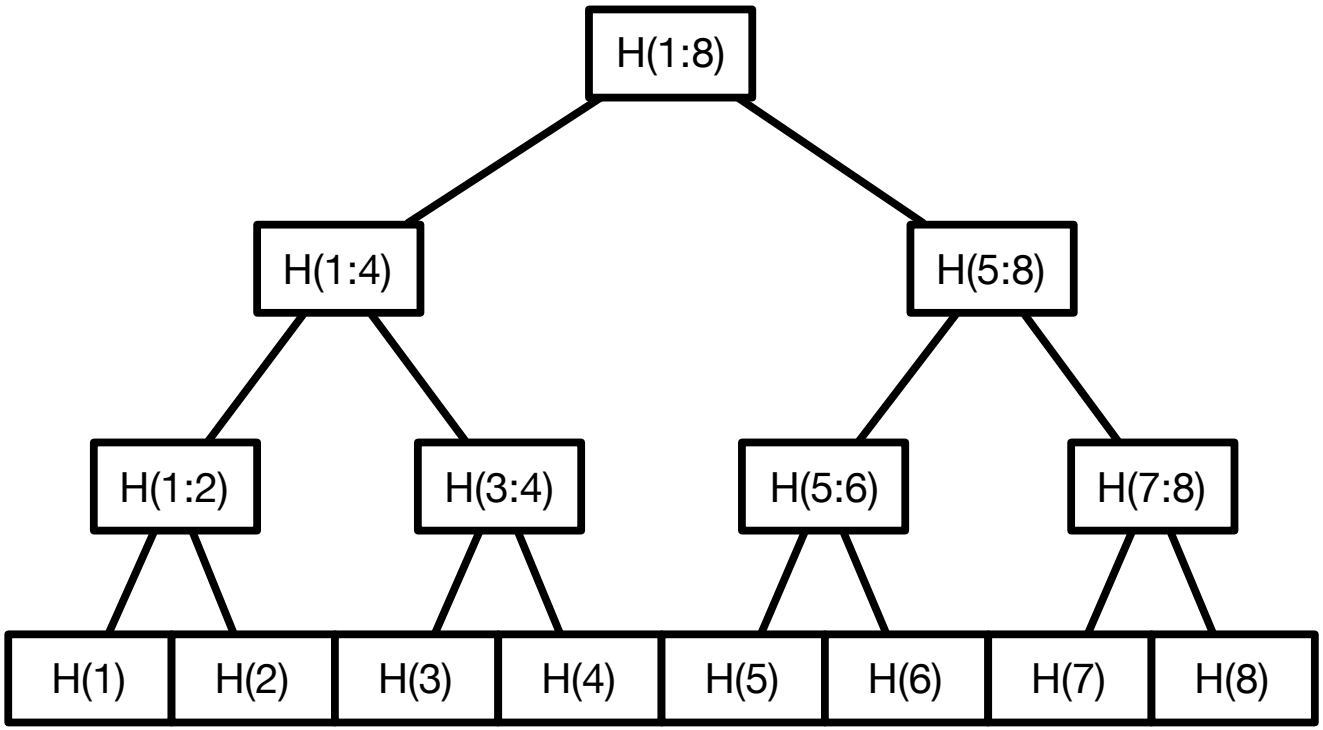
Синхронизация больших объектов: синий цвет — изменённые данные, серый — передаваемые блоки

Синхронизация больших объектов: продвинутый алгоритм

- Классические хеш-функции отображают множество ключей на множество значений.
- Зная значения функции для сообщений `A` и `B`, соответственно как `H(A)` и `H(B)` мы обычно не можем вычислить `H(AB)`, где `AB` — конкатенация сообщений `A` и `B`.
- *аддитивная* хеш-функция по `H(A)` и `H(B)` способна вычислить `H(AB)`.

Синхронизация больших объектов

Пусть синхронизируемый объект состоит из 2^N блоков.



Структура данных для усовершенствованной репликации

Синхронизация больших объектов

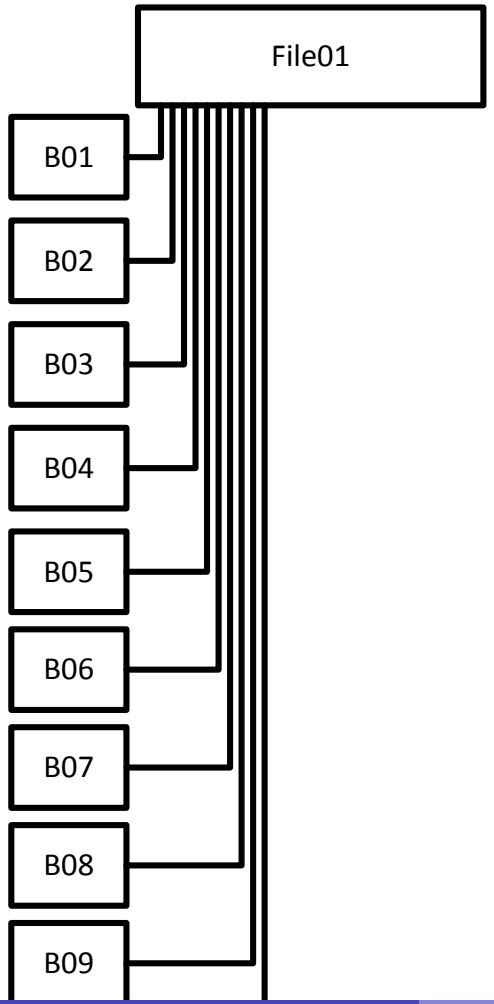
- Пусть `H(U:V)` — значение аддитивной хеш-функции от множества блоков от `U` до `V` включительно.
- Структура данных — дерево, на вершине которого находится узел, содержащий значение хеш-функции от всего объекта.
- Уровнем ниже — два узла, содержащие значения хеш-функции от половины объекта и так далее.
- Терминальные узлы содержат значения хеш-функции от отдельных блоков.
- Свойство аддитивности позволяет нам восстановить любой узел дерева по значению его потомков.

Синхронизация больших объектов: алгоритм синхронизации

- После первой фазы построения имеются деревьев с обоих сторон.
- Если значения хеш-функций для корня дерева совпали — алгоритм завершается.
- Рассматриваются потомки узла и спуск по дереву производится только в случае несовпадения значений хеш-функции на стороне оригинала и на стороне копии.

Дедупликация

- Имеется блочное хранилище.
- Пусть в хранилище имеется копия файла `File01`, состоящего из 10 блоков, от `B1` до `B10`.



Дедупликация

- Изменились блоки B3 и B7.
- Вариант 1: создать копию нового файла, содержащую все 10 блоков.
- 8 блоков — B1, B2 ... будут совпадать.
- Если есть возможность определить, что изменились именно блоки B3 и B7, то в хранилище достаточно передать именно эти блоки и заменить ими старые блоки B3 и B7.
- Старые блоки B3 и B7 сохраняются.
- Наличие новые блоков B3’ и B7’ позволит нам иметь два поколения файла.
- Соответствующими запросами можно будет извлечь две разных версии файла размером в 10 блоков, хотя в хранилище находятся только 12 блоков

Дедупликация

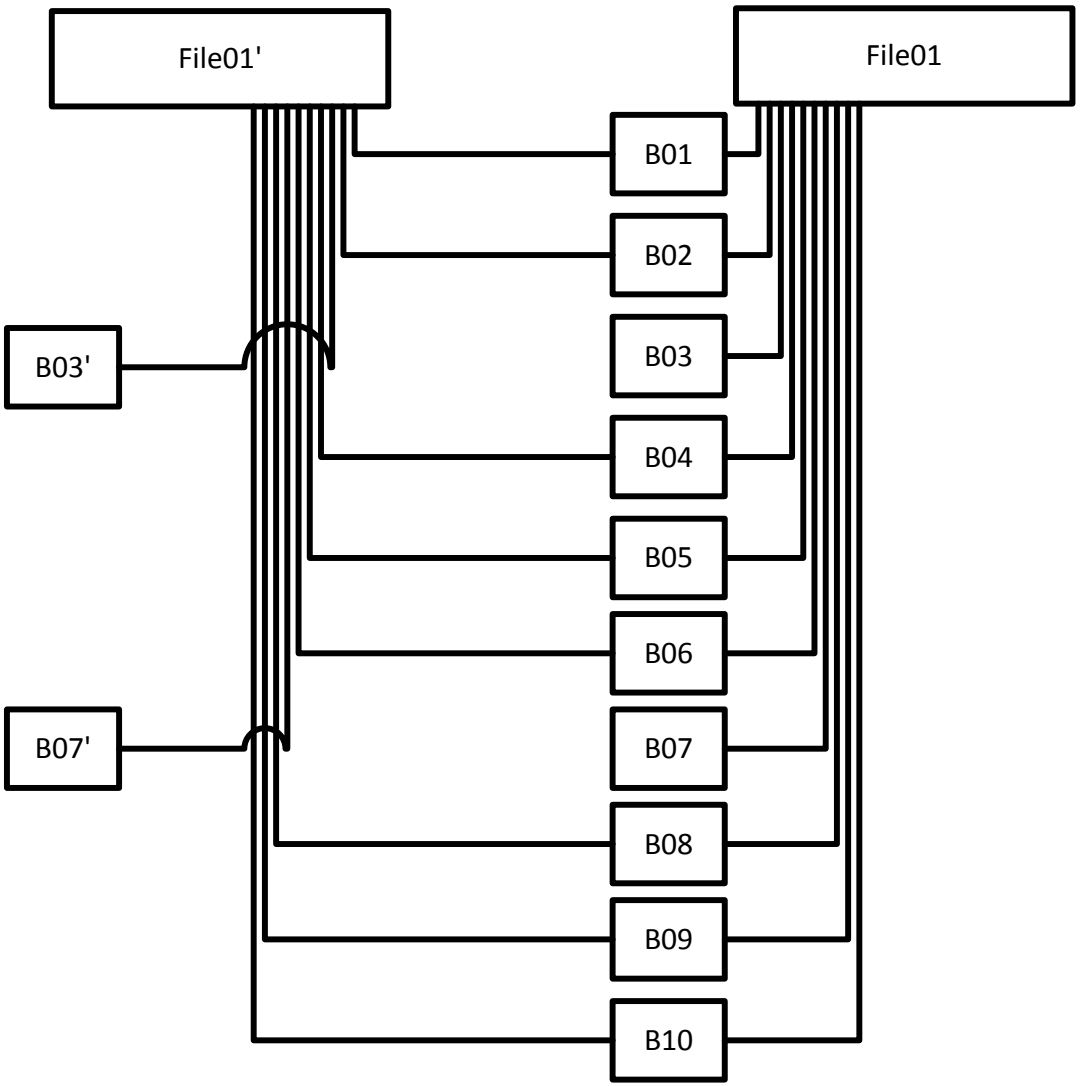


Схема хранения нескольких версий файла

Дедупликация

- Дедуплицированное хранилище содержит только уникальные блоки.
- Каждый блок при поступлении в хранилище проверяется на уникальность — имеется ли уже блок с таким содержимым.
- При совпадении блока в файле с уже имеющимся блоком, в карте хранения файла делается соответствующая запись.

Дедупликация: алгоритм

1. Определить множество блоков, участвующих в операции
2. Для каждого из блоков множества вычислить хеш
3. Если блок с таким хешем имеется в пуле, связать блок файла с блоком пула
4. Если блока с таким хешем не имеется, создать новый блок пула, связать блок файла с вновь созданным блоком пула

Дедупликация: проблемы

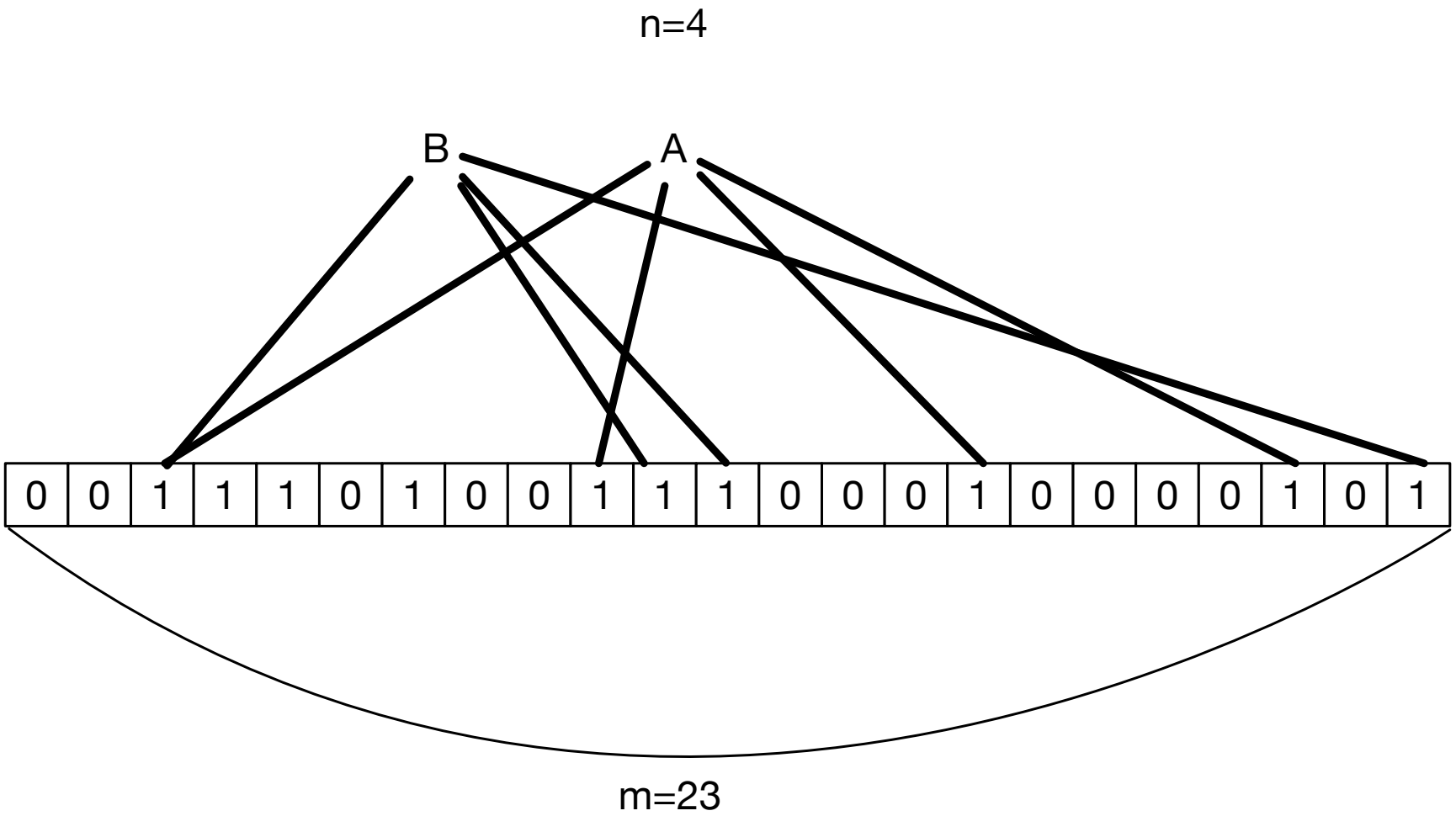
- Основная операция — поиск блока.
- Если блоков немного — создаётся отображение хешей блоков на реальное хранилище самих блоков.
- Подобное отображение — *персистентная* таблица.
- Серьёзное ограничение — размер таблицы.
- Оперативной памяти хватит лишь на миллиарды записей.
- Используют B-дерево (B+-дерево) или хеш-таблицу.
- Для уменьшения количества операций отображения применяют *вероятностные множества*.

Вероятностные множества

- *Вероятностное множество* реализует функциональность абстракции *множество*: операции `insert` и `find` с отсутствием гарантии точности результата поиска в этом множества.
- Результаты поиска могут быть *ложноположительными*, если элемент отсутствует, но операция `find` вернула истину.
- Отсутствие элемента всегда определяется точно, то есть, *ложноотрицательных* результатов быть не может.

Фильтр Блума

Реализация фильтра Блума: битовый массив из m бит и n различных хеш-функций $h_1, \ldots h_n$, равномерно отображающих элементы на номера битов.

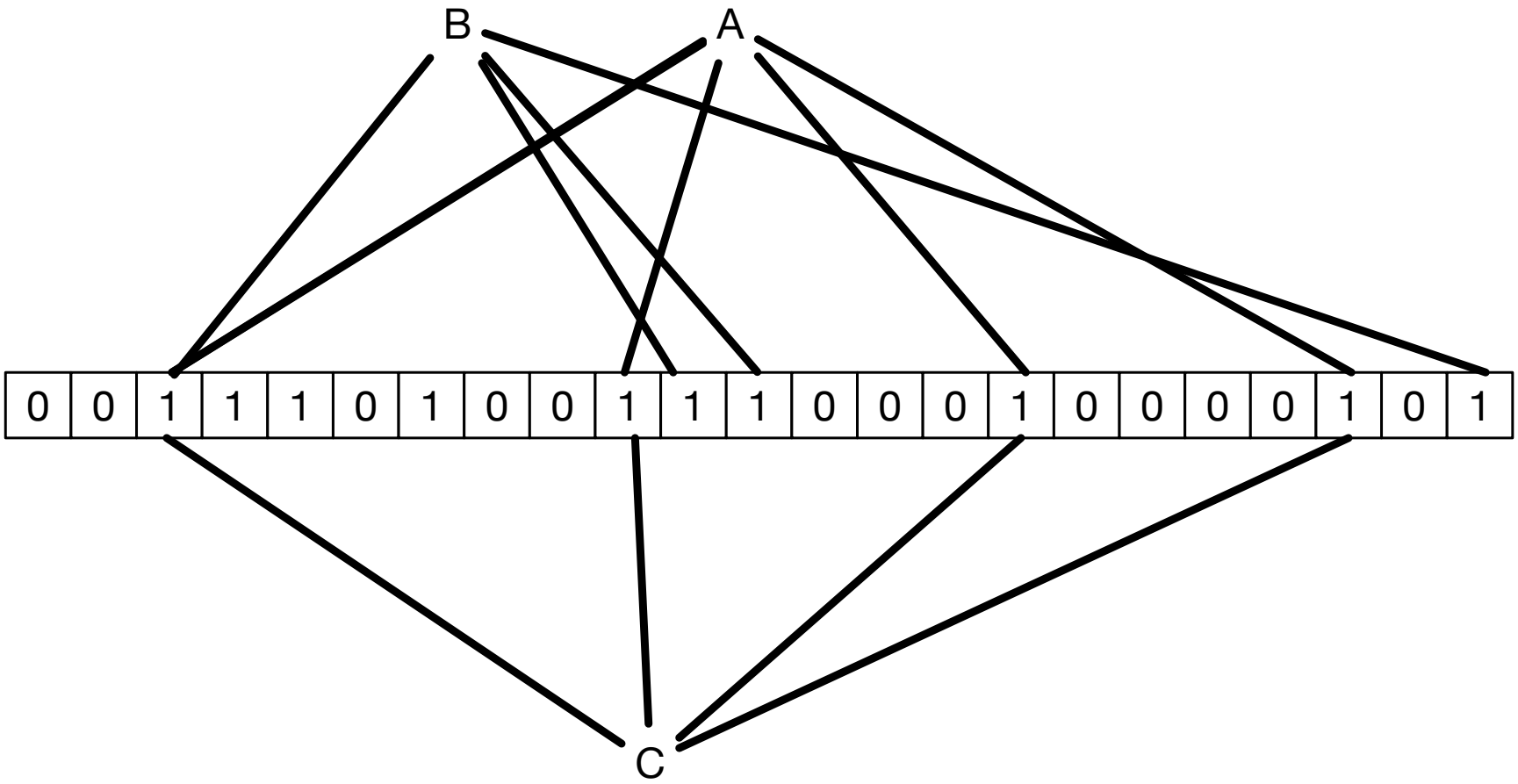


Фильтр Блума

- **create**: все биты равны нулю.
- **insert**: вычисляются все n хеш-функций и устанавливаются соответствующие биты.
- **find**: вычисляются все n хеш-функций.
  - Если хотя бы один бит не совпал, то ответ точен: **НЕТ.**
  - Если совпали все биты, то ответ: **МОЖЕТ БЫТЬ.**

Фильтр Блума

Для элементов A и C не равных друг другу все их хеши могут совпасть:



Фильтр Блума: свойства

- Это действительно фильтр, который помогает отсеять заведомо ненужные элементы.
- При добавлении элементов количество установленных битов увеличивается и его точность уменьшается.
- Предельный случай: все биты установлены. Любой элемент **«может быть»**.
- Оптимальное число хеш-функций для m битов и t элементов

$$b = \log_2 \frac{m}{t}$$

- Идеально приспособлен для уменьшения числа сложных операций (обращение к внешней памяти) и замене их более простыми (обращение к оперативной памяти).
- Операции удаления реализуются тяжело (требуется изменение представления).

Фильтр Блума: применение

- `Google Chrome`: По имени сайта быстрая проверка, не вредоносный ли он.
- `Google BigTable`: По заданной строке или столбцу базы данных определяет их наличие в таблице. Многократно уменьшает количество запросов к жёсткому диску за данными.
- `Распределённые системы хранения`: быстрое определение отсутствия требуемых данных.

Алгоритм Карпа-Рабина

Поиск подстрок в строке: алгоритм Карпа-Рабина

Имеется исходная строки и образец. Определить позицию в исходной строке, содержащую образец.

Упростим задачу.

Пусть строки состоят из символов `A`, `B`, `C`, `D`.
Отобразим их в 1, 2, 3, 4. Почему не с нуля? Чтобы различать `A` и `AAAA`.

Строка-образец — pat=`ABAC` или 1213.

Строка-источник — src=`ACABAACABACAABCA`

Поиск подстрок в строке: алгоритм Карпа-Рабина

A	B	A	C
1	2	1	3

A	C	A	B	A	A	C	A	B	A	C	A	A	B	C	A
1	3	1	2	1	1	3	1	2	1	3	1	1	2	3	1

Выберем *простое* число, немного превышающее мощность алфавита $P = 5$. Составим таблицу T степеней числа P по модулю 2^{32}

0	1	2	3	4	5	6	7	8	9
1	5	25	125	625	3125	15625	78125	390625	1953125

10	11	12	13	14	15
9765625	48828125	244140625	1220703125	1808548329	452807053

Алгоритм Карпа-Рабина

Хеш-функция от строки `S` в поддиапазоне `[k...r]`:

$$H(S_{[k,r]}) = \sum_{i=k}^{r} S_{i-k} \cdot P^{i-k} = \sum_{i=k}^{r} S_{i-k} \cdot T_{[i-k]}$$

Алгоритм Карпа-Рабина

Вычислим хеш-функцию для строки `pat` и от подстрок строки `src` длиной 4:

$$H\left(pat_{[0,3]}\right) = H(ABAC) = 0 \cdot 5^0 + 1 \cdot 5^1 + 0 \cdot 5^2 + 2 \cdot 5^3 = 411$$

$$\begin{aligned}
H\left(src_{[0,3]}\right) &= 291 \\
H\left(src_{[1,4]}\right) &= 183 \\
H\left(src_{[2,5]}\right) &= 161 \\
H\left(src_{[3,6]}\right) &= 407 \\
H\left(src_{[4,7]}\right) &= 206 \\
H\left(src_{[5,8]}\right) &= 291 \\
H\left(src_{[6,9]}\right) &= 183 \\
H\left(src_{[7,10]}\right) &= 411 \\
H\left(src_{[8,11]}\right) &= 207 \\
H\left(src_{[9,12]}\right) &= 166 \\
H\left(src_{[10,13]}\right) &= 283 \\
H\left(src_{[11,14]}\right) &= 431
\end{aligned}$$

Алгоритм Карпа-Рабина

Хеш-функция для наших строк:

```
unsigned hash(string s, unsigned l, unsigned r, unsigned *ptab) {
  unsigned sum = 0;
  for (unsigned i = l; i < r; i++) {
    sum += (s[i]-'A'+1) * ptab[i-l];
  }
  return sum;
}
```

Алгоритм Карпа-Рабина

Поиск подстроки:

```
  unsigned hs1 = hash(s1, 0, s1.size(), ptab);
  for (unsigned i = 0; i < s2.size() - s1.size(); i++) {
    unsigned hs2 = hash(s2, i, i+s1.size(), ptab);
    if (hs2 == hs1) {
        bool ok = true;
        for (unsigned j = 0; ok && j < s1.size(); j++) {
            if (s1[j] != s2[i+j]) {
                ok = false;
            }
        }
        if (ok) {
            printf("match at: %u\n", i);
        }
    }
}
```

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src.size()`, `M=pat.size()`?

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src.size()`, `M=pat.size()`?
$O(NM)$

Что не так?

Алгоритм Карпа-Рабина

Какова сложность кода при условии, что `N=src.size()`, `M=pat.size()`?
$O(NM)$

Что не так?
Мы делали много лишних вычислений.

Алгоритм Карпа-Рабина

$$H\left(s_{[0,4]}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3$$

Попробуем применить индукцию и вычислить $H\left(s_{[1,4]}\right)$.

$$H\left(s_{[1,5]}\right) = s_1 + s_2 \cdot p^1 + s_3 \cdot p^2 + s_4 \cdot p^3$$

Умножим на p^1:

$$H\left(s_{[1,5]}\right) \cdot p^1 = s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4$$

Сравним с

$$H\left(s_{[0,5]}\right) = s_0 + s_1 \cdot p^1 + s_2 \cdot p^2 + s_3 \cdot p^3 + s_4 \cdot p^4$$

$$H\left(s_{[k,l]}\right) \cdot p^k = H\left(s_{[0,l]}\right) - H\left(s_{[0,k]}\right)$$

Алгоритм Карпа-Рабина

Достаточно вычислить значения хеш-функции от всех подстрок строки `src`.

$$
\begin{aligned}
H\left(src_{[0,0]}\right) &= 0 \\
H\left(src_{[0,1]}\right) &= 1 \\
H\left(src_{[0,2]}\right) &= 16 \\
H\left(src_{[0,3]}\right) &= 41 \\
H\left(src_{[0,4]}\right) &= 291 \\
H\left(src_{[0,5]}\right) &= 916 \\
H\left(src_{[0,6]}\right) &= 4041 \\
H\left(src_{[0,7]}\right) &= 50916 \\
H\left(src_{[0,8]}\right) &= 129041 \\
H\left(src_{[0,9]}\right) &= 910291 \\
H\left(src_{[0,10]}\right) &= 2863416 \\
H\left(src_{[0,11]}\right) &= 32160291 \\
H\left(src_{[0,12]}\right) &= 80988416 \\
&\dots
\end{aligned}
$$

Алгоритм Карпа-Рабина

```
int karp_rabin(string s1, string s2, vector<unsigned> ptab) {
    unsigned hs1 = hash(s1, 0, s1.size(), ptab);
    vector<unsigned> htab(s2.size());
    for (unsigned i = 1; i < s2.size(); i++)
        htab[i] = htab[i-1] + (s2[i-1] - 'A' + 1)*ptab[i-1];
    for (unsigned i = 0; i < s2.size() - s1.size(); i++) {
        unsigned hs2 = htab[i+s1.size()] - htab[i];
        if (hs2 == hs1) {
            bool ok = true;
            for (unsigned j = 0; j < s1.size(); j++)
                if (s1[j] != s2[i+j])
                    ok = false;
            if (ok)
                return i;
        }
        hs1 *= 5;
    }
    return -1;
}
```

Алгоритм Карпа-Рабина

Применённая здесь функция имеет свойство *rolling-hash* и можно идти другим путём.

Если существуют такие p и x, что уравнение

$$p \cdot x = 1 \pmod{M}$$

имеет решение, то элемент x является обратным элементом для p в кольце вычетов по модулю M.

Необходимое условие: $\gcd(p, m) = 1$.
Малая теорема Ферма: для простого m и любого $p \in \mathbb{N}$:, p не делится на m,

$$p^{m-1} \equiv 1 \pmod{m}$$

Соответственно, $p \cdot p^{m-2} \equiv 1 \pmod{m}$, $x = p^{m-2} \pmod{m}$

Алгоритм Карпа-Рабина

При соблюдении условий можно заменить деление на p умножением на p^{-1}. В нашей задаче $p = 5$, $m = 2^{32}$, $p^{-1} \pmod{m} = 3435973837$. Что выведет программа:

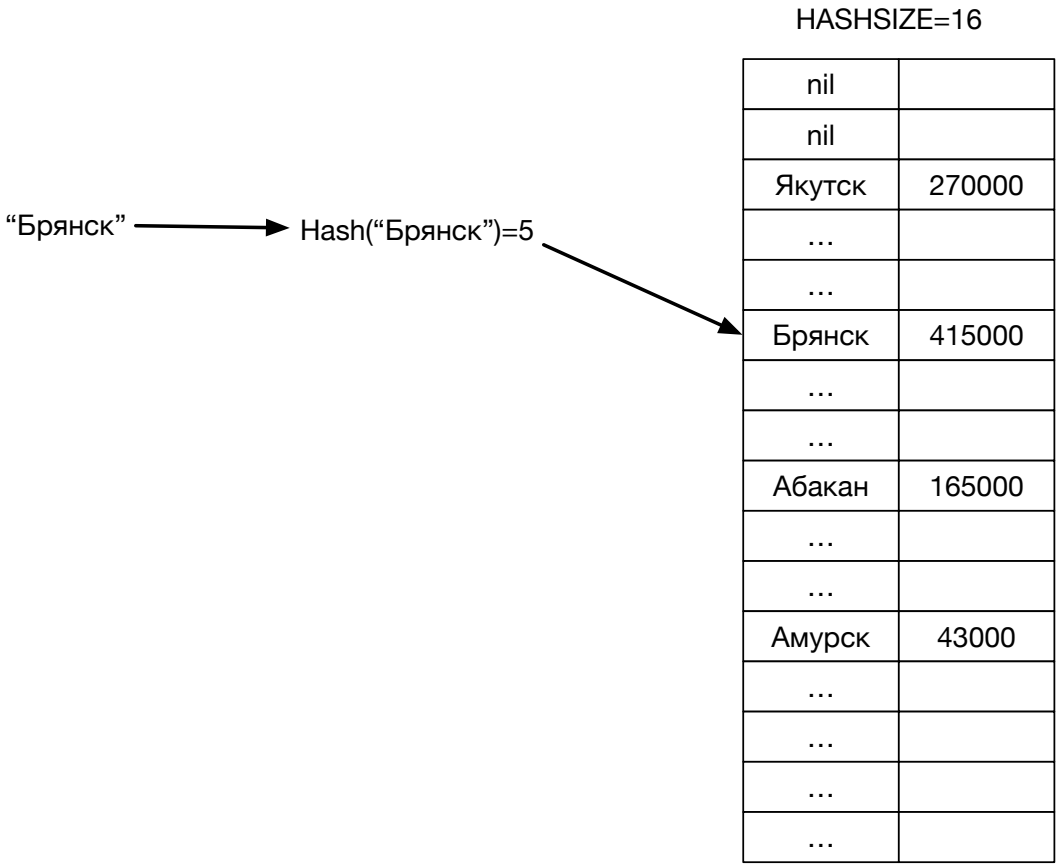
```
#include <stdio.h>

int main() {
    for (unsigned x = 5; x < 1000; x += 5) {
        printf("%u\n", x * 3435973837u);
    }
}
```

Хеш-таблицы

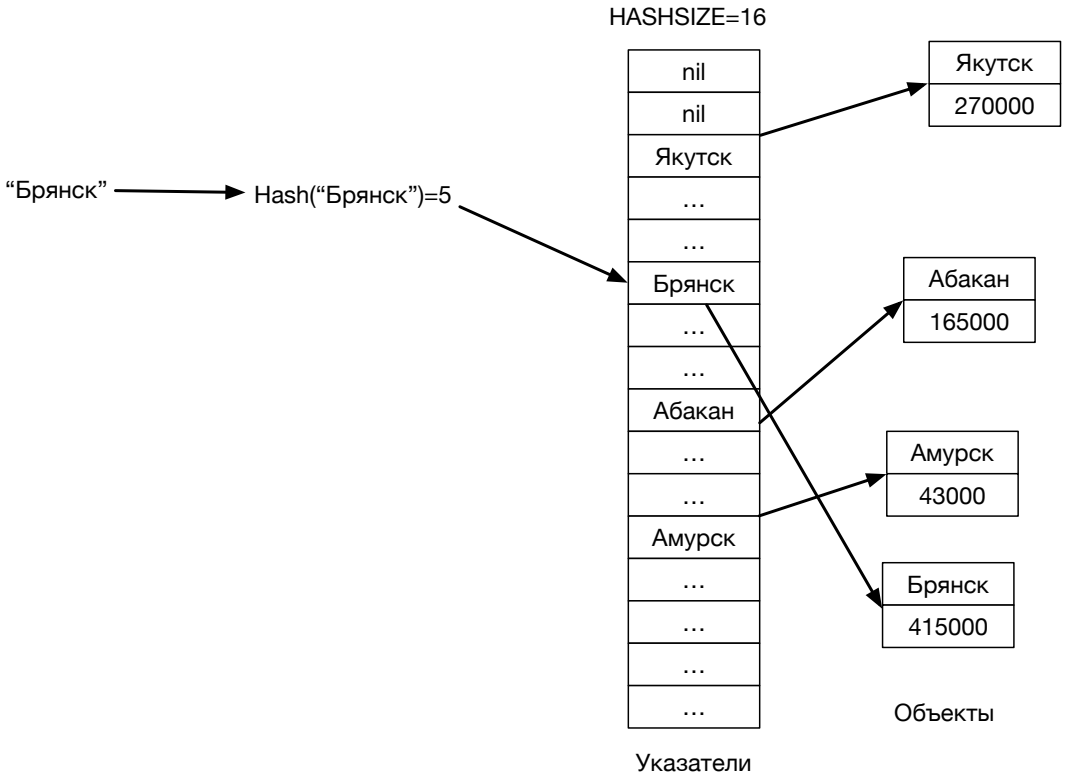
Хеш-таблицы

- Простая хеш-таблица



Хеш-таблицы

- Простая хеш-таблица, обычная реализация в виде массива указателей

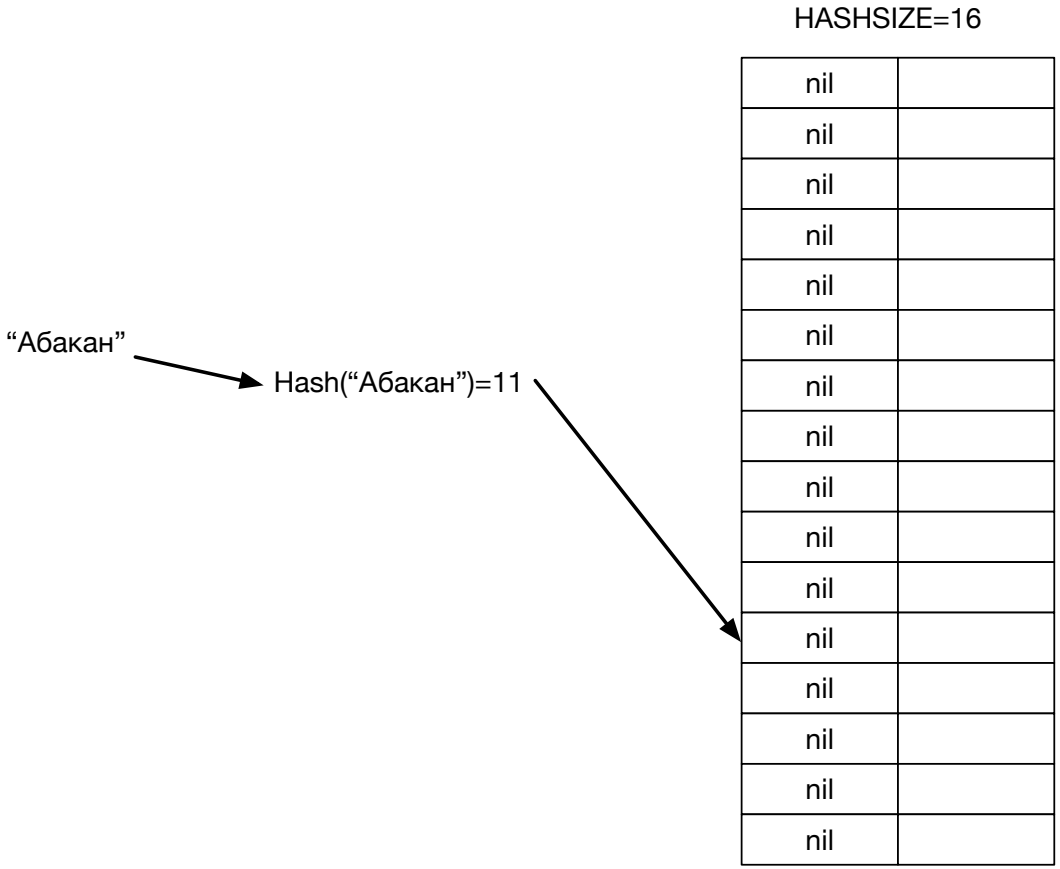


Хеш-таблицы

- Известно количество элементов в контейнере C
- Известен размер массива M
- $\alpha = \frac{C}{M}$ — коэффициент заполнения, *fill-factor*, *load-factor*.
- α — главный показатель хеш-таблицы.

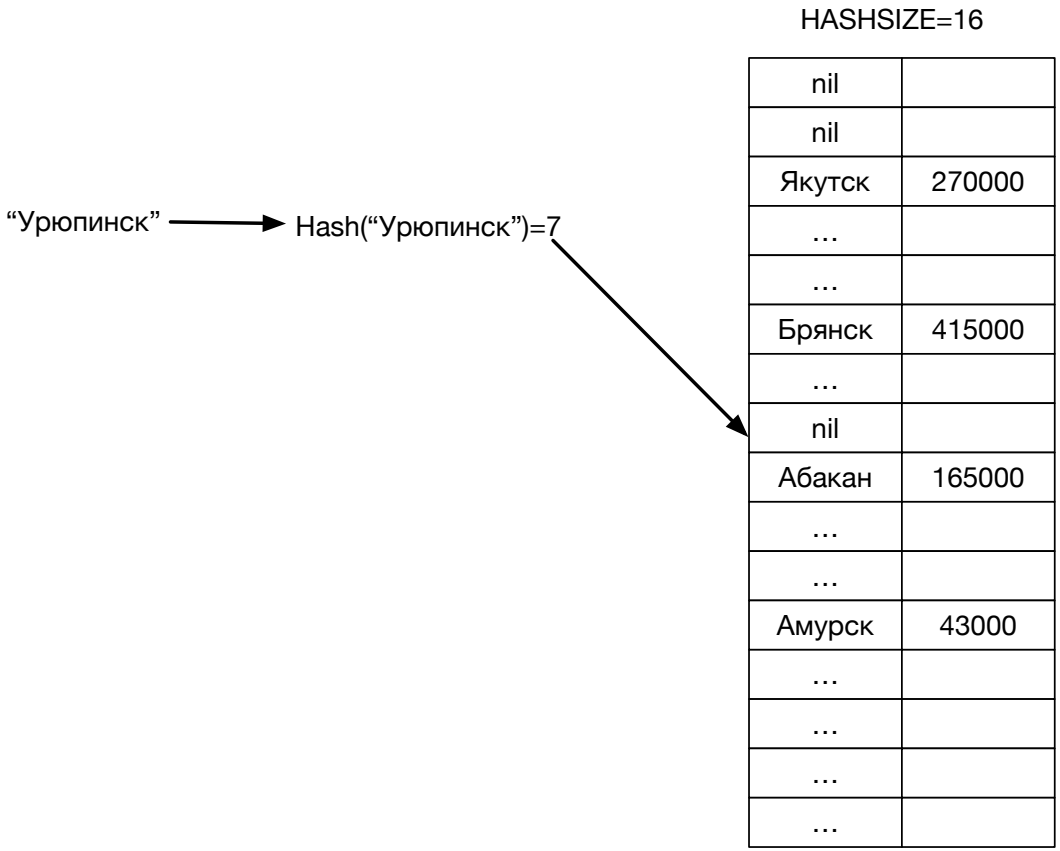
Хеш-таблицы

- Операция создания хеш-таблицы



Хеш-таблицы

- Операция создания хеш-таблицы требует операцию поиска.

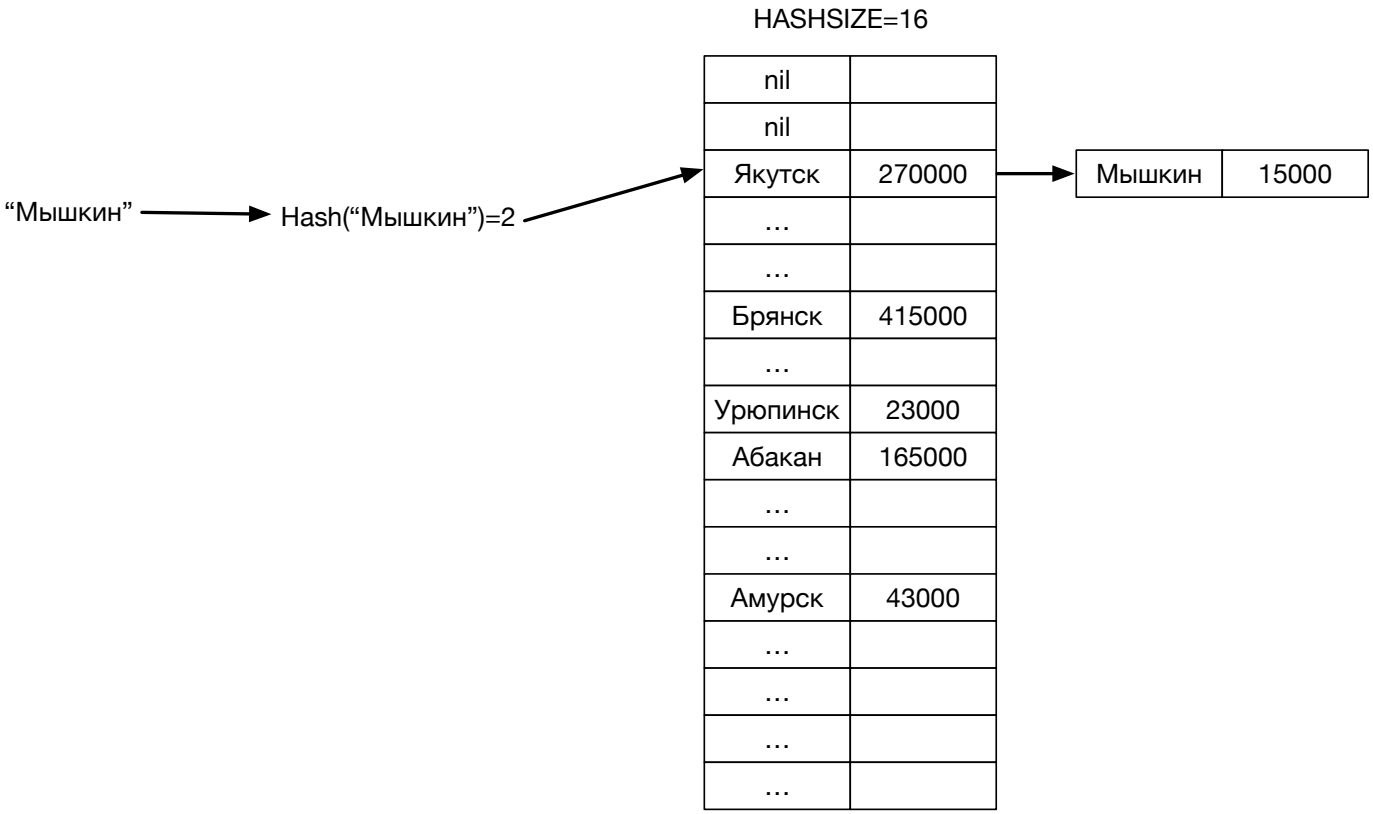


Хеш-таблицы

- Hash("Якутск") = 2
- Hash("Мышкин") = 2
- Это — *коллизия*
- Коллизии — нежелательны.
- Без коллизий сложность операций поиска и вставки равна $O(1)$
- Способы борьбы с коллизиями:
  - Прямая или закрытая адресация
  - Открытая адресация
  - Рехеширование

Хеш-таблицы с прямой адресацией

- При коллизии во время создания элемента создаётся связный список конфликтующих.
- Технически можно создать любую поисковую структуру данных



Хеш-таблицы с прямой адресацией

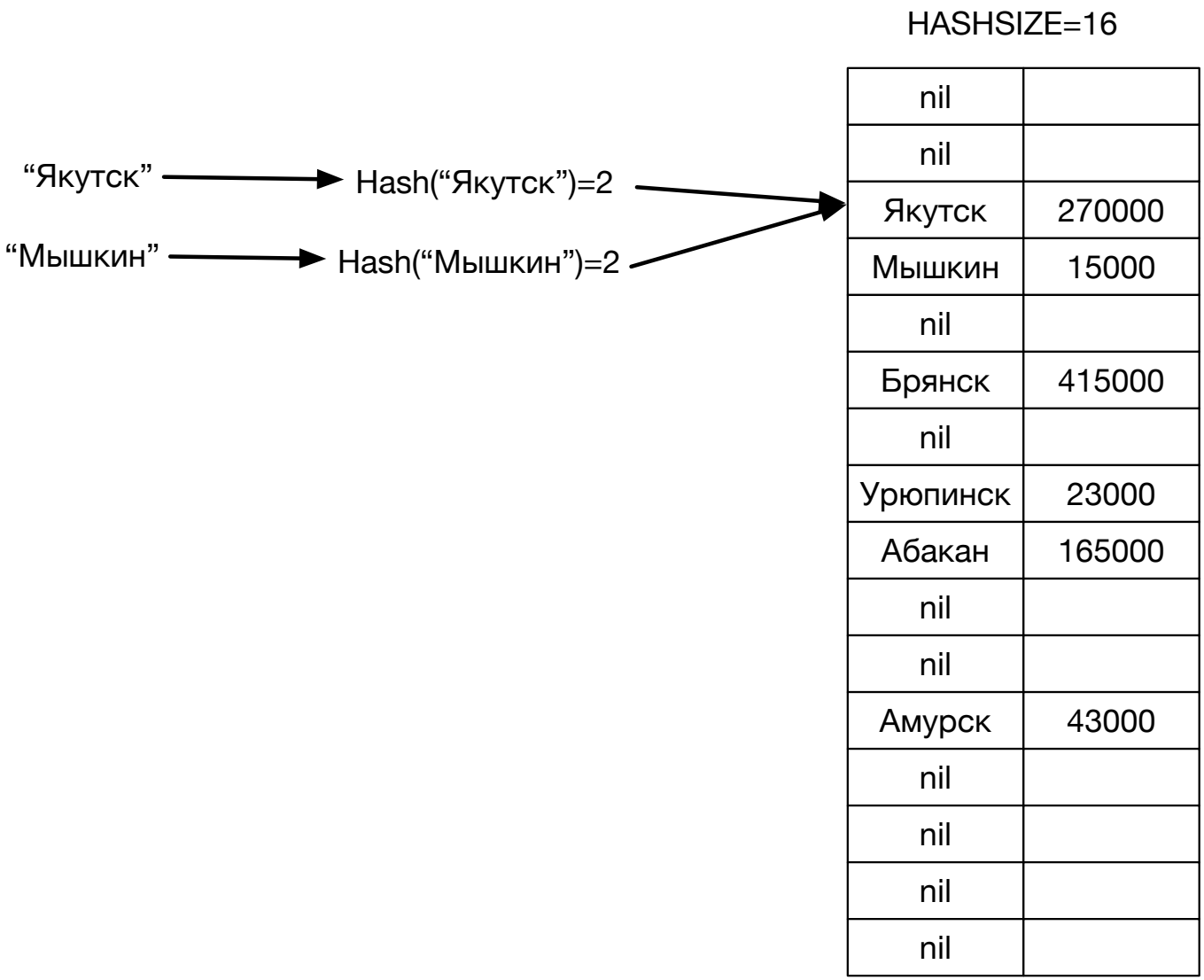
1. При поиске вычисляется хеш-функция.
2. Определяется место поиска — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент ищется во вторичной структуре.

Хеш-таблицы с прямой адресацией

1. При удалении вычисляется хеш-функция.
2. Определяется место поиска — вторичная поисковая структуре данных.
3. Если вторичной структуры нет, то нет и элемента.
4. Иначе элемент удаляется из вторичной структуре.
5. Если вторичная структура пуста, удаляет точку входа.

Хеш-таблицы с открытой адресацией

- Другой способ поиска — искать в той же таблице повторно.



Хеш-таблицы с открытой адресацией

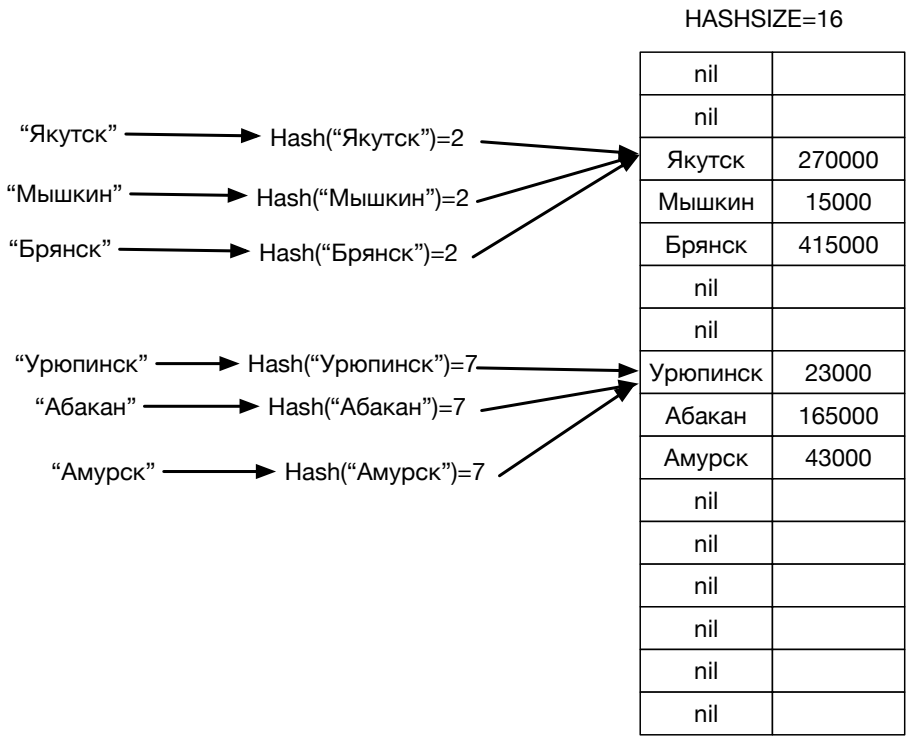
1. При поиске существующего вычисляется хеш-функция.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента.
4. Иначе по индексу — элемент с нашим ключом — элемент найден.
5. Если по индексу — элемент с другим ключом или элемент помечен удалённым, индекс увеличиваем на единицу и переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Хеш-таблицы с открытой адресацией

1. При вставке вычисляется хеш-функция.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет или элемент помечен удалённым, то вставляем по индексу и выходим.
4. Если по индексу элемент с нашим ключом — меняем данные и выходим.
5. Если по индексу элемент с другим ключом то индекс увеличиваем на единицу и переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Хеш-таблицы с открытой адресацией

1. Почему мы требуем свойства равномерности от хеш-функции.



Хеш-таблицы с открытой адресацией

1. При удалении вычисляется хеш-функция.
2. Определяется место поиска — индекс в хеш-таблице.
3. Если по индексу ничего нет, то нет и элемента.
4. Иначе по индексу — элемент с нашим ключом — элемент найден.
5. Если по индексу — элемент с другим ключом, индекс увеличивается на единицу и переходим к пункту 3.
6. Следующий индекс вычисляется по формуле $(index + 1) \mod M$.

Расширение хеш-таблиц

Когда *fill-factor* начинает превосходить 0.7-0.8 таблицу расширяют.

- Создаётся другой массив указателей с нужным размером
- Из оригинального массива в порядке увеличения индексов извлекаются элементы и вставляются в новый массив (таблицу).
- Старый массив удаляется.

Подсчёт амортизационных расходов.

- Амортизационные расходы на закрытую адресацию
- Амортизационные расходы на открытую адресацию
- Амортизационные расходы на рехеширование

Рехеширование уменьшает потребность в памяти.
Открытые обычно быстрее

Хеш-таблицы с открытой адресацией

Рекомендации по использованию.

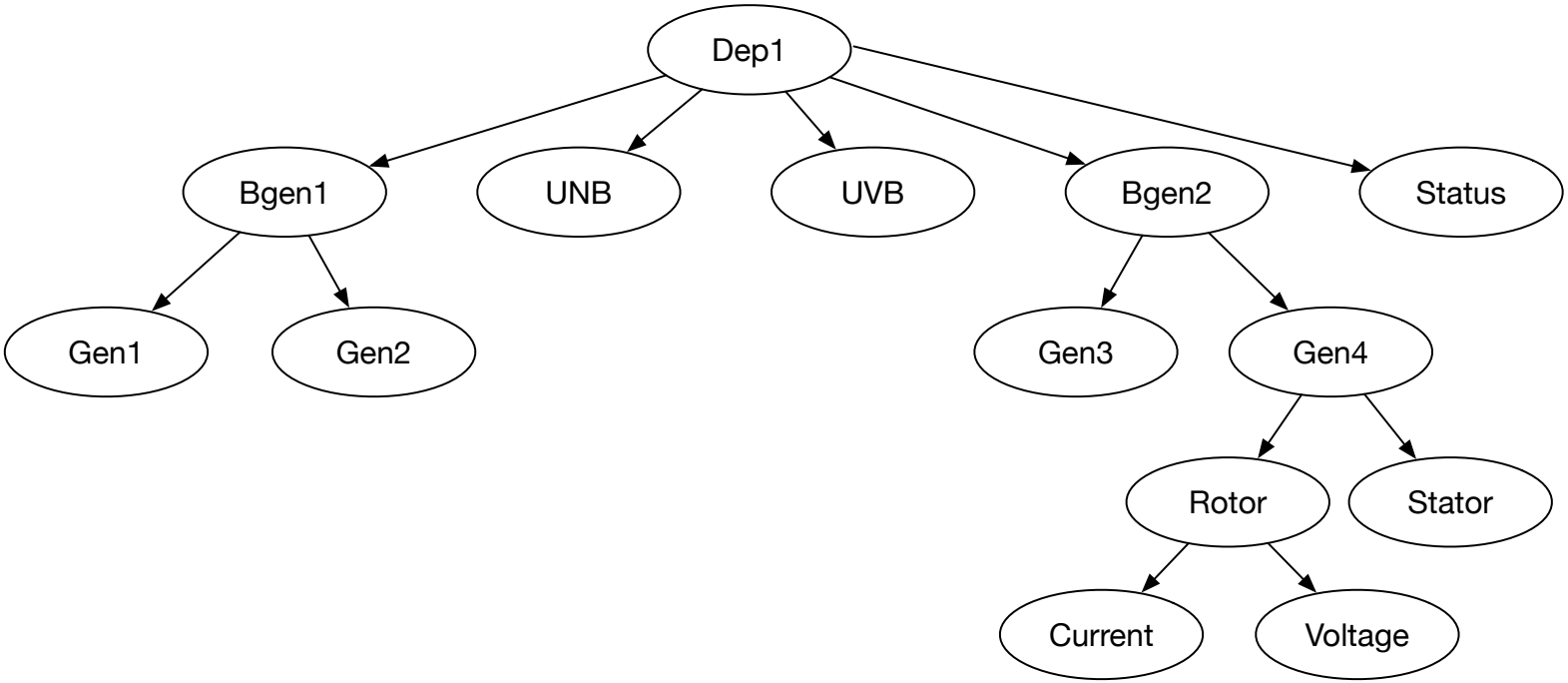
1. Всегда использовать хорошую хеш-функцию!
2. Использовать *fill-factor* не больше 0.5-0.6.

Сочетание хеш-таблиц и деревьев

Хеш-таблицы и деревья

Задача:

- Имеется набор объектов, представляющих иерархию гидроэлектростанции (фрагмент)



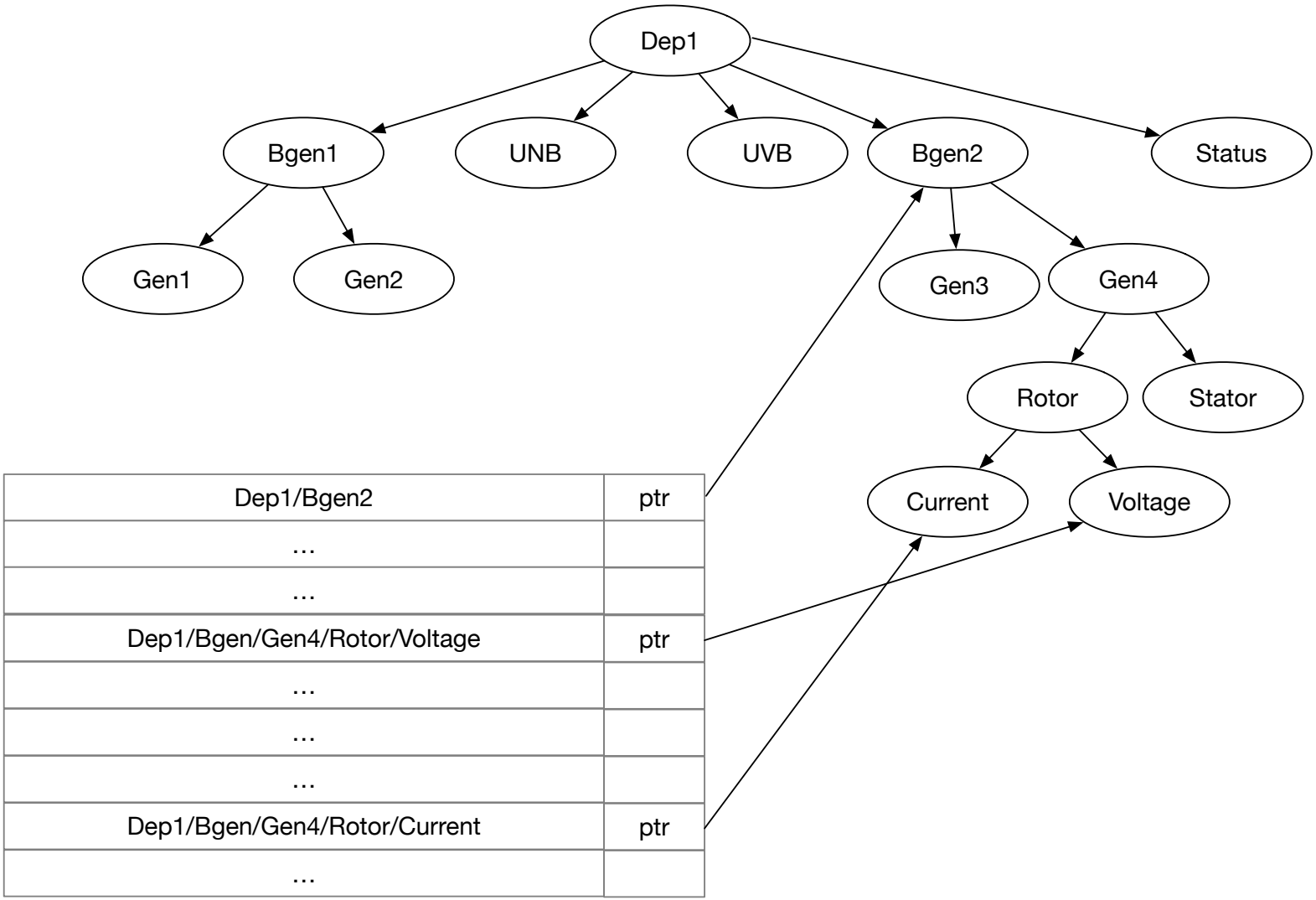
Dep1/Bgen/Gen4/Rotor/Voltage=127

Хеш-таблицы и деревья

- Дерево представляет объекты в виде указателей.
- Для функционирования важна структура дерева.
- Дерево не является деревом поиска, содержит разное число потомков.
- Поиск в дереве медленный.
- Доступ к `любому` объекту возможен через полное квалифицированное имя (`FQN`).
- Не все объекты одинаково часто используются.
- Ресурсы на компьютере сильно ограничены.

Хеш-таблицы и деревья

- Доступ к элементу через кэш, реализованный в виде хеш-таблицы.



Хеш-таблицы и деревья

- Имеется хеш-таблица небольшого размера, содержащая `FQN` и указатель на узел дерева.
- При поиске `FQN` просматривается хеш-таблица. Если такая запись есть — возвращается указатель на объект.
- Если записи нет, то производится поиск по дереву и на место в хеш-таблице записывается новый ключ и найденный указатель.
- При незначительном расходе памяти удалось ускорить амортизированно типичные поиски в несколько раз.

Хеш-таблицы во внешней памяти

Хеш-таблицы во внешней памяти

Задача: имеется $5 \cdot 10^9$ записей, состоящих из уникального ключа размером 129 байтов и данных, размером 260 байтов.

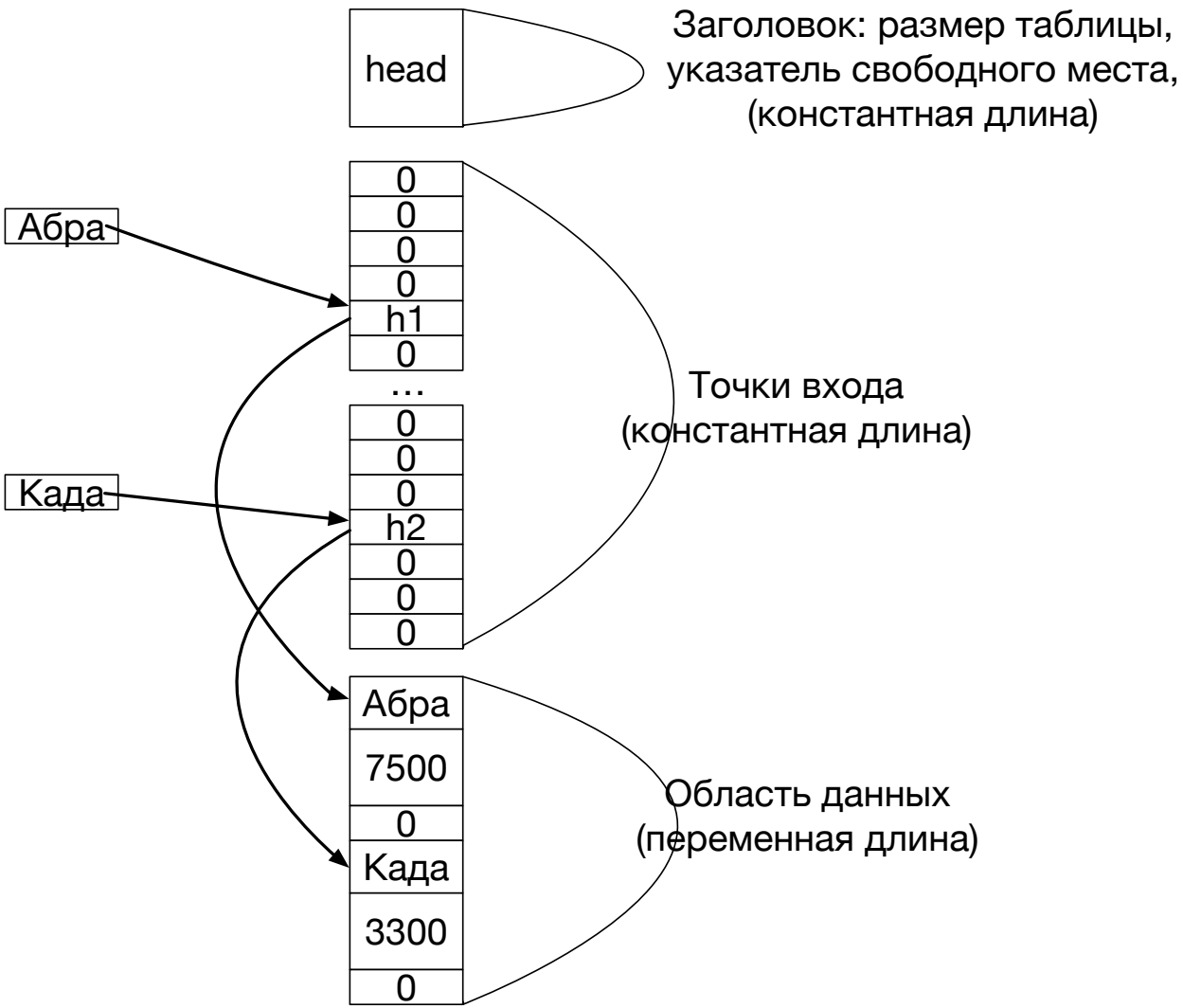
Данные располагаются в 5000000 файлах, в каждом из которых по 1000 строк.

Требуется организовать данные так, чтобы обеспечить быстрый поиск по ключу.

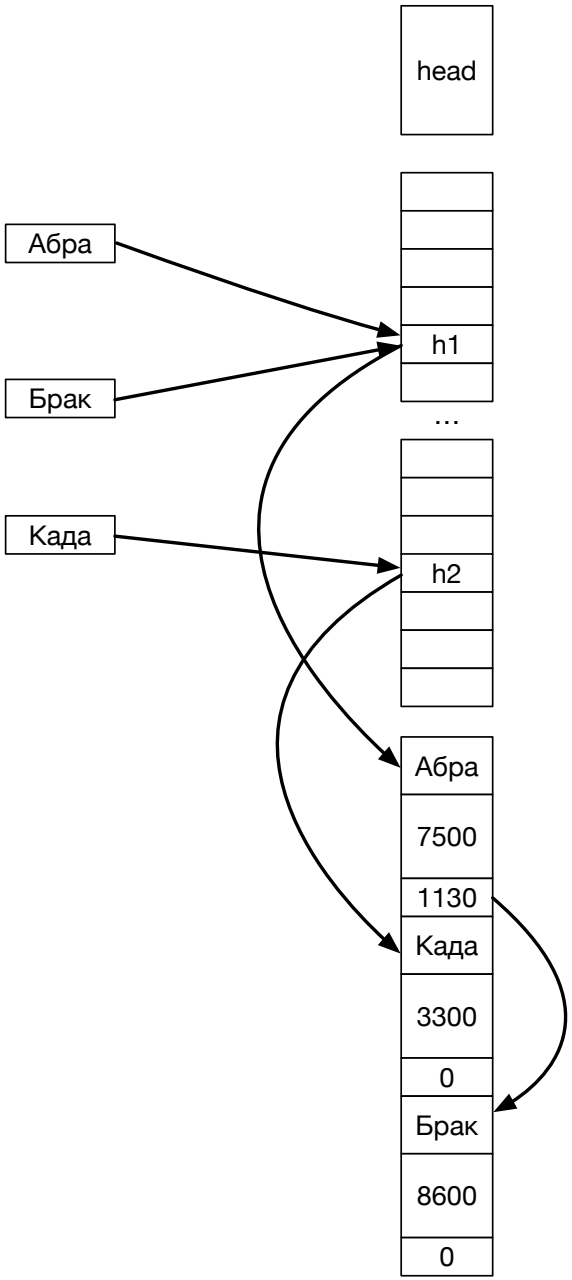
Хеш-таблицы во внешней памяти

- Общий размер превышает 300GB.
- Поиск нужен будет в непредсказуемое время.
- Количество поисков велико, но много меньше общего числа записей.
- Допустимо хранение результатов преобразования данных на устройстве с произвольным доступом.

Хеш-таблицы во внешней памяти



Хеш-таблицы во внешней памяти



Хеш-таблицы во внешней памяти

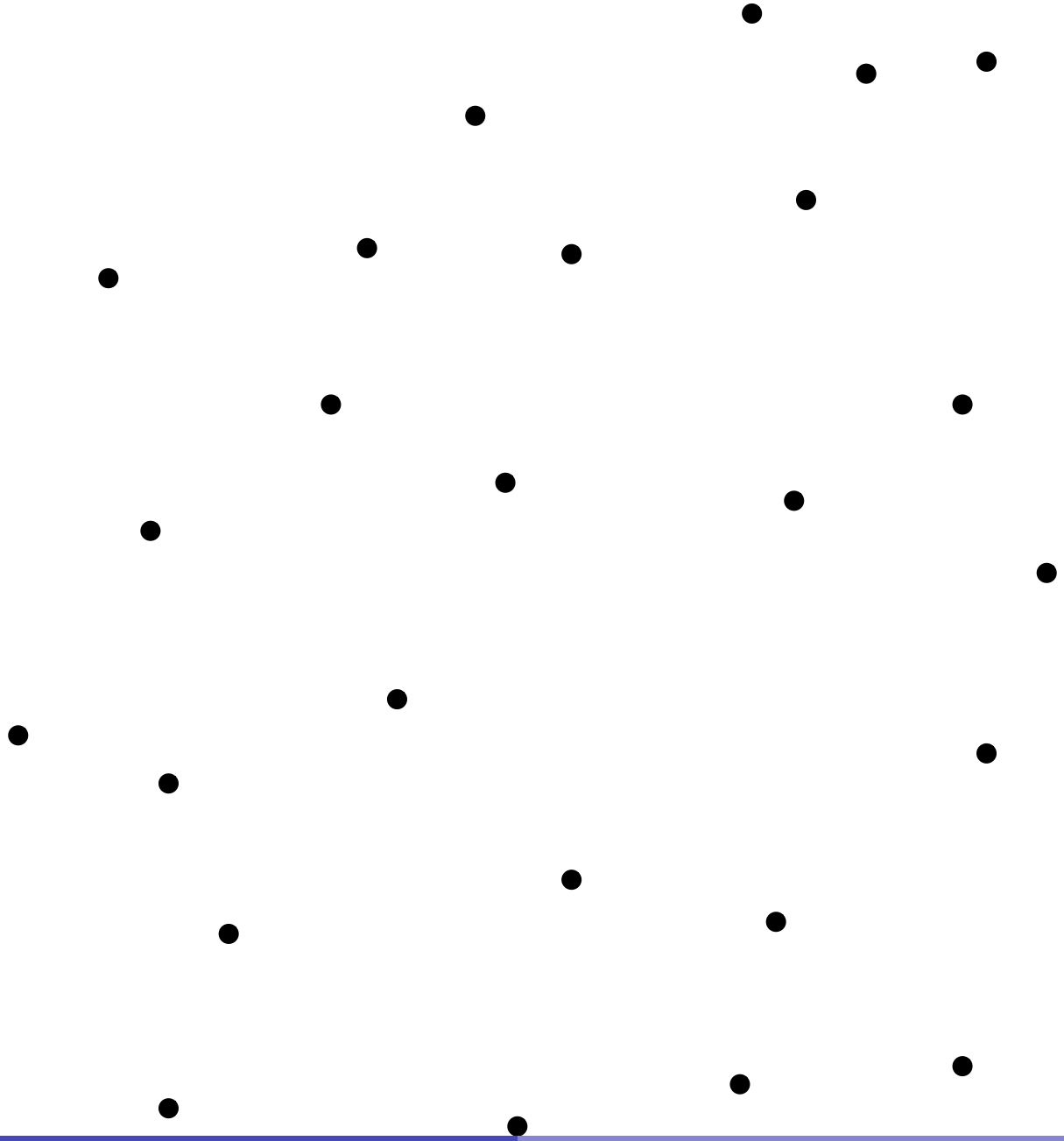
- Должны сохраняться при завершении программы (*persistent*)
- Минимизировать количество операций.
- Для оптимизации работы использовать кэширование.

Практическое использование АСД

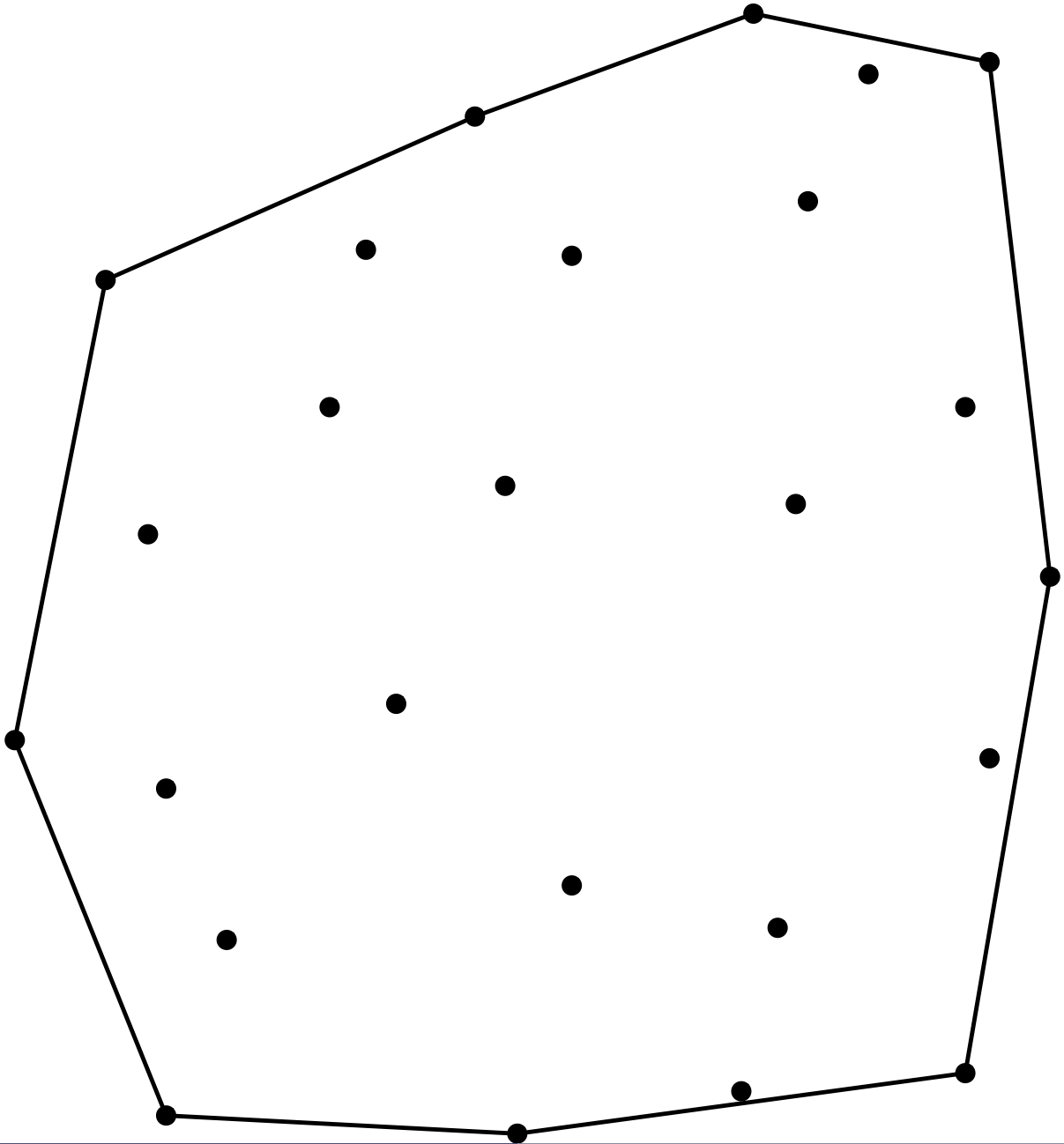
Практическое использование: задача триангуляции

- Требуется соединить все точки между собой так, чтобы сумма периметров полученных треугольников была минимальной.

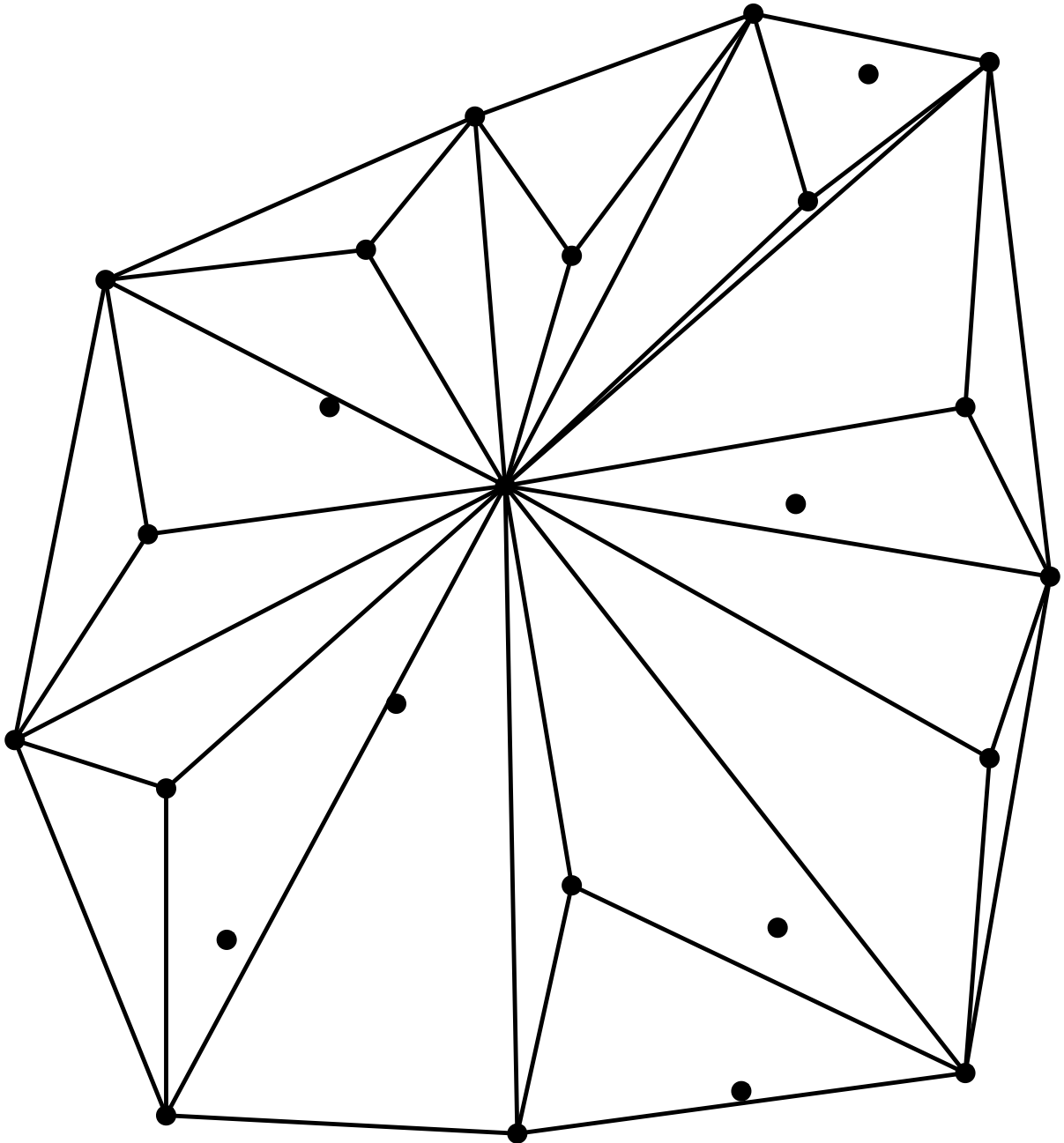
Практическое использование: задача триангуляции



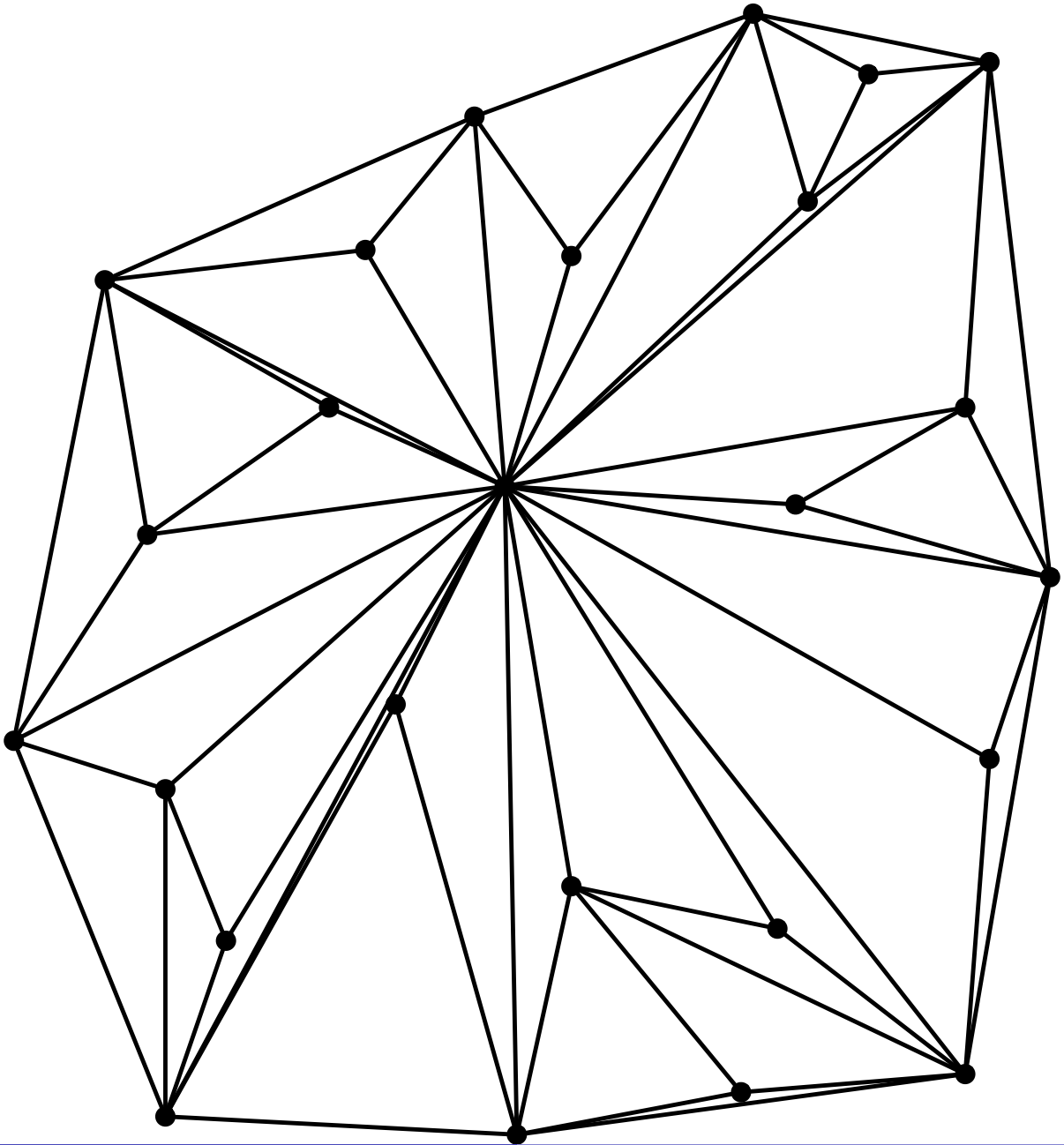
Практическое использование: задача триангуляции



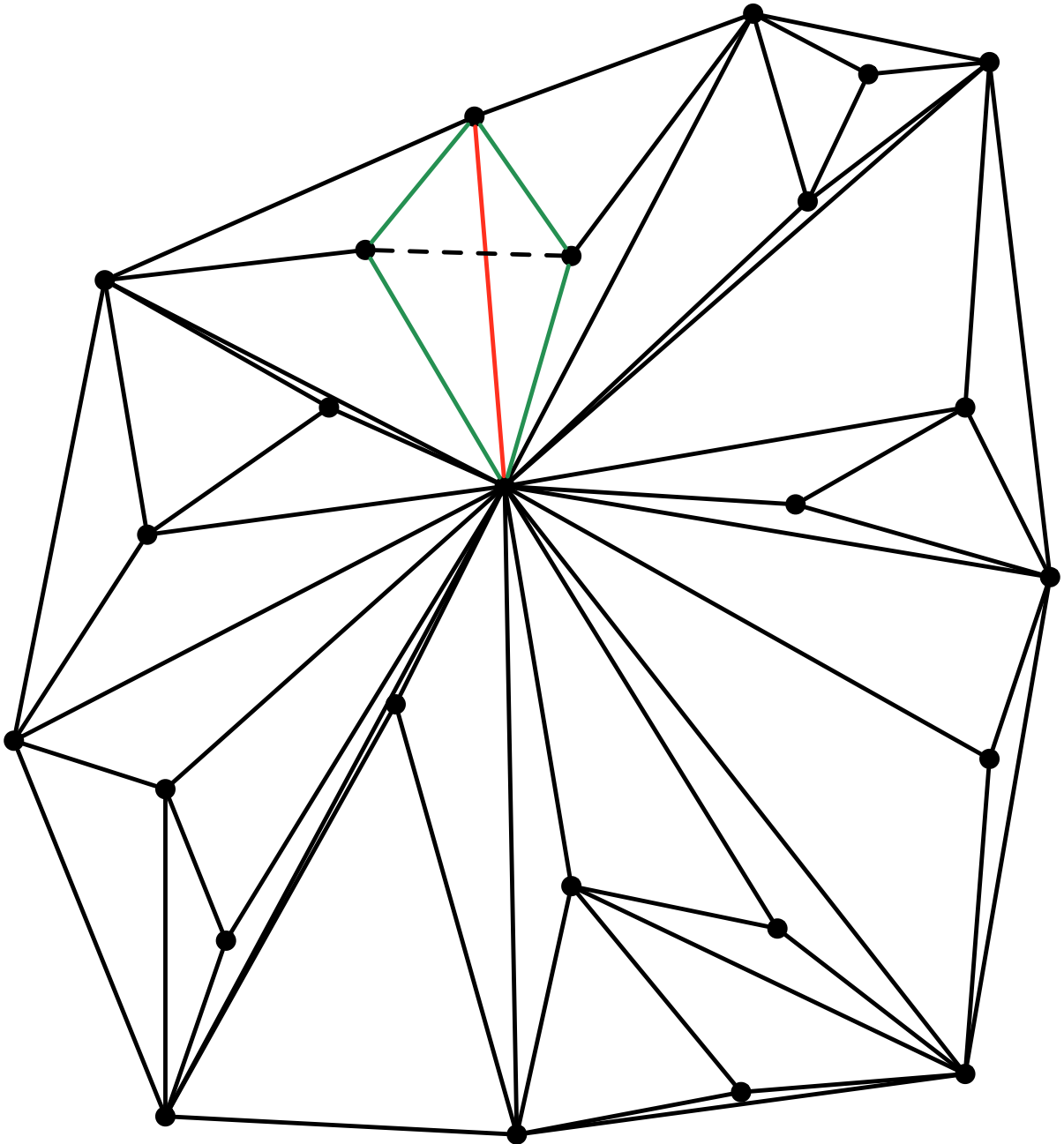
Практическое использование: задача триангуляции



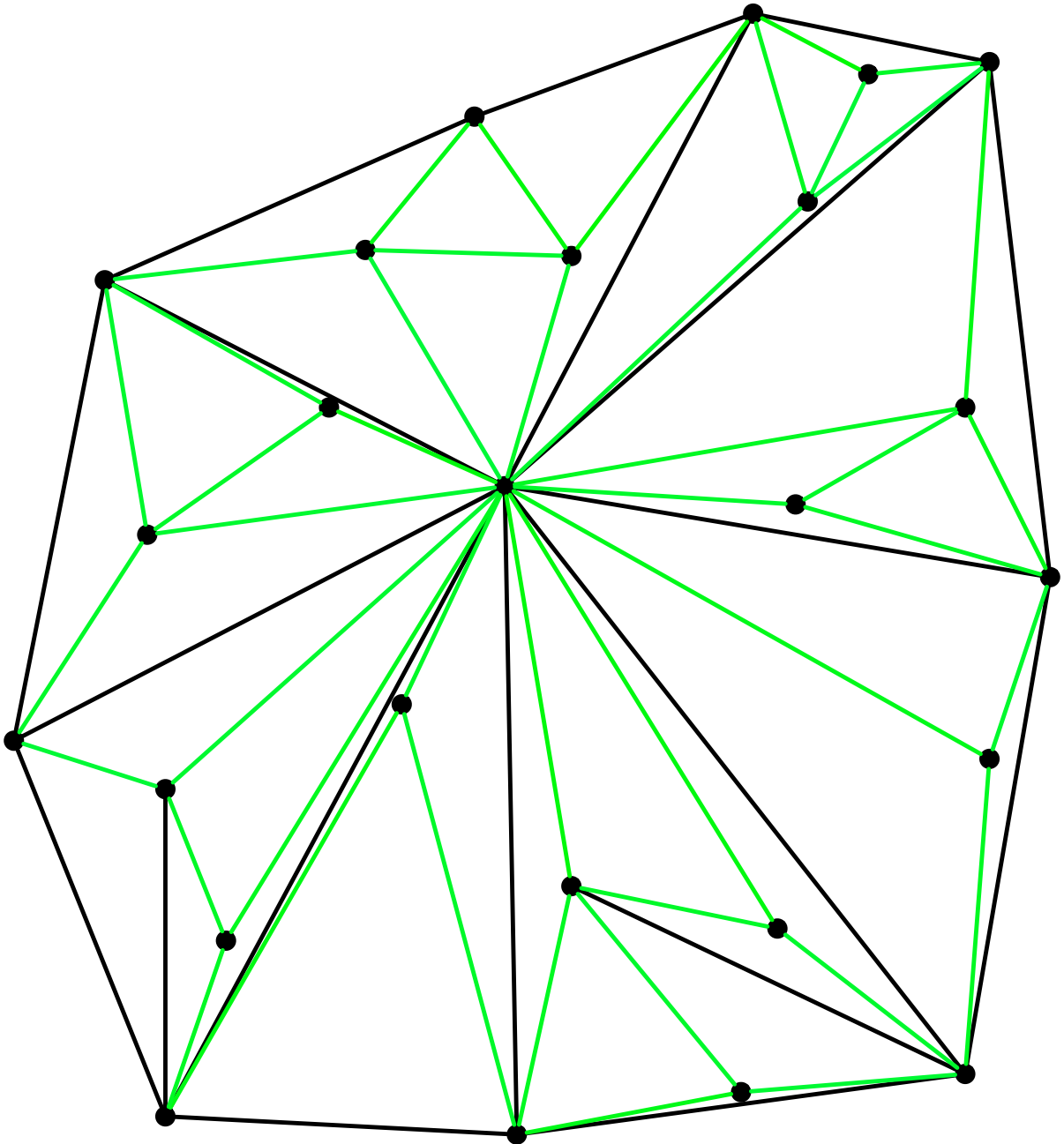
Практическое использование: задача триангуляции



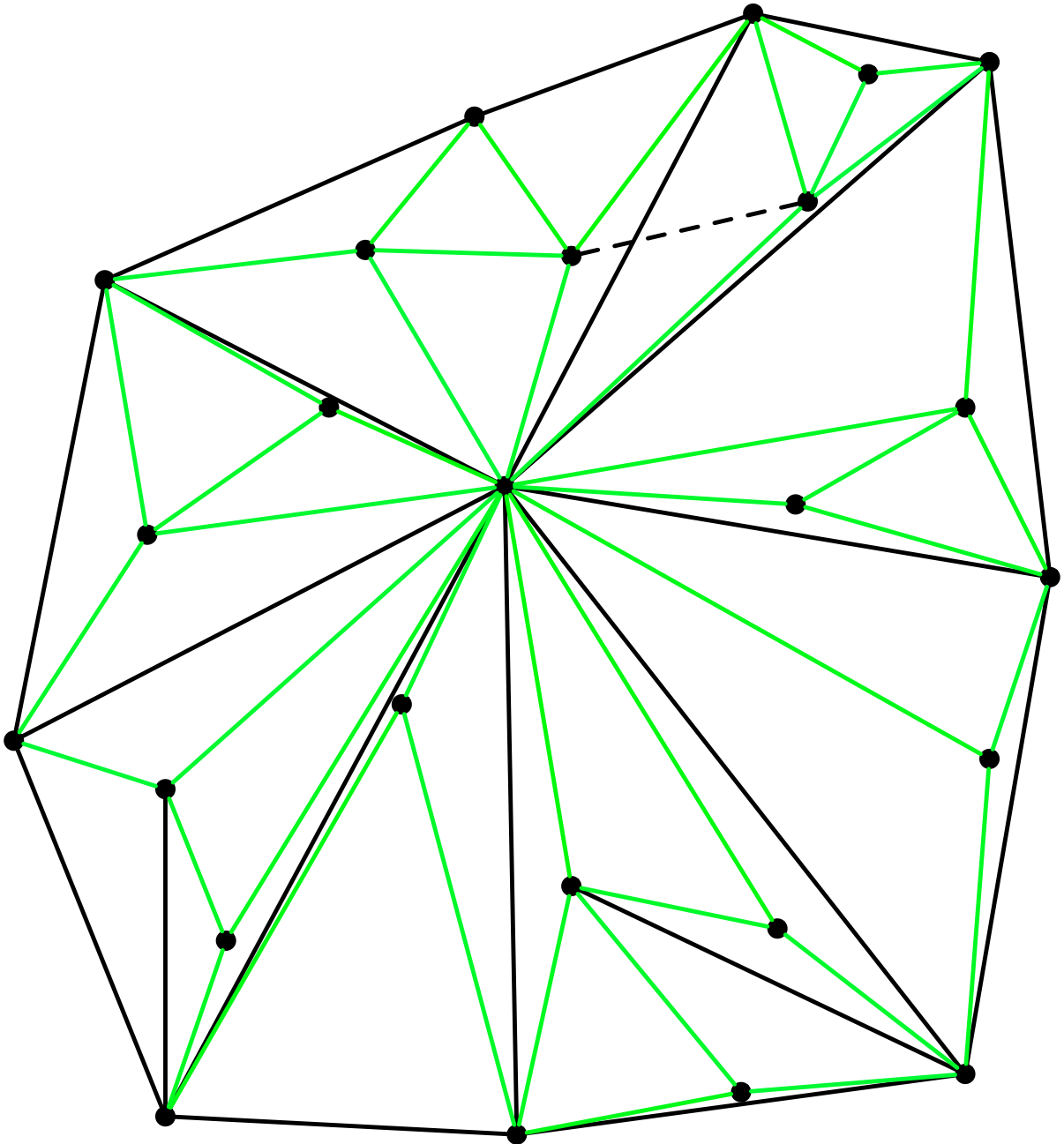
Практическое использование: задача триангуляции



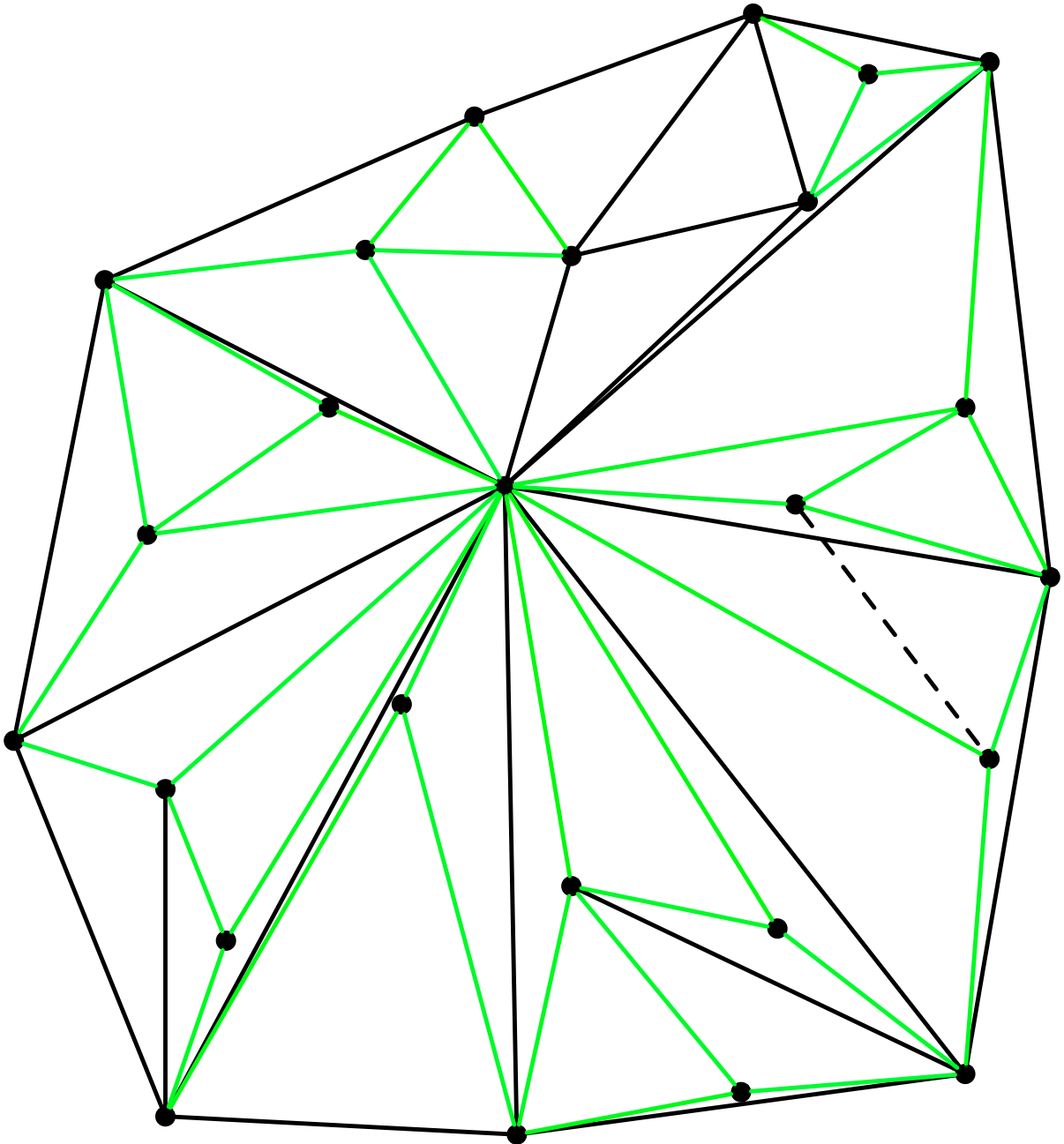
Практическое использование: задача триангуляции



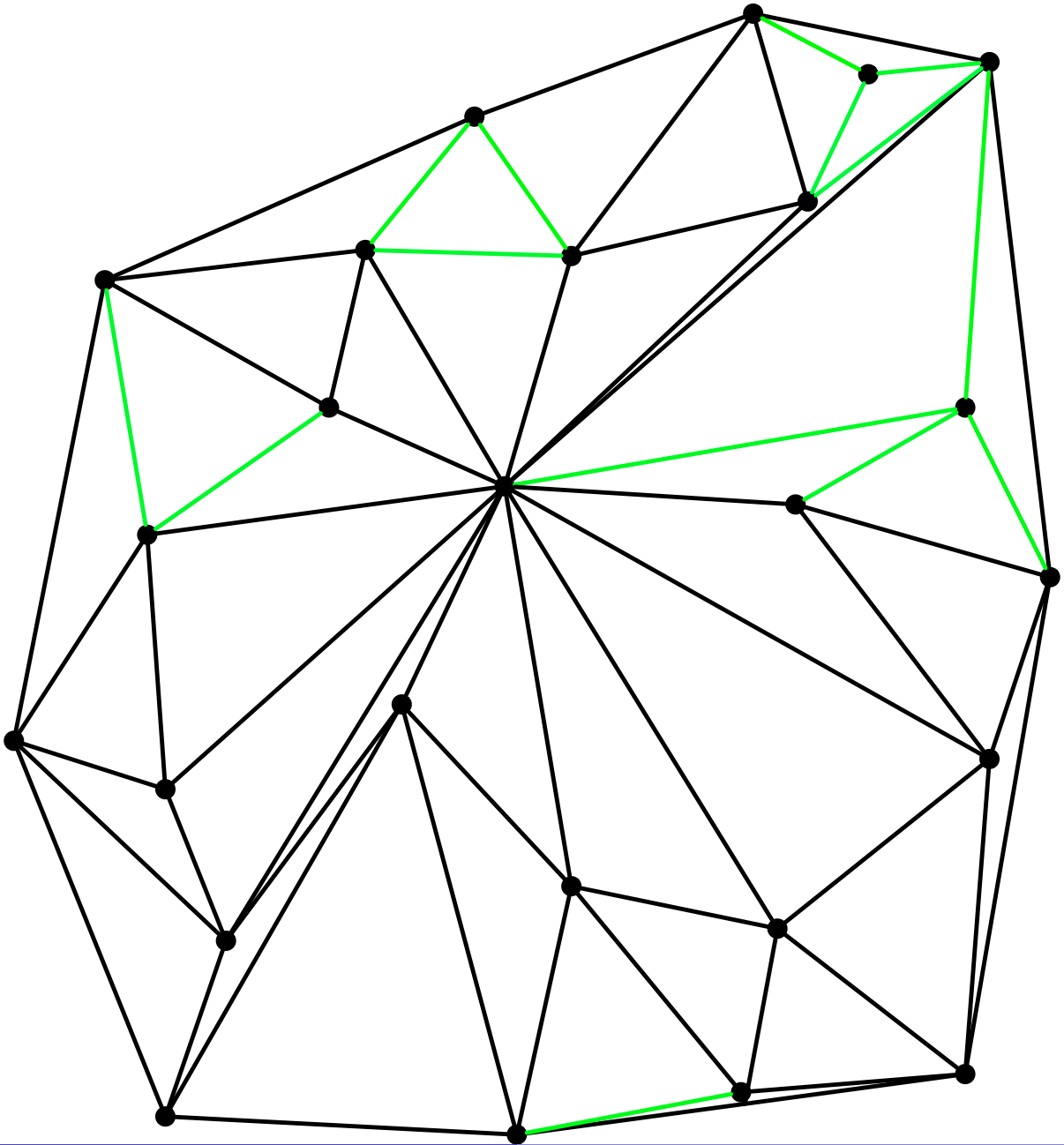
Использование алгоритмов и структур данных



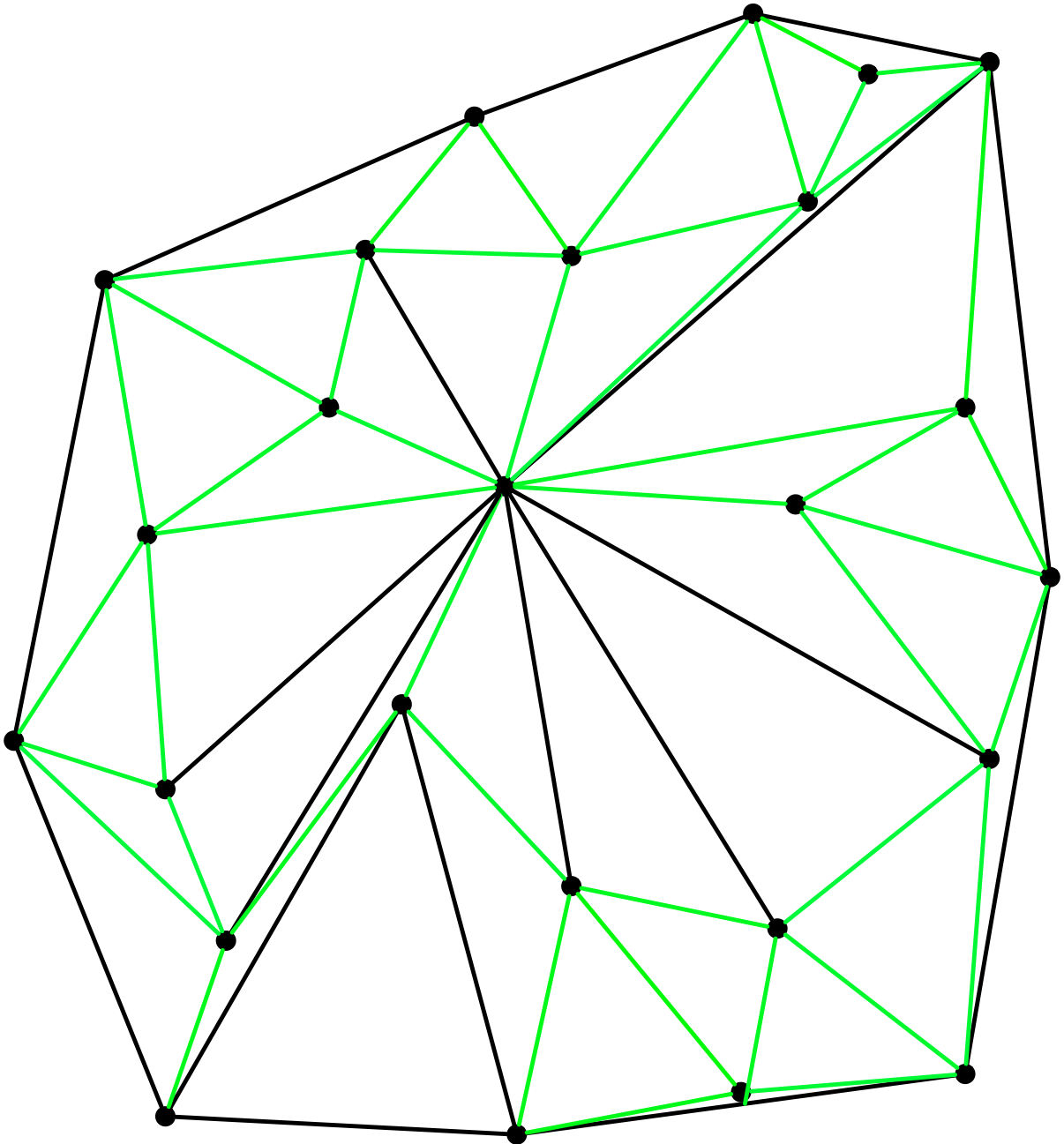
Использование алгоритмов и структур данных



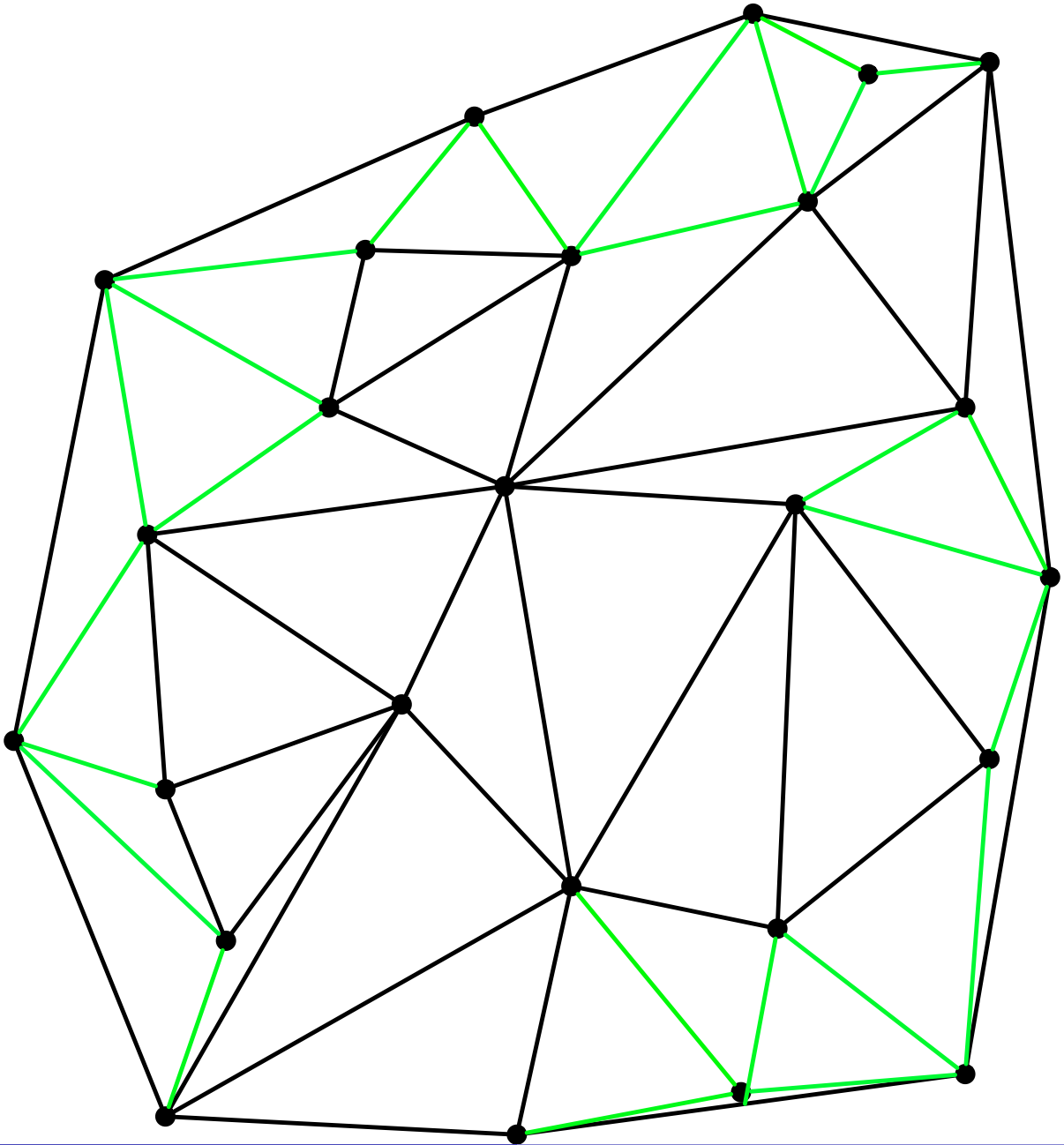
Использование алгоритмов и структур данных



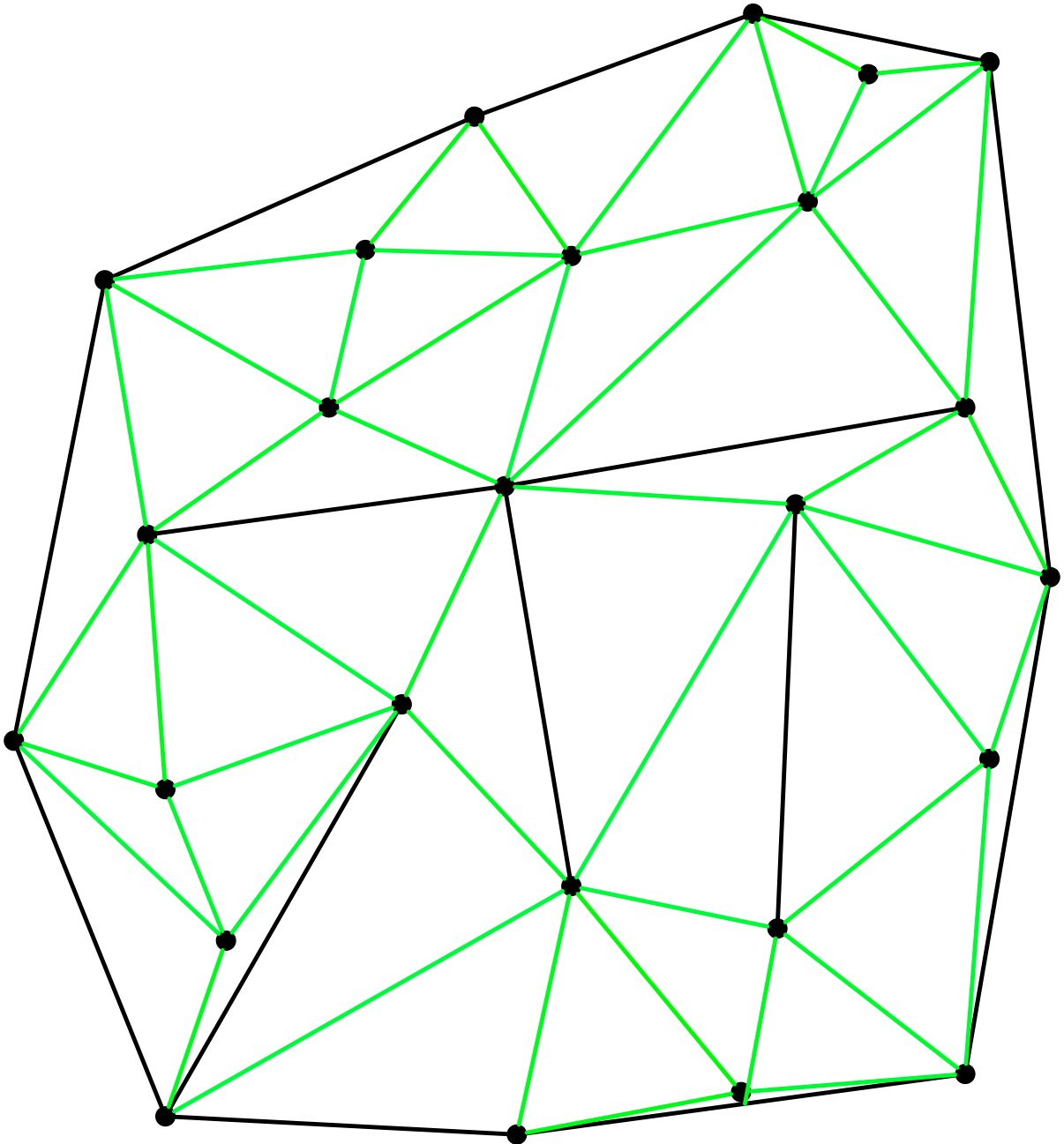
Использование алгоритмов и структур данных



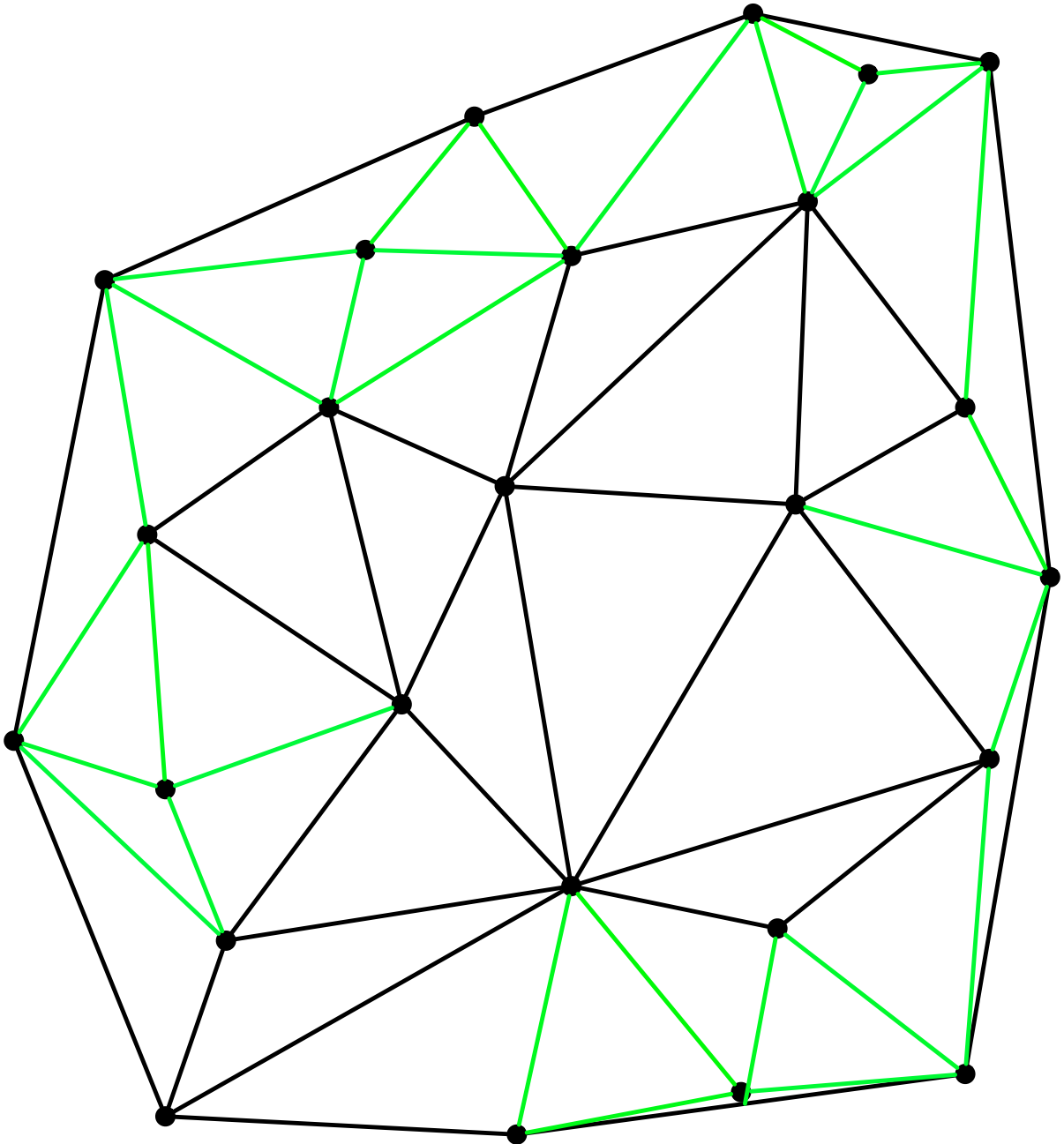
Использование алгоритмов и структур данных



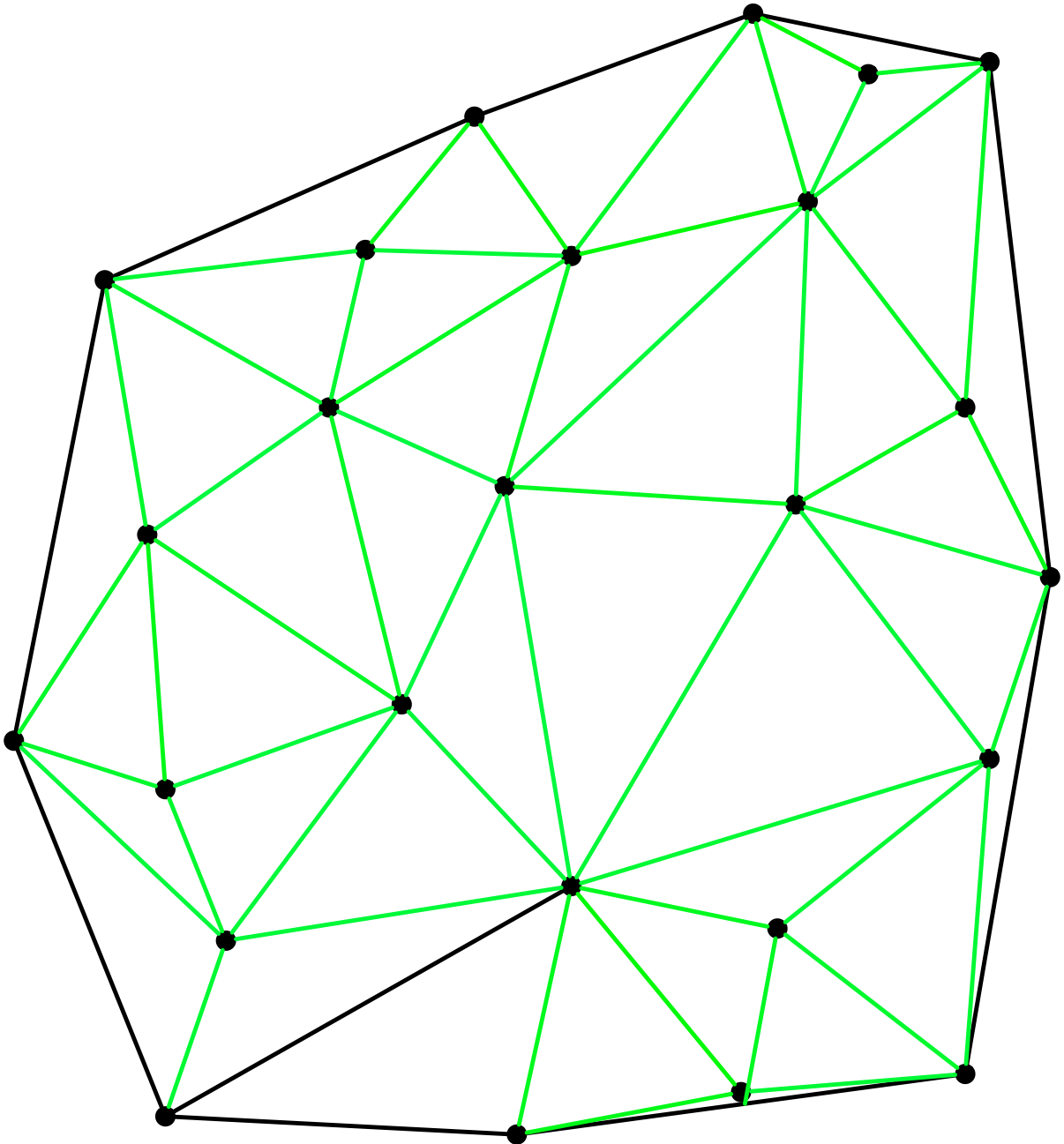
Использование алгоритмов и структур данных



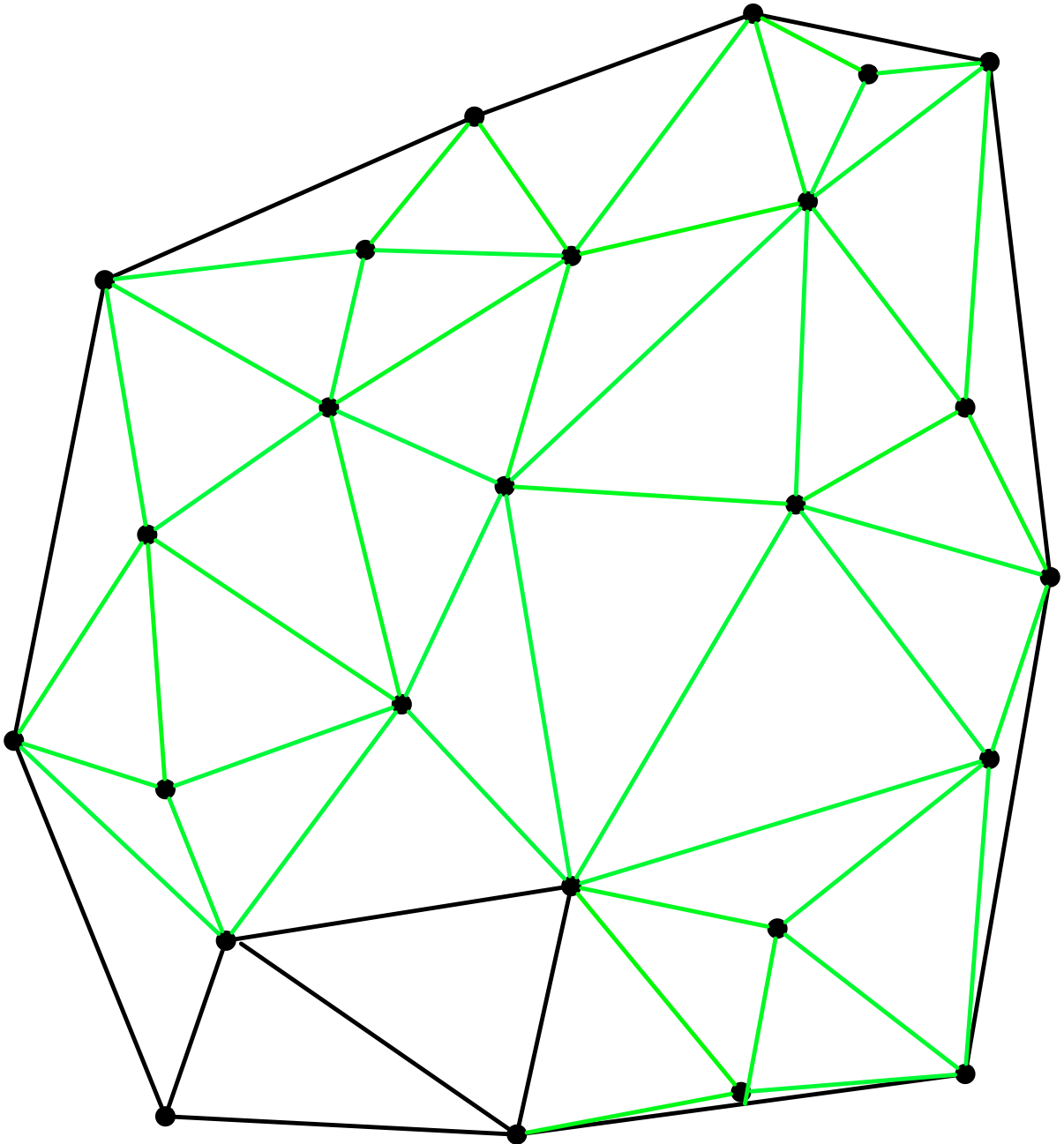
Использование алгоритмов и структур данных



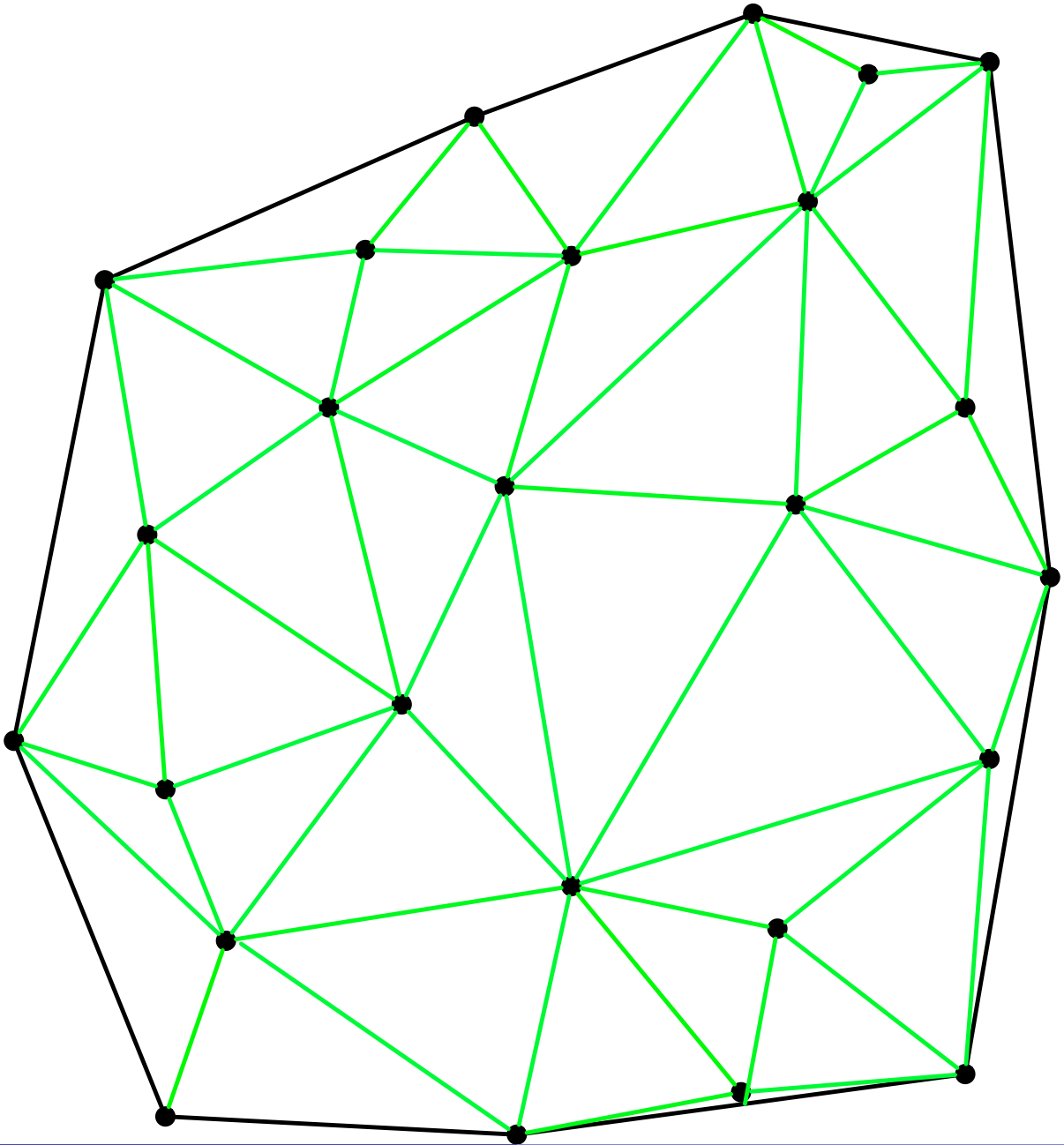
Использование алгоритмов и структур данных



Использование алгоритмов и структур данных



Использование алгоритмов и структур данных



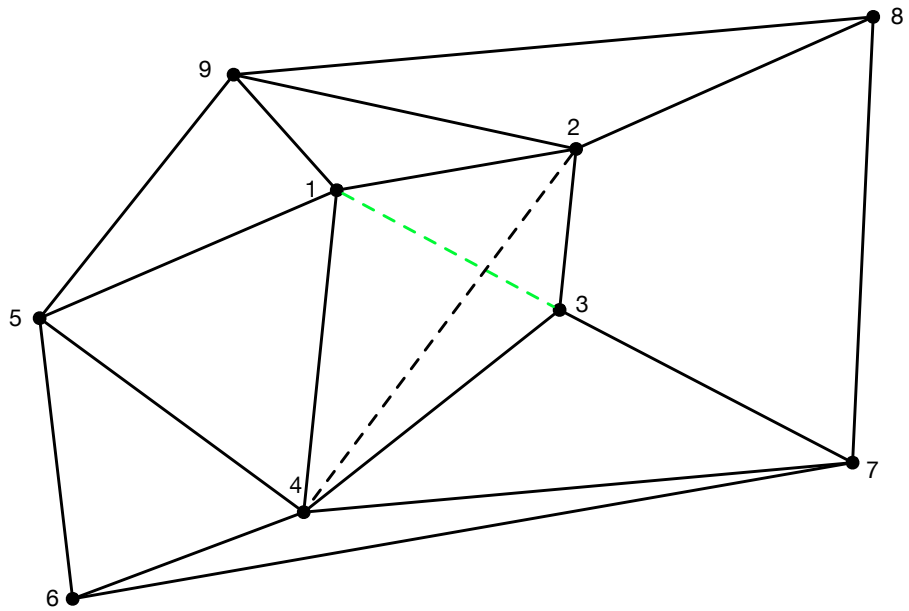
Использование алгоритмов и структур данных

Требуемые для решения задачи структуры данных.

- Сами точки. Массив.
- Связи между точками. Свойства:
  - После начальной разбивки размер не изменяется
- Связи, которые требуется проверить (чёрные)
  - Повторения не допускаются
  - Можно выбирать в любом порядке
  - Размер не превосходит общего количества связей
- Вершины двух треугольников, для которых связь является основанием.
  - Совпадает с количеством связей
  - Изменяется при изменении топологии

Использование алгоритмов и структур данных

- Связь
  - вершины двух опорных треугольников
  - чёрная/зелёная
- Замена чёрной связи (2,4) на зелёную (1,3)
  - Теперь связи (1,2), (2,3), (1,4), (3,4) — чёрные.
  - Надо их быстро найти, пометить их чёрными и поместить в множество на обработку.
  - Надо удалить связь (2,4) и создать связь (1,3)



Использование алгоритмов и структур данных

- Точки: массив N.
- Связи: хеш-таблица.
  - Размер $\Theta(N)$ не изменяется.
  - Частый поиск.
  - Более редкое изменение.
- Приоритетная очередь связей на обработку.

Спасибо за внимание.

Следующая лекция — Динамическое программирование.