Структуры и алгоритмы обработки данных

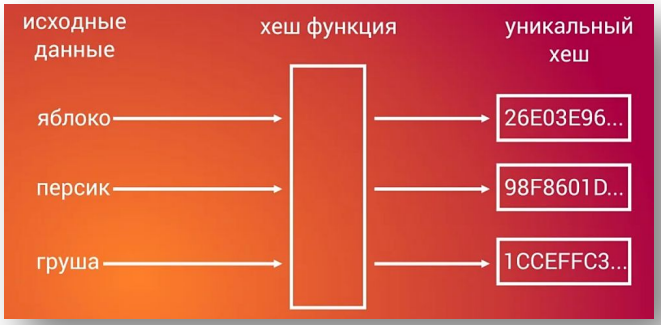
Лекции 7-8:
Рекурсивные алгоритмы.
Поиск в массиве (таблице). Хеширование.
Поиск в тексте.

Рысин М.Л.

11. Хеширование.

Хеширование –

- Это преобразование (отображение) **входного ключа** – исходных данных (сообщения, массива) в **выходную битовую строку** определённой длины, производимое алгоритмом **хеш-функции**:



- Пусть есть **динамическое множество** – набор записей, количество и состав которого может изменяться на этапе исполнения кода
- Часто динамические множества поддерживают только **словарные операции** – добавление, удаление и поиск записей по **уникальному** ключевому полю (**ключу**)
- Механизм хеширования используется для **ускорения поиска** – в динамическом множестве обеспечивается доступ к записи за время **O(1)** в лучшем и среднем случаях и **O(n)** – в худшем
- Достижение сложности O(1) возможно только при **прямом обращении** к нужному элементу, как в массиве по индексу

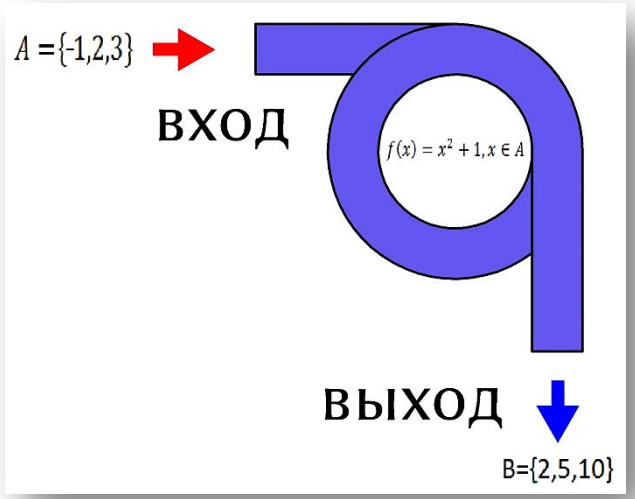
Пример 1: ключи как индексы в массиве

- Пусть требуется создать БД персонала предприятия, численность сотрудников в котором 100 человек, хранить нужно анкетные данные этих сотрудников, в т.ч. **ИНН**, состоящий из **10 цифр**

ИНН	Имя	Рост	Возраст	Профессия
…	…	…	…	…
3213456784	Иван Иванов	181	33	Банкир
3446346346	Семён Семёнов	181	30	Дезинсектор
…	…	…	…	…

- Формально **ключом** к данным может стать ИНН
- Но для **прямого доступа** к данным по ключу (без дополнительных преобразований) потребуется создать массив из $\mathbf{10^{10}}$ записей (**разреженный массив**)
- Т.к. сотрудников всего 100, то это **нерациональная организация** данных в памяти →

Отображение (в математике) –

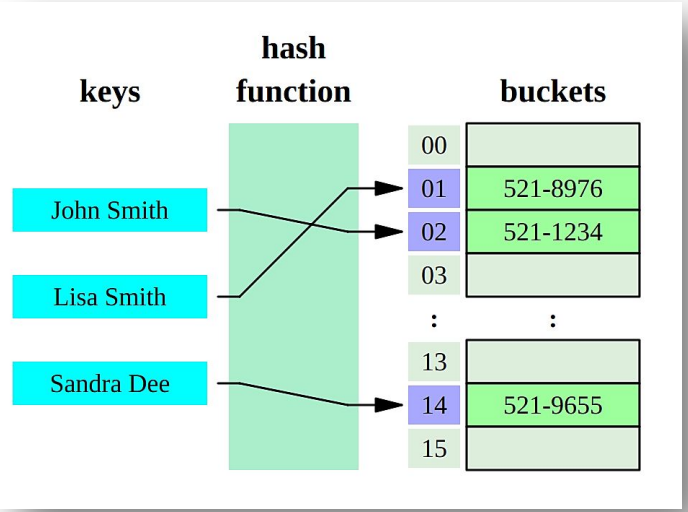


- Это **функция** M, определенная на множестве элементов (в **области определения**) одного типа и принимающая значения на множестве элементов (в **области значений**) другого типа:

M(d) = r,

где d – значение из области определения функции M, r – значение из области значений функции M

Хеш-функция



- Хеширование (**хеш-функция**) создаёт **отображение** (соответствие) **множества ключей** в **множество индексов** массива (множество целых чисел)
- Алгоритм хеш-функции преобразует **уникальный ключ** записи **в её индекс** в таблице
- Структура, позволяющая хранить пары «ключ» – «хеш-код» (индекс), называется **хеш-таблица**
- Хеш-таблица может хранить как сами элементы данных (**записи**) исходного динамического множества, так и **ссылки** на эти данные.

Пример 2:
поиск на основе хеш-таблицы

Индекс	Ключ
0	1111112300
1	1111112301
2	1111112302
3	
4	
5	
6	
7	
8	
9	
10	
11	1111112311
99	

- Пусть хеш-таблица имеет размер L=100 (по количеству сотрудников)
- Тогда хеш-функция **h(key)** должна генерировать индекс в пределах (множество значений) от 0 до 99
- Алгоритм вычисления индекса на основе ключа (ИНН) может быть простым:

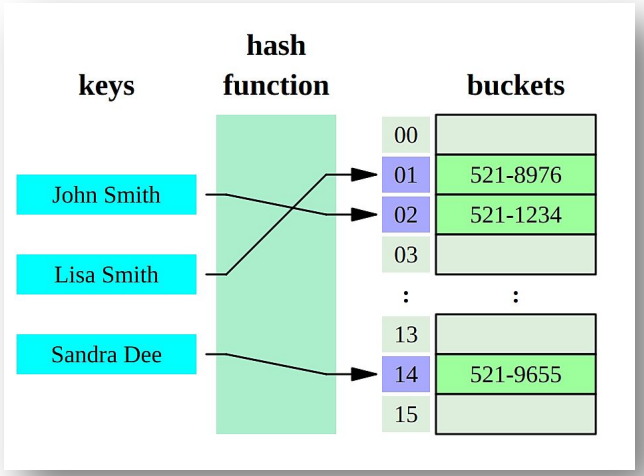
  h(key) = key % L

- После формирования хеш-таблицы можно решать задачи **поиска**, удаления и вставки записей
- Например, для поиска **ключ передаётся** той же **хеш-функции**, которая **вернёт индекс** искомой записи в массиве
- Т.о. хеш-таблица обеспечивает и **оптимальность** организации данных в памяти, и **быстрый поиск**.

Свойства хеш-функции

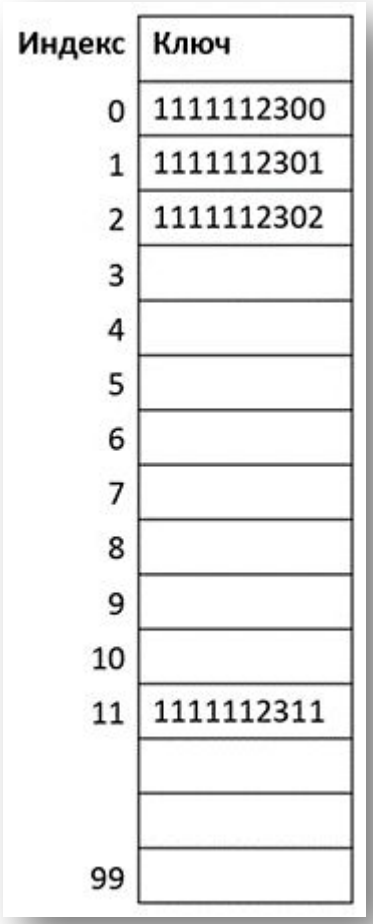
1. Простой (**быстрый**) алгоритм вычисления хеш-значения
2. Должна всегда возвращать одно и то же хеш-значение для одного и того же ключа
3. ***Не обязательно*** возвращает **разные** хеш-значения для **разных ключей**
4. Использует всю область значений с одинаковой вероятностью
5. **Однородность (равномерность) хеширования**:
   - Для каждого ключа **равновероятна** генерация любого значения из области значений хеш-функции **независимо** от хеширования остальных ключей
   - Т.е. созданный индекс **не должен зависеть** от индексов, с которыми хешированы другие элементы
   - Для **успешного хеширования** очень важно, чтобы функция создавала такие индексы, что размещаемые данные были **равномерно (однородно) распределены** по хеш-таблице (признак качественной хеш-функции).

Алгоритмы хеширования



- Основанные на **делении**
- Основанные на **умножении**
- **Универсальное хеширование** – выбор функции из заданного универсального семейства по случайному алгоритму.

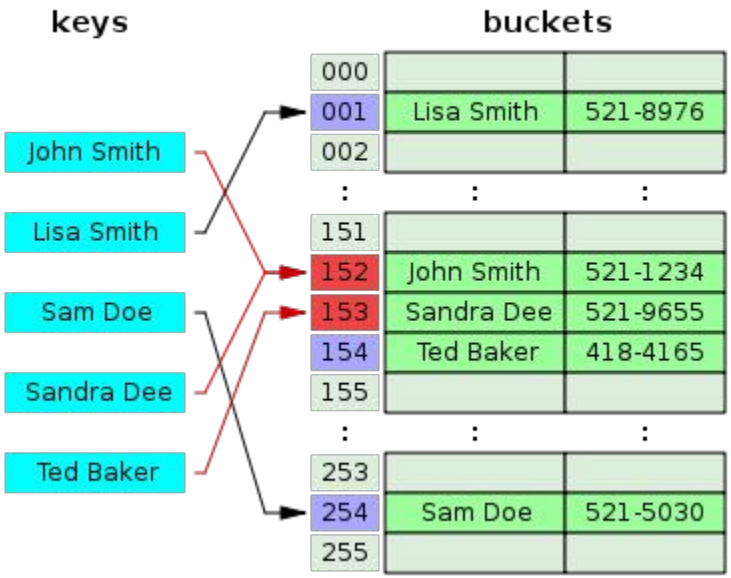
Коллизия –



Индекс	Ключ
0	1111112300
1	1111112301
2	1111112302
3	
4	
5	
6	
7	
8	
9	
10	
11	1111112311
99	

- Это ситуация, когда для **разных ключей** хеш-функция создаёт **одинаковые значения** (индексы)
- Так, в нашем примере, для нового ключа **key=1111112401** хеш-функция **key%100** вернёт **1** – индекс **уже занятой** ячейки
- Коллизии в общем случае **возможны**, поэтому процесс хеширования помимо вызова хеш-функции обязательно включает в себя обработку (**устранение**) возникающих **коллизий**.

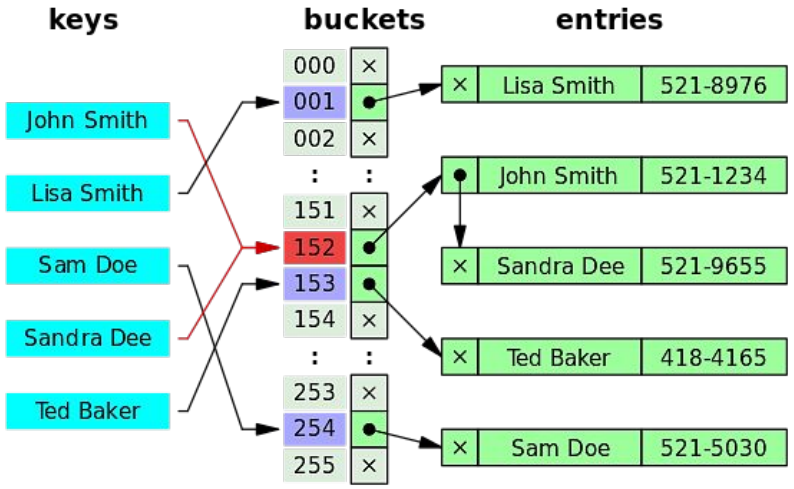
Подходы (технологии) устранения коллизий



Два подхода:

1. **Цепное хеширование** (формирование списков из элементов, хешированных с одним индексом) →
2. **Открытая адресация** (хеш-таблица большого объёма).

1. Цепное хеширование –

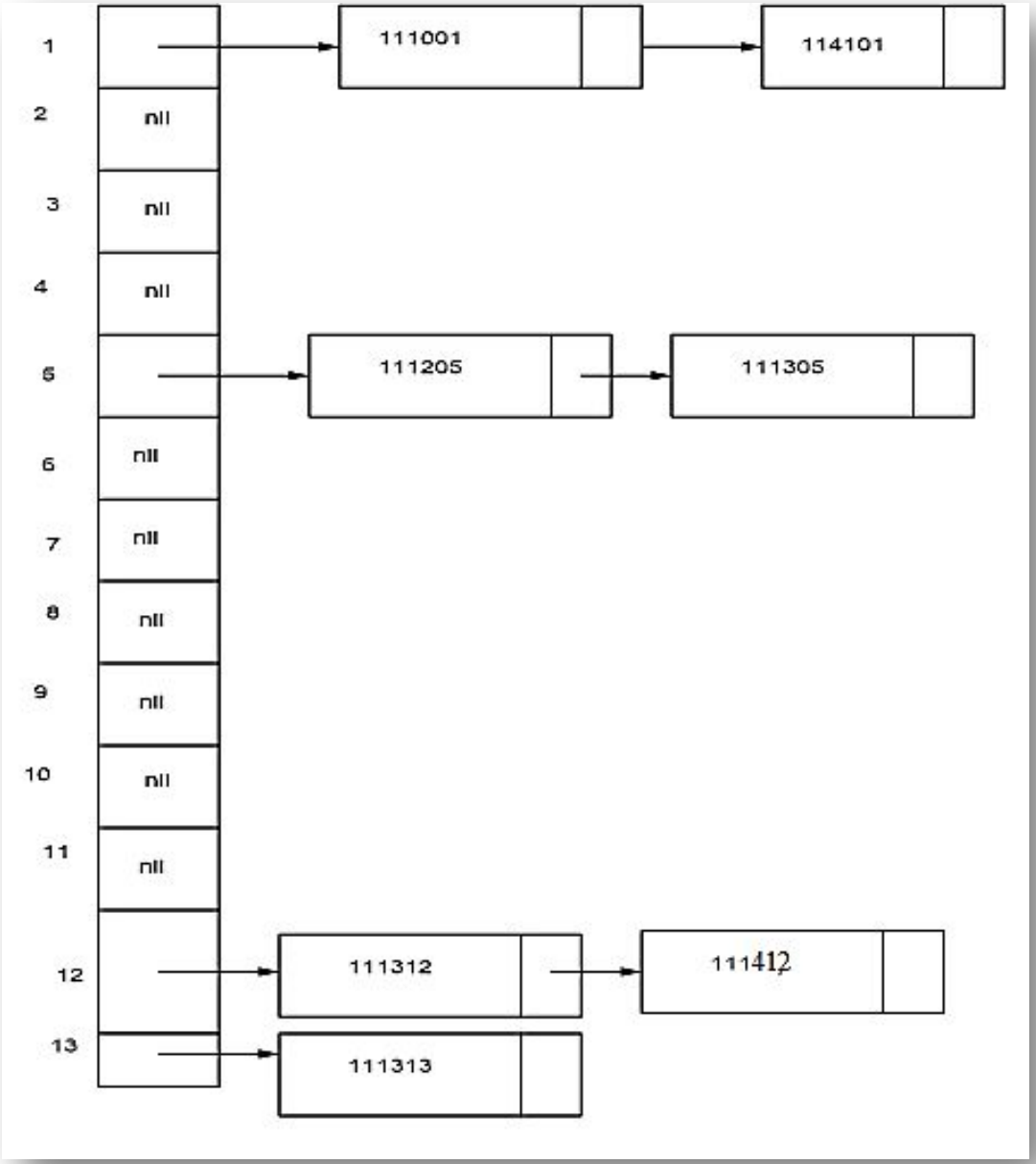


- Это способ разрешения коллизий, когда элементы, ключи которых получили **один индекс**, хранятся в одном **однонаправленном списке** (цепочке переполнения) с вершиной в **хеш-таблице**.
- Тогда хеш-таблица – это массив указателей на такие списки
- Пример. →

Пример 3: Хеш – таблица с динамическими списками для устранения коллизий (1/2)

Ключ	Индекс	
111001	1	
111205	5	
111312	12	
114101	1	коллизия
111312	12	коллизия
111313	13	
111305	5	коллизия

- Создадим хеш-таблицу для ключей из примера выше с множественными коллизиями
- При возникновении коллизии данные с ключом помещаются в начало списка, который в таблице связан с элементом, для которого хеш-функция создала индекс →



Пример 3: Хеш – таблица с динамическим и списками для устранения коллизий (2/2)

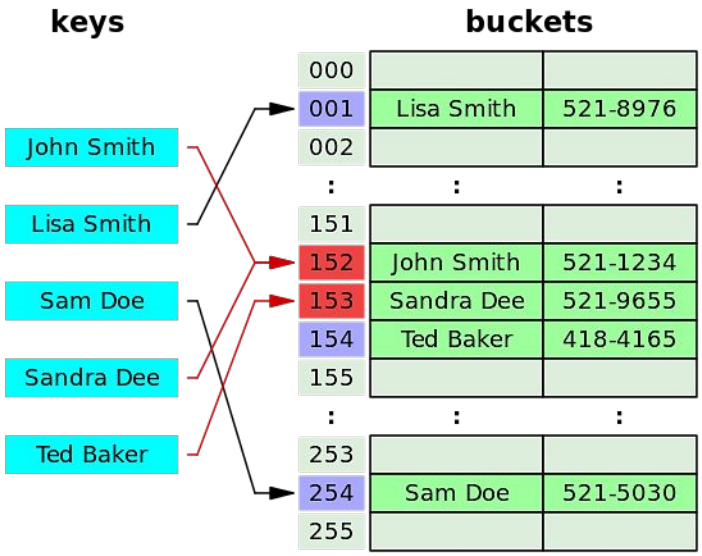
Проблемы цепного хеширования

1. Проблема **однородного хеширования**:
   - Желательно, чтобы **цепочки** переполнения были **примерно одной длины**, иначе сложность поиска увеличивается от константной к **линейной**
2. Проблема **размера хеш-таблицы**:
   - Если сразу создать хеш-таблицу большого размера, то многие элементы могут не использоваться
   - Если создать небольшую таблицу, то длина цепочек будет расти (с увеличением времени поиска)

Выход – по мере увеличения количества элементов в таблице **перестраивать** её – увеличивать её размер с **рехешированием**

- Для этого вводится **коэффициент нагрузки n/m**, где n – количество записей с ключами в таблице, m – длина таблицы
- Если **n/m>0,75**, то следует увеличить размер таблицы **вдвое**, это гарантирует, что размер списков будет небольшим.

2. Хеширование с открытой адресацией



- В хеш-таблице элементы могут содержать только **пары ключ – значение** (или ключ – ссылка на значение)
- **Открытый адрес** – это свободная ячейка хеш-таблицы, **закрытый адрес** – это занятая ячейка
- Алгоритм поиска **просматривает** занятые **ячейки**, пока не найдётся элемент с искомым ключом (**успех**) или незанятая ячейка (что означает отсутствие искомого ключа – **неудача** поиска). →

Алгоритм открытой адресации для поиска и вставки в общем виде:

1. Вычисляем адрес точки входа в хеш-таблицу $\mathbf{a_0 = f(k)}$ по ключу поиска **k** (счетчик **i = 0**).

2. Если строка T[a] **свободна**, значит, записи с ключом **k** в таблице **нет** (при поиске – неудача, а при вставке – успех).

3. Если **ключ** в строке T[a] **равен k**, тогда при поиске — **успех** и возврат адреса строки $\mathbf{a_i}$ (при вставке — конец алгоритма по признаку «дублирующий ключ»).

4. Если при вставке ключ в строке T[a] **не равен k** — это **коллизия**. Увеличиваем счетчик **i = i+1** ($i \leq N$) и по некоторой функции $\mathbf{a_i = g(f(k), i)}$ вычисляем **другой адрес** ячейки внутри таблицы.

5. **Переход** на п. 2.

Последовательность проб

- Функции **g(f(k), i)**, при 1 ≤ i ≤ N, определяют последовательность адресов для просмотра элементов таблицы – **последовательность проб**
- Их выбор **ситуативен** – определяется особенностями использования таблицы при решении конкретной задачи
- Наиболее распространённые **схемы последовательности проб** в открытой адресации:
  - **линейное пробирование**; →
  - **квадратичное пробирование**;
  - **двойное хеширование**.

Линейное пробирование

- Это простейшая схема **адрес коллизии + смещение**, при которой a_i**= g(f(k), i) = f(k) + c*i**, где c – константа (наиболее простая схема при c = 1)
- Чтобы избежать **первичного скучивания** (**кластеризации**) записей, **c и N** должны быть **взаимно простыми**, c – не очень малым числом
- Схема работает хорошо, пока таблица не слишком заполнена, по мере заполнения процесс замедляется
- При c = 1 **среднее число проб** при неудачном поиске **0.5*(1 + (1 – n/m)-2)**,
  при удачном **0.5*(1 + 1/(1 – n/m))**, где **n/m** – коэффициент заполнения таблицы (нагрузки).

Пример создания хеш-таблицы с линейной открытой адресацией

ключ	фамилия
111001	Иванов
111002	Петров
111007	Сидоров
112001	Волков
212001	Лисицын
303002	Жаворонков
304002	Медведев
305002	Рыбин
303010	Акулов

исходные данные

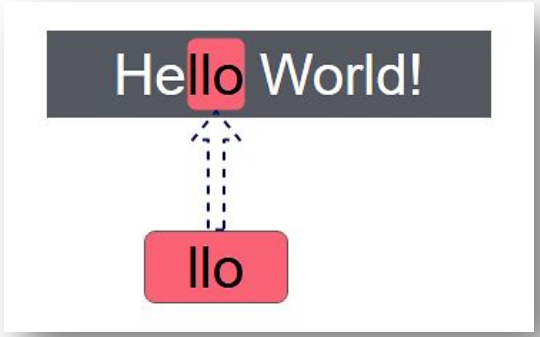
индекс	ключ	Фамилия	Признак открытой ячейки
1	111001	Иванов	false
2	111002	Петров	false
3	112001	Волков	false
4	212001	Лисицын	false
5	303002	Жаворонков	false
6	304002	Медведев	false
7	111007	Сидоров	false
8	305002	Рыбин	false
9			true
10	303010	Акулов	false
11			true
…	…..	……	true
100			true

Открытая адресация с двойным хешированием

- По этой схеме $\mathbf{a_i = g(f(k), i) = f(k) + i*h(k)}$, где h(k) — хеш-функция, почти такая же, что и f(k), но не эквивалентная ей
- Возможны различные варианты f(k) и h(k). Например, если N — простое число и **f(k) = k%N**, то можно взять **h(k) = 1 + k%(N - 2)**, т.е. $\mathbf{a_i = g(f(k), i) = k\%N + i*(1 + k*\%(N - 2))}$
- В этом случае, функции f(k) и h(k) являются независимыми в том смысле, что на различных ключах значения обеих функций f и h совпадут с вероятностью $\mathbf{O(1/N^2)}$, а не **O(1/N)**
- **Среднее число проб** при неудачном поиске **1/(1 - n/m)**, при удачном **1/k*ln(1/(1 - n/m))**, где **n/m** — коэффициент заполнения таблицы (нагрузки)
- Если функция h(k) зависит от f(k), то **вторичные скучивания** вызывают увеличение средних значений.

12. Поиск образца в тексте

Постановка задачи



Дано:

- Некоторый **текст T** (haystack) и **образец** или шаблон **W** (needle) – тоже текст (подстрока)
- Или формально: Пусть заданы массивы T и W из **n** и **m** символов соответственно, причем **0<m≤n**
- Элементы массивов T и W – символы некоторого **конечного алфавита** – например: {0, 1}, или {a, …, z}, или {а, …, я}.

Результат:

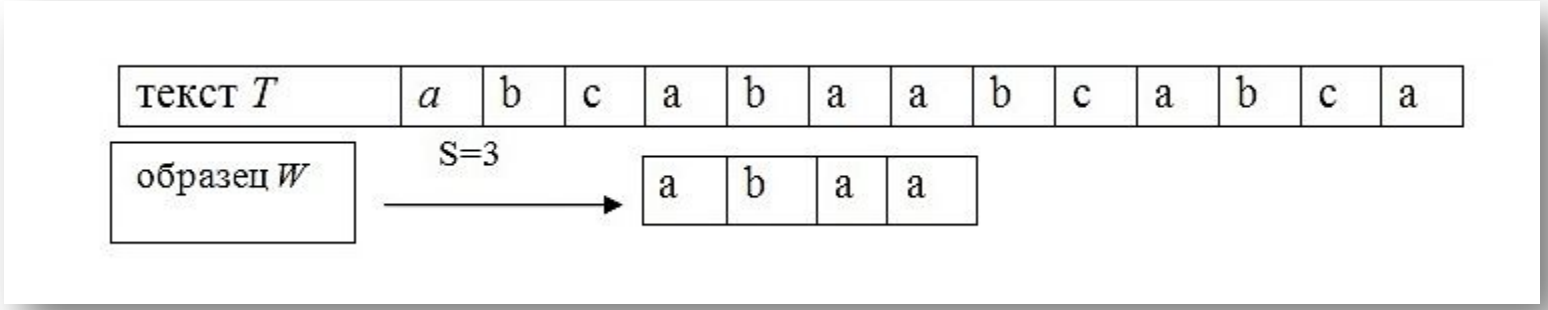
- Найти **первое слева вхождение** этого образца в указанный текст, т. е. сообщить о нахождении и, возможно, вернуть **индекс**, начиная с которого образец присутствует в тексте.

Линейный (последовательный, прямой) поиск

- Примитивный, **наивный** алгоритм (англ. brute force algorithm)

Пример:

- Требуется найти все вхождения образца W = **abaa** в тексте T = **abcabaabcabca**
- Со сдвигом **S** от **0** до **|T| – |W|** посимвольно сравниваются **T** и **W**
- Образец входит в текст со сдвигом S=3, индекс i=4:



Здесь и далее **сдвиг** – **не физический** в памяти, а приращение значения индексной переменной.

Псевдокод алгоритма прямого поиска

```
findstr (char *s, char *temp) do
   //i – индекс в строке s, j – индекс в подстроке temp, ls – длина s, lt – длина temp
   i ← 0, j ← 0, f ← 0, ls ← length(s), lt ← length(temp);
   if (ls < lt) возврат -1;
   while (i < ls-lt)          //сдвиг по s не дальше, чем ls - lt
   do
      f ← i;                  //начальный индекс возможного вхождения
      while ((j<lt) and (s[i]==temp[j])) do //сравнение до конца образца
         i ← i+1; j ← j+1;
      od
      if (j = lt) возврат f;     // удачный поиск
      j ← 0; i ← f+1;   //сдвиг++ и возврат к началу подстроки
   od
   возврат -1;  // неудачный поиск
od
```

Сложность алгоритма прямого поиска подстроки

- **Худший случай** – обнаружение образца **в конце текста**:

T→{AAA….AAAB}, длина **|T| =n**; **W→{A….AB}**, длина **|W| =m**

- Для обнаружения совпадения в конце строки потребуется произвести порядка **n·m** сравнений, то есть **O(n·m)**
- Недостатки наивного алгоритма:

1. Алгоритм «**грубой силы**» → высокая сложность: **O(n·m)** в худшем, O(n-m+1) в лучшем, O(2·n) в среднем;
2. После несовпадения просмотр всегда начинается с первого символа **W** – если образец читается из внешней памяти, то такие **возвраты занимают много времени**;
3. Информация о тексте **T**, получаемая при проверке со

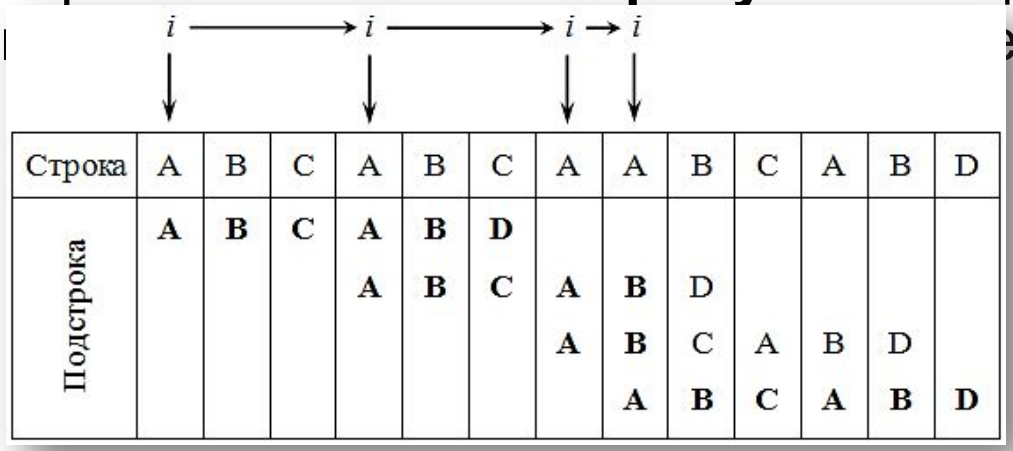
Улучшение эффективности поиска подстроки

- **Цель** улучшений – по возможности при поиске сдвинуть образец на >1 позицию
- Механизм – **препроцессинг** – это предварительная обработка **образца**, позволяющая учесть частичные совпадения с текстом
- Реализации:

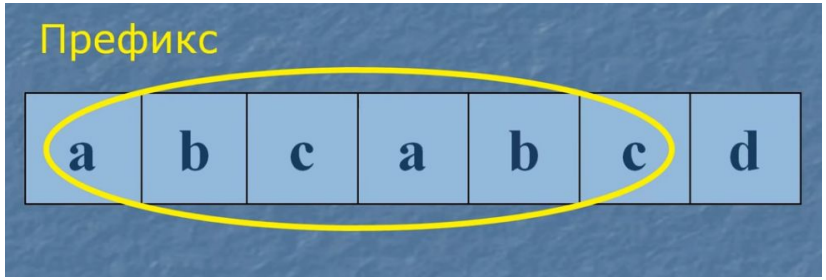
1. Алгоритм **Кнута-Морриса-Пратта** →
2. Алгоритм **Бойера-Мура**
3. Алгоритм **конечного автомата**.

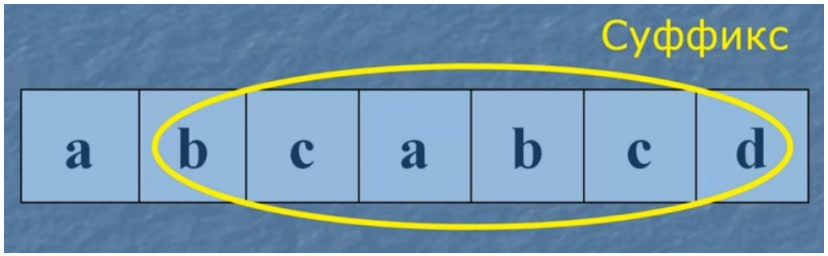
1. Алгоритм Кнута-Морриса-Пратта (КМП)

- 1970-е, Д.Кнут и В.Пратт и, независимо от них, Д. Моррис
- При **каждом несовпадении** пары символов <u>важно</u>:
  - Обеспечить **максимально допустимый** сдвиг образца
  - Гарантированно **избежать пропуска** вхождений – велич ельно.



Префикс и суффикс



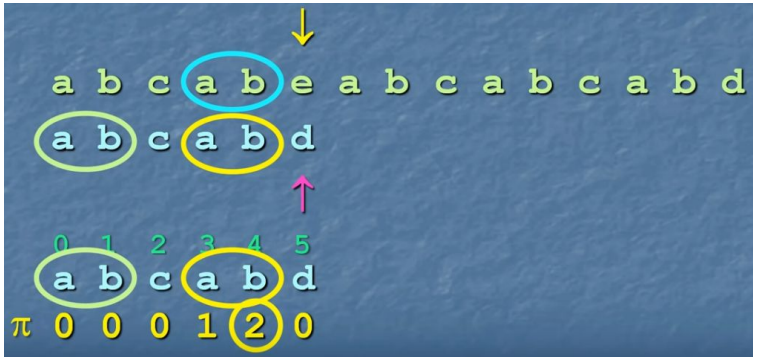


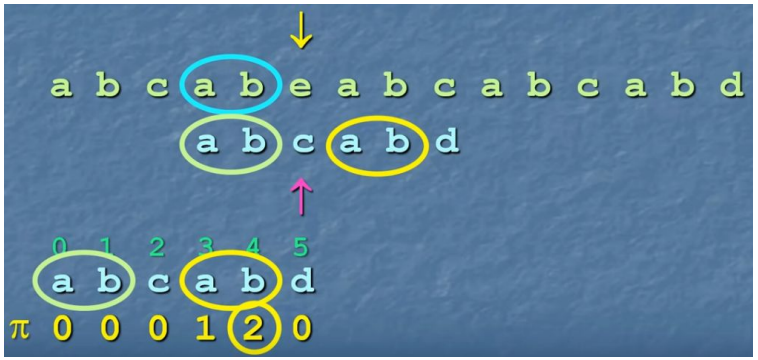
- **Префикс** – начальные символы последовательности (с начального по максимум предпоследний)
- **Суффикс** – окончание последовательности символов
- **Собственные** префикс и суффикс – не могут быть всей строкой.

Идея расчёта сдвига (1/2)

- Организуется **посимвольное сравнение** текста и образца вплоть до первого несовпадения
- **Суффикс образца** (без последнего не совпавшего символа) – зона **частичного совпадения** образца с текстом – важен при расчёте сдвига
- Пусть **суффикс** образца **совпадает** с его **префиксом**
- Тогда образец можно **сдвинуть вправо** так, чтобы его **префикс расположился под** зоной совпадения в основном тексте
- Т.о. сдвиг образца может быть **больше, чем на 1** символ
- Посимвольное сравнение можно **продолжить** с **точки несовпадения** в основном тексте (без возврата).

Идея расчёта сдвига (2/2)





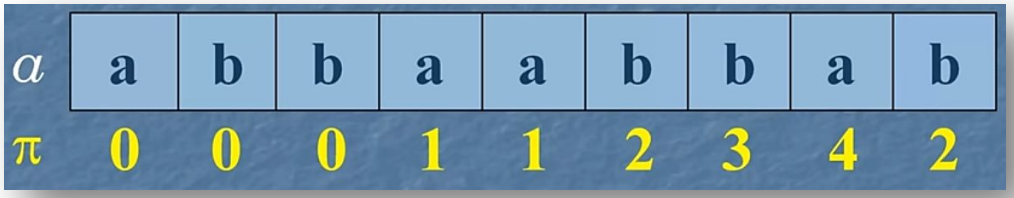
- Индекс последнего совпавшего с текстом символа в образце должен стать ключом к **длине суффикса, равного префиксу** совпавшей части образца
- Тогда этап, предваряющий сам поиск (**препроцессинг**) – создание массива (**вектора**) длиной, равной длине образца, в котором каждое i-е значение есть длина **наибольшего** собственного **префикса** части образца, одновременно являющегося его **суффиксом**
- Такой вектор (**π**) называется

Префикс-функция

Пусть дана строка (образец) **a[1..m]**

Требуется вычислить для неё **префикс-функцию**, т.е. массив (**вектор**) чисел **π[1..m]**, где:

- **π[i]** определяется как **наибольшая** длина собственного суффикса подстроки **a[1..i]**, совпадающего с её префиксом
- В частности, значение **π[1]** полагается равным 0.



1 этап – Препроцессинг – формирование массива префиксов

- Пусть **j = π[s, i-1]**. Вычисление префикс-функции для **i**:

1. При **s[i] == s[j]** определить **π[s, i] = j+1**.
2. Иначе при **j==0** определить **π[s, i] = j** (т.е. 0)
3. Иначе установить **j = π[s, i-1]** и перейти к п.1.

- Алгоритм требует не более 2·|s| итераций, т.о. для шаблона поиска длиной **m** сложность составит **O(m)**.

```
9   vector<size_t> prefix_function(string s) {
10  //префикс-функция для строки s
11      size_t n = s.length();
12      size_t j;
13      vector<size_t> pi(n); // префикс-функция, p[0]=0 всегда
14      for (size_t i = 1; i < n; ++i) {
15          // ищем, какой префикс-суффикс можно расширить
16          j = pi[i - 1]; // длина предыдущего префикса-суффикса, возможно нулевая
17          while ((j > 0) && (s[i] != s[j])) // этот нельзя расширить,
18              j = pi[j - 1];   // берем длину меньшего префикса-суффикса
19          if (s[i] == s[j])
20              ++j;  // расширяем найденный префикс-суффикс
21          pi[i] = j;
22      }
23      return pi; //возврат результа
24  }
```

a	a	b	b	a	a	b	b	a	b
π	0	0	0	1	1	2	3	4	2

2 этап – сам КМП-поиск (вариант реализации на C++)

```
29  void KMP_string_serch (string str, string obr, vector<size_t> pi) {
30      //Поиск подстроки obr в тексте str с помощью префикс-функции pi
31      size_t t = str.length();    // длина текста
32      size_t l = obr.length();    // длина образца
33      size_t j = 0; // количество совпавших символов = индекс сравниваемого символа в образце
34      for (size_t i = 0; i < t; ++i) { // В строке сравниваемый символ - индекс i
35          while ((j > 0) && (str[i] != obr[j]))
36              // Очередной символ строки не совпал с символом в образце. Сдвигаем образец
37              j = pi[j - 1];    // если j=0, то достигли начала образца - цикл следует прервать
38          if (str[i] == obr[j]) // есть совпадение очередного символа
39              ++j;              // увеличиваем длину совпавшего фрагмента на 1
40          if (j == l) {
41              cout << "Индекс подстроки (с нуля): " << i - l + 1 << "\n"; return;
42          }
43      }
44      cout << "Подстроки нет!" << "\n"; return;
45  }

47  int main() {    //начало программы
48      SetConsoleCP(1251); // установка кодовой страницы Windows-1251 в поток ввода
49      SetConsoleOutputCP(1251); // установка кодовой страницы Windows-1251 в поток вывода
50
51      string t = "Мама мыла раму", s = "ы";  //текст и шаблон поиска
52      vector<size_t> pi = prefix_function(s); //препроцессинг
53      KMP_string_serch(t, s, pi); //поиск
54
55      system("PAUSE");
56      return 0;   //нормальное завершение работы программы
57  }
```

Приём в реализации алгоритма КМП

- Склеим образец **aabaa** и основной текст **aabaabaaaabaabaaab** через символ-разделитель:

aabaa@aabaabaaaabaabaaab

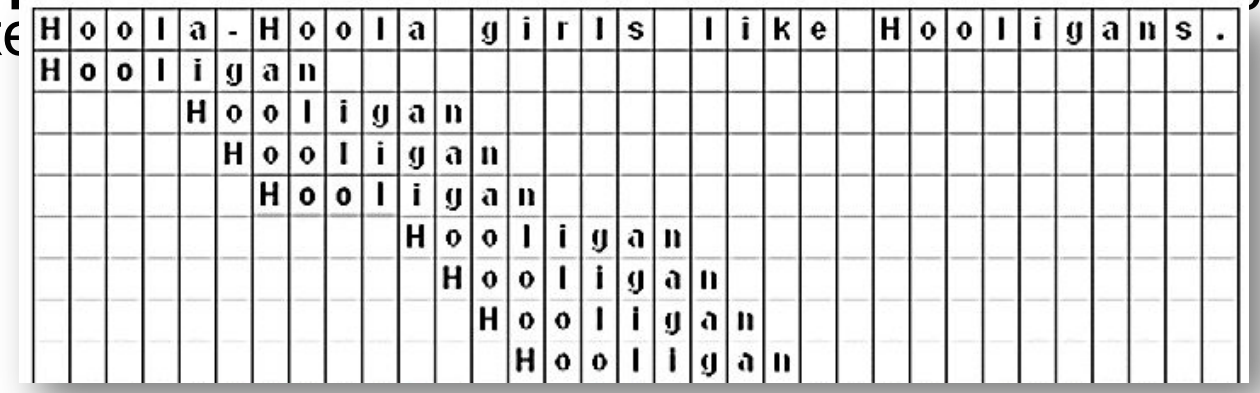
- Вызовем **для всей склейки** префикс-функцию:

a	a	b	a	a	@	a	a	b	a	a	b	a	a	a	a	b	a	a
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	0	1	2	0	1	2	3	4	5	3	4	5	2	2	3	4	5

- Тогда достаточно последовательным **просмотром** части этого массива **после разделителя** найти **позиции** со значениями, равными **длине образца**
- Здесь будут **индексы вхождений** образца в основной текст

Особенности алгоритма КМП

- Движение по основному тексту – **только вперёд** (без возврата при нахождении несоответствия) – это преимущество при работе с данными во **внешней памяти**
- Сложность **O(n+m)** – **сублинейна** – даже с учётом префикс-функции (**O(n+m)** < **O(n·m)** простого поиска)
- **Затраты времени** на префикс-функцию окупаются при поиске, если неудаче предшествовало некоторое количество **совпадений**
- Вероятность совпадения ниже несовпадения → на практике **выигрыш** при использовании КМП-поиска **невелик** (пример: 17 смещений вместо 23 в простом поиске



- КМП-поиск – это основа для алгоритма **Ахо-Корасика**.

2. Алгоритм Бойера-Мура (БМ)

- Как и в КМП-поиске, ценой предварительной работы (препроцессинга) над образцом **пропускается часть проверок** на этапе поиска
- Идея:
  1. Прикладывание образца к тексту и его **сдвиг** слева вправо до успеха или до достижения конца строки (неуспеха)
  2. Сравнение образца **справа налево**
  3. **Правило** сдвига по **стоп-символу** («плохому», несовпавшему) →
  4. **Правило** сдвига по **совпавшему** («хорошему», безопасному) **суффиксу** образца
  5. Сдвиг образца на **наибольшее** из этих двух значений.
- **Совместный эффект** пунктов 2-4 даст **<(n+m)** сравнени