БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Организация поиска

Хеширование

Словарные операции

1) поиск элемента x;
2) добавление нового элемента x;
3) удаление элемента x.

Структуры данных для выполнения словарных операций

1. массив;
2. поисковые деревья;
3. хеш-таблицы.

Существуют интересные гибриды, находящиеся посередине между деревьями поиска и хеш-таблицами, которые реализуют концепцию упорядоченного множества. Это всевозможные деревья Ван Эмде Боасса (Van Emde Boas tree), X-fast-, Y-fast- и Fusion-деревья, у которых в оценках временной сложности появляется двойной логарифм.

Структуры данных на основе хеш-таблиц реализованы в стандартных библиотеках всех широко используемых языков программирования.

Абстрактный тип данных: множество (set)

C++	JAVA	Python
контейнер std::set, который реализует множество на основе сбалансированного дерева (обычно красно-чёрного) контейнер **std::unordered_set, построенный на базе хеш-таблицы**	интерфейс Set, у которого есть несколько реализаций, среди которых классы TreeSet (работает на основе красно-чёрного дерева) интерфейс **HashSet (на основе хеш-таблицы)**	нет готового класса множества, построенного на сбалансированных деревьях. встроенный тип **set, использующий хеширование**

Абстрактный тип данных: ассоциативный массив (map)

C++	JAVA	Python
в стандартной библиотеке C++: контейнер std::map, работающий на основе сбалансированного дерева (обычно красно-чёрного) **контейнер std::unordered_map, работающий на основе хеш-таблицы.**	интерфейс Map, который реализуется несколькими классами, в частности классом TreeMap (базируется на красно-чёрном дереве) и **HashMap (базируется на хеш-таблице)**	встроенный тип **dict (**этот словарь используют внутри хеширование)

Устройство хеш-таблицы

Для простоты будем считать, что ключи являются целыми числами из диапазона **[0, *N*)**.

Обозначим через ***K*** множество возможных ключей: ***K* = {0, 1, 2, . . . , N − 1}**.

На практике множество ***K*** обычно довольно большое.

Часто в качестве ключей в промышленном программировании применяются 32-битные или 64-битные целые числа, т. е.

$$N = 2^{32} \approx 4{,}2 \cdot 10^{9}$$

или

$$N = 2^{64} \approx 1{,}8 \cdot 10^{19} .$$

1. Прямая адресация

$K = \{0, 1, 2, \ldots, N - 1\}$

Если достаточно памяти для массива, число элементов которого равно числу всех возможных ключей, то для каждого возможного ключа можно отвести ячейку в этом массиве.

Имеем булев массив `T` размера `N`, называемый **таблицей с прямой адресацией**, в котором **элемент** t_i содержит истинное значение, если ключ `i` входит в множество, и ложное значение, если ключ `i` в множестве отсутствует.

T:

0	1	2	3	...	N-2	N-1
True	False	True	False	...	False	True

базовые операции:

- добавление ключа;
- проверка наличия ключа;
- удаление ключа.

O(1)

Недостатки прямой адр

$K = \{0, 1, 2, \ldots, N - 1\}$

T

0	1	2	3	…	N-2	N-1
True	False	True	False	…	False	True

:

- ✔ размер таблицы с прямой адресацией не зависит от того, сколько элементов реально содержится в множестве;
- ✔ если число реально присутствующих в таблице записей мало по сравнению с **N**, то много памяти тратится зря;
- ✔ если множество **K** всевозможных ключей велико, то хранить в памяти массив **T** размера **N** непрактично, а то и невозможно:

> Минимальным адресуемым набором данных в современных компьютерах является один байт, состоящий из восьми битов.
> Не представляет трудности реализовать таблицу с прямой адресацией так, чтобы каждый бит был использован для хранения одной ячейки.
> Если N — мощность множества возможных ключей, то для прямой адресации требуется выделить последовательный блок из как минимум N бит памяти. Так, **для размеров множества K в 10^9 элементов таблица займёт около 120 МБ памяти** (10^9 бит≈$1{,}2*10^8$ байт=$120*10^6$ байт=120Мбайт).
> Во многих случаях такой расход памяти неприемлем, особенно когда есть необходимость создавать несколько таблиц.

Тем не менее при сравнительно небольших N метод прямой адресации успешно используется на практике.

$K = \{0, 1, 2, \ldots, N - 1\}$.

2. Хеш-функция (англ. *hash function*)

Введём некоторую функцию, называемую **хеш-функцией**, которая отображает множество ключей в некоторое гораздо более узкое множество:

$$h : \{0, 1, 2, \ldots, N - 1\} \rightarrow \{0, 1, \ldots, M - 1\},$$

Величина $h(x)$ называется **хеш-значением** (англ. hash value) ключа **x**.

Далее вместо того, чтобы работать с ключами, мы работаем с хеш-значениями.

Если разные ключи получают одинаковые хеш-значения: $x \neq y$, $h(x) = h(y)$, то говорят, что произошла **коллизия** (англ. collisions).

Хотелось бы выбрать хеш-функцию так, чтобы коллизии были невозможны. Но в общем случае при $M < N$ это неосуществимо: согласно принципу Дирихле (1834 г.), нельзя построить инъективное отображение из большего множества в меньшее.

$h(x) \equiv 0$

любая пара различных ключей будет давать коллизию;

такая хеш-функция бесполезна несмотря на то, что она простая и быстро вычисляется;

$h(x) = rand(M)$

всякий раз возвращает случайное число от 0 до M − 1 включительно, выбранное равновероятно независимо от x

не может быть использована как хеш-функция, потому что хеш-функция обязана для равных ключей возвращать одинаковые значения;

деление с остатком

$h(x) = x \bmod M$

возвращает остаток от деления ключа x на M

является вполне годной для практики хеш-функцией и часто применяется;

если ключи возникают, как десятичные числа, то нежелательно выбирать в качестве *M* степень 10 (т.к. в этом случае окажется, что часть цифр числа уже полностью определяют хеш-значение);

в качестве *M* предпочтителен выбор простого числа, далеко отстоящего от степени 2;

умножение

$h(x) = \lfloor M \cdot (x \cdot A \bmod 1) \rfloor$

$x \cdot A \bmod 1$ - дробная часть числа $x \cdot A$

в качестве *M* выбирают степень 2;

утверждают, что наиболее удачное значение константы *A* = 0,6180339887 (золотое сечение).

Велика ли вероятность коллизий?

каждый элемент может принять
одно из M значений

0-й ключ	1-й ключ	2-й ключ	3-й ключ	…	(n-1)-й ключ
1	4	1	9	…	8

Пусть осуществляется хеширование для **n** различных ключей, т.е. мы **строим вектор** длины **n**, где назначаем каждому элементу одно из **M** значений (предположим, что хеш-значения независимы и распределены идеально равномерно от 0 до **M** – 1).

Число векторов длины **n**, которые могут быть при этом сгенерированы:

$$M^n$$

Число векторов длины **n**, которые могут быть при этом сгенерированы и в которых элементы не повторяются:

$$M \cdot (M-1) \cdot (M-2) \cdot \ldots \cdot (M-n+1) = \frac{M!}{(M-n)!}$$

Вероятность того, что все элементы сгенерированного вектора различны, т.е. **нет коллизий**:

$$\frac{M!}{(M-n)!} : M^n$$

Вероятность того, что **будут коллизии**:

$$1 - \frac{M!}{(M-n)! \cdot M^n}$$

Пусть осуществляется хеширование для **n** ≪ **M** различных ключей (предположим, что хеш-значения независимы и распределены идеально равномерно от 0 до **M** – 1).

Используя приближенную формулу для значения факториала (Стирлинга), когда **M** велико, а **n** ≪ **M** получим приближенную формулу вероятности коллизии:

$$1 - \frac{M!}{(M-n)!\,M^n} \approx \mathbf{1} - \boldsymbol{e}^{-\frac{\boldsymbol{n^2}}{\boldsymbol{2 \cdot M}}}$$

Несложно увидеть, что уже при при $n \approx \sqrt[2]{M}$ с вероятностью $\simeq 50\%$, будут коллизии.

Формула Стирлинга: $n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$

Ряд Тейлора для экспоненты: $e^x = 1 + x + \frac{x^2}{2!} + \cdots \simeq 1 + x$, аппроксимация для $|x| \ll 1$.

Пусть **M** = 10^6 и хеширование выполняется для **n = 2 450** уникальных ключей, тогда с вероятностью ≈ **95%** найдутся такие два ключа, что их хеш-значения будут одинаковыми, т.е. будет иметь место коллизия.

Пусть p – вероятность того, что будет коллизия.

$$p = 1 - e^{-\frac{n^2}{2 \cdot M}}$$

$$e^{-\frac{n^2}{2 \cdot M}} = p - 1$$

$$e^{\frac{n^2}{2 \cdot M}} = \frac{1}{p - 1}$$

$$\frac{n^2}{2 \cdot M} = \log_e \frac{1}{1 - p}$$

$$n^2 = 2 \cdot M \cdot \log_e \frac{1}{1 - p}$$

$$n = \sqrt[2]{2 \cdot M \cdot \log_e \frac{1}{1 - p}}$$

или

$$n = \sqrt[2]{2 \cdot M \cdot \log_e \frac{1}{p'}}$$

где p' – желаемая вероятность того, чтобы не было коллизий

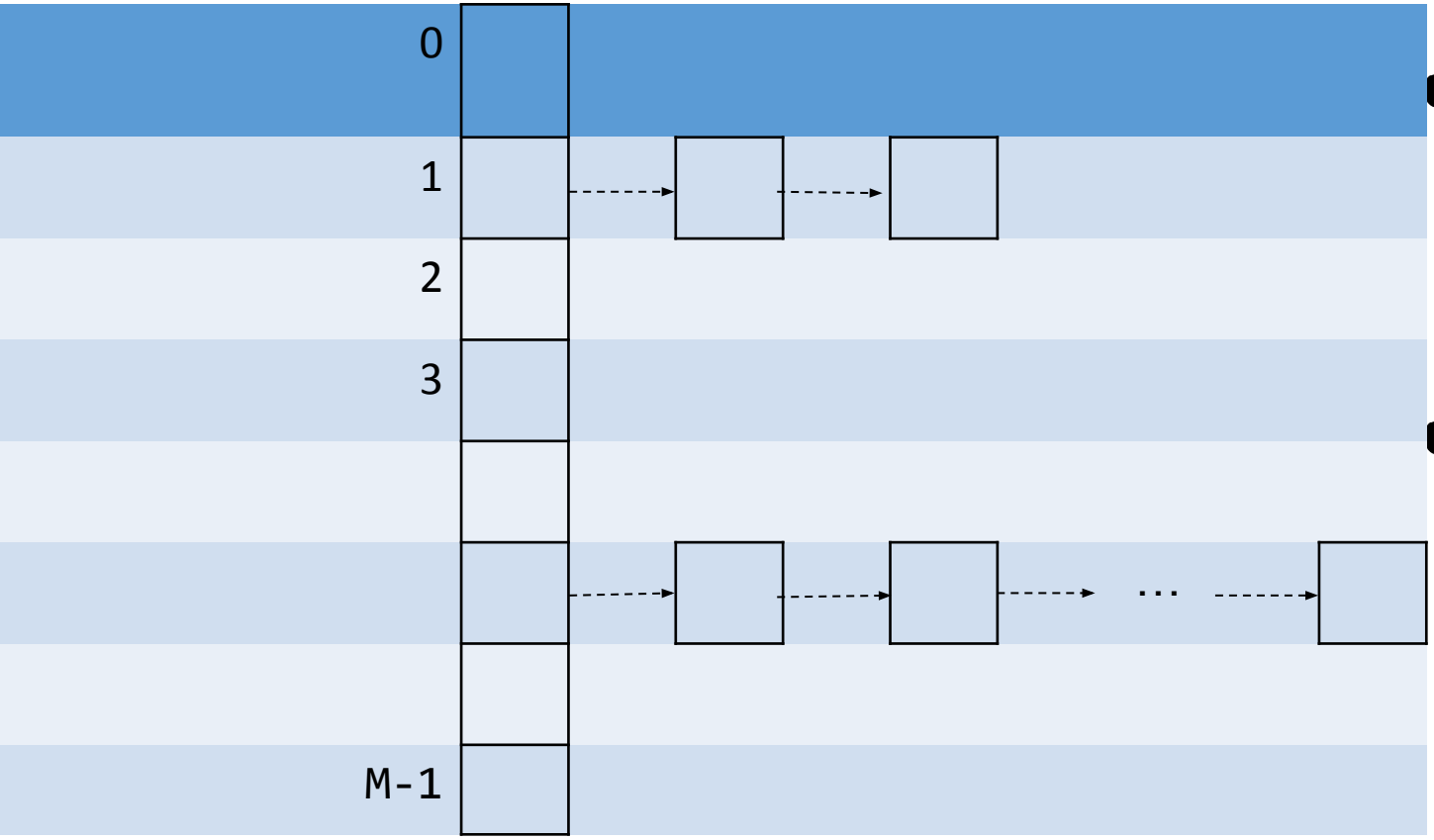
Разработано несколько стратегий разрешения коллизий

Разрешение коллизий **методом цепочек**
(англ. separate chaining)

Разрешение коллизий **методом открытой адресации**
(англ. open addressing)

Разрешение коллизий **методом цепочек** (англ. separate chaining)

Хеш-функция **h** раскладывает исходные ключи **x** по **корзинам** (англ. bins, buckets) или **слотам** (англ. slots).



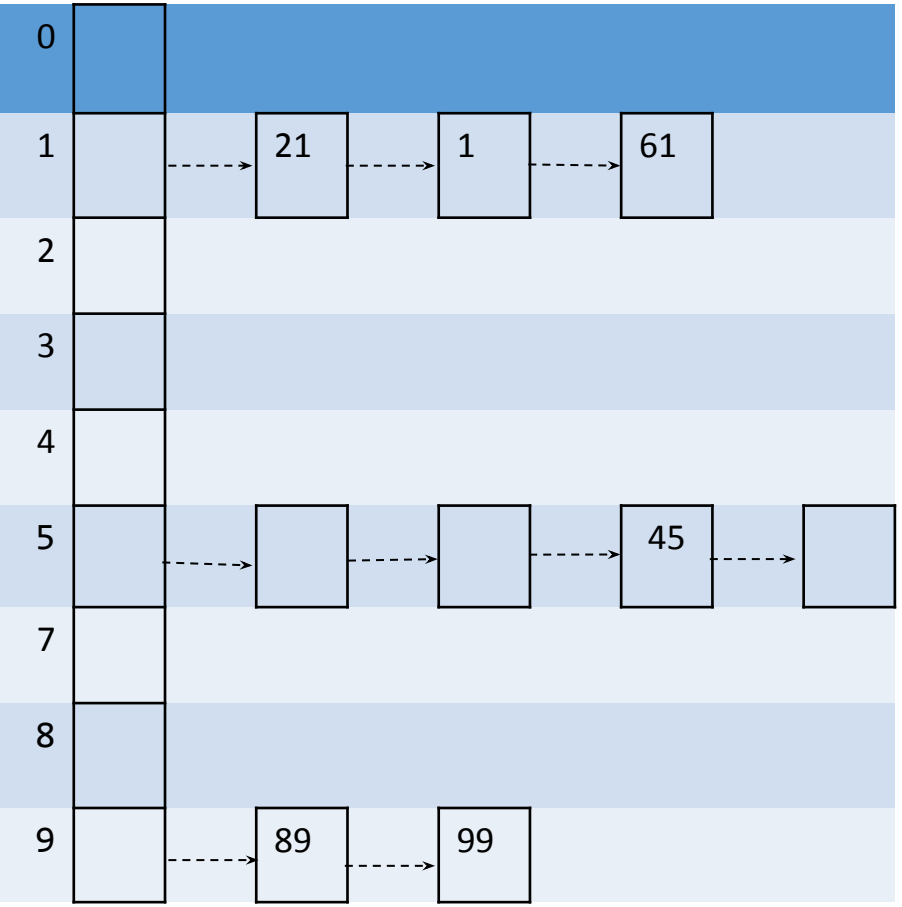
- ключ **x** попадает в корзину с номером, равным хеш-значению **h(x)**;
- для хранения элементов с одинаковыми хеш-значениями внутри одной корзины можно использовать связные списки;

На верхнем уровне организуется массив размера **M** — по числу различных значений хеш-функции, каждый элемент которого — это односвязный список, состоящий из ключей, имеющих конкретное хеш-значение. Возникают цепочки ключей, из-за чего метод и получил название метода цепочек.

Операция вставки элемента

K={21,1,89,5,61,55,45,15,99}

h(x)=x mod 10



Сначала вычисляется хеш-значение h(x) для ключа x, а затем происходит обращение к соответствующему связному списку.

Если НЕ СТОИТ ЗАДАЧА ПРОВЕРЯТЬ, ПРИСУТСТВУЕТ ЭЛЕМЕНТ x В ТАБЛИЦЕ ИЛИ НЕТ, то операция вставки может быть реализована за константное время: всегда можно добавить элемент в начало списка, и не нужно идти по всему связному списку.

Если требуется поддерживать УНИКАЛЬНОСТЬ ЭЛЕМЕНТОВ, то сначала надо проверить, есть элемент x в таблице или нет, и добавлять только уникальные элементы. Поэтому операция вставки вначале выполняет проход по списку, и на это расходуется время, пропорциональное длине соответствующей цепочки.

Поддерживать уникальность ключей удобно в силу ряда причин:

(1) можно легко отвечать на запросы о числе элементов в множестве;

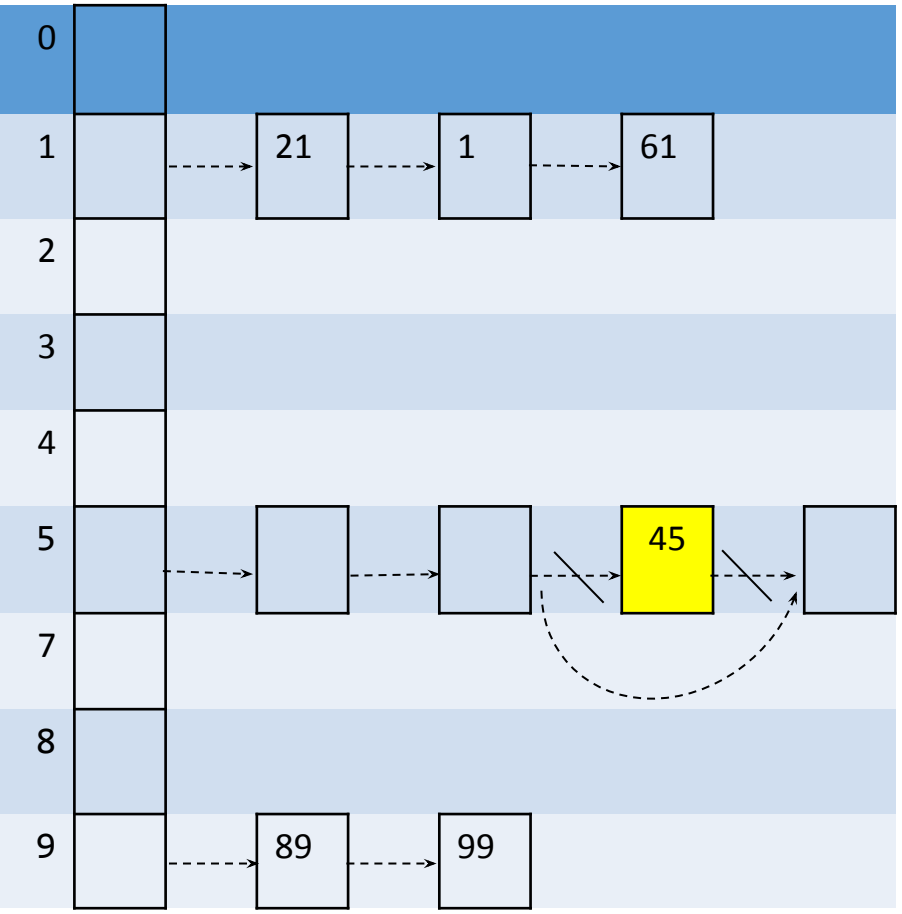
(2) меньше расход памяти (нередко на практике вставок выполняется много, но среди ключей мало различных);

(3) проще организовать удаление ключа

ФПМИ БГУ

Операция удаления элемента

h(x)=x mod 10
x=45



Вычисляем сначала хеш-значение для ключа x: h(45)=5.

Затем выполняем прохождение соответствующего списка в поиске элемента x.

В общем случае из односвязного списка удалить элемент из середины сложно. Однако в рассматриваемом случае, несмотря на то, что список односвязный удалять из него нетрудно:

- ✔ так как мы движемся слева направо, то можем поддерживать указатель на текущий элемент и на предыдущий;
- ✔ при удалении указатель у предыдущего элемента перенаправляется на следующий элемент, а память из-под текущего элемента освобождается.

Таким образом, производительность всей конструкции связана с таким параметром, как длина цепочки.

Пусть $l_0, l_1, \ldots, l_{M-1}$ – длины цепочек (для каждого хеш-значения длина цепочки своя).

Каждая из базовых операций с ключом **x** требует времени $O(1 + l_i)$, где l_i — длина цепочки, в которую попадает ключ **x**.

ВНИМАНИЕ

Даже если цепочка имеет нулевую длину, то требуется выделить время на то, чтобы вычислить хеш-значение (мы полагаем, что хеш-функция от ключа вычисляется за константу) и обратиться к соответствующей цепочке.

Разрешение коллизий **методом открытой адресации**
(англ. open addressing)

В линейном массиве хранятся непосредственно ключи, а не заголовки связных списков.

0	1	2	3	4	5	6	7	8	9
38	**27**	**18**			**25**		**7**	**67**	**29**

В каждой ячейке массива разрешено хранить только один элемент.

Что делать, если произошла коллизия?

Последовательность проб

(англ. probe sequence)

Обозначим через **h(x,i)** номер ячейки в массиве, к которой следует обращаться на **i**-й попытке при выполнении операций с ключом **x**.

Будем нумеровать попытки с **0** для каждого вновь поступающего элемента.

Последовательность проб для ключа **x** получается такой:

h(x,0),
h(x,1),
h(x,2),
...

Для успешной работы алгоритмов поиска последовательность проб должна быть такой, чтобы в результате **M** проб все ячейки хеш-таблицы оказались просмотренными ровно по одному разу в каком-либо порядке:

$$\forall x \in K \{h(x,i) \mid i = 0,1,...,M-1\} = \{0,1,...,M-1\}.$$

Широко используются три вида последовательностей проб:

1. линейная;
2. квадратична
3. двойное хеширование.

Линейное пробирование

Ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом c между ячейками:

$$h(x, i) = (h'(x) + c \cdot i) \bmod M,$$

где $h'(x)$ — некоторая хеш-функция.

Функция $h(x,i)=(x+2\cdot i) \bmod 10$ **НЕ ПОДХОДИТ** в качестве последовательности проб.

Например, при **x** = 5, подставляя i от 0 до 9, будем получать индексы:

5,7,9,1,3,5,7,9,1,3

в результате чётные позиции оказываются не посещены.

Функция $h(x,i)=(x+3\cdot i) \bmod 10$ **ПОДХОДИТ** в качестве последовательности проб

Например, при x = 5, подставляя i от 0 до 9, будем получать индексы (каждый индекс возвращается один раз):

5,8,1,4,7,0,3,6,9,2

$$h(x, i) = (h'(x) + c \cdot i) \bmod M$$

Теорема

Для того чтобы в ходе M проб все ячейки таблицы оказались просмотренными по одному разу, необходимо и достаточно, чтобы число c было взаимно простым с размером хеш-таблицы M.

В простейшем случае можно взять единицу в качестве константы c

$$h(x,i) = (h'(x)+i) \bmod M$$

Теорема

> Для того чтобы в ходе M проб все ячейки таблицы оказались просмотренными по одному разу, необходимо и достаточно, чтобы число c было взаимно простым с размером хеш-таблицы M.

Доказательство.

Воспользуемся сведениями из элементарной теории чисел.

Докажем необходимость

Пусть $h(x, i)$ пробегает все значения от 0 до $M - 1$. Значит, для любого t найдётся такой индекс i, что $c \cdot i \equiv t \pmod{M}$. В частности, это верно для $t = 1$. Следовательно, есть такое i, что $c \cdot i \equiv 1 \pmod{M}$, или, другими словами, $c \cdot i - 1$ делится на M. Пусть d — общий делитель чисел c и M. Тогда число 1 также вынуждено делиться на d. Значит, $d = \pm 1$, т.е. числа c и M взаимно просты и не могут иметь других общих делителей.

Докажем достаточность

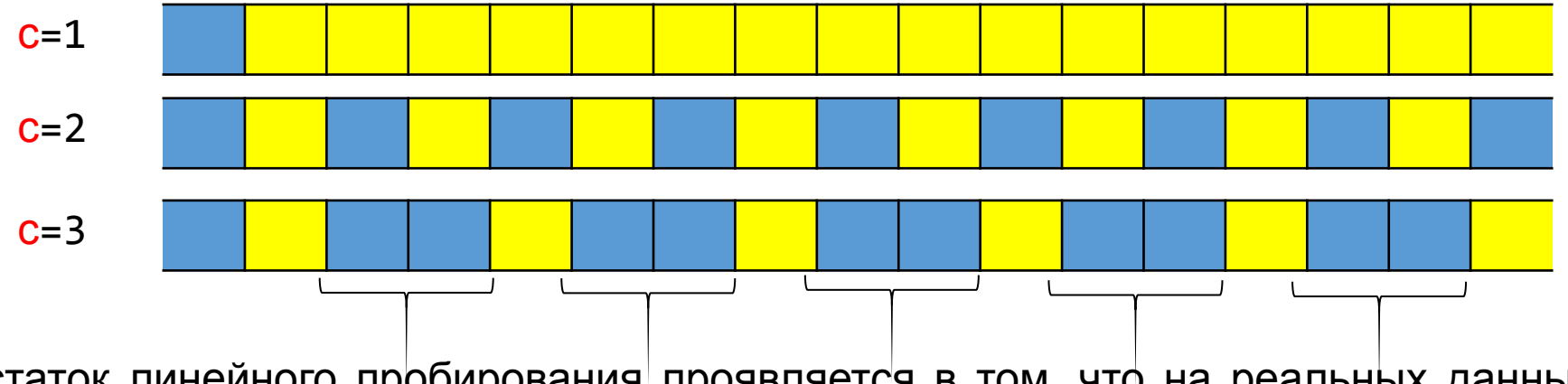
Пусть числа c и M взаимно просты. Предположим от противного, что при i-й и j-й пробах получаются одинаковые индексы: $h(x, i) = h(x, j)$. Но это значит, что $c \cdot i \equiv c \cdot j \pmod{M}$, или

$c \cdot (i - j) \equiv 0 \pmod{M}$. Отсюда следует, что разность $i - j$ делится на M без остатка. Но раз номера попыток i и j оба лежат на отрезке от 0 до $M - 1$, то это возможно лишь в случае, когда $i = j$. Противоречие. Значит, при всех попытках пробы получаются разными. Поскольку попыток всего M, каждая ячейка будет учтена по одному разу.

Линейное пробирование:

$$h(x,i) = (h'(x) + c \cdot i) \bmod M$$

Ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом c между ячейками:



Недостаток линейного пробирования проявляется в том, что на реальных данных часто образуются **кластеры из занятых ячеек** (длинные последовательности ячеек, идущих подряд).

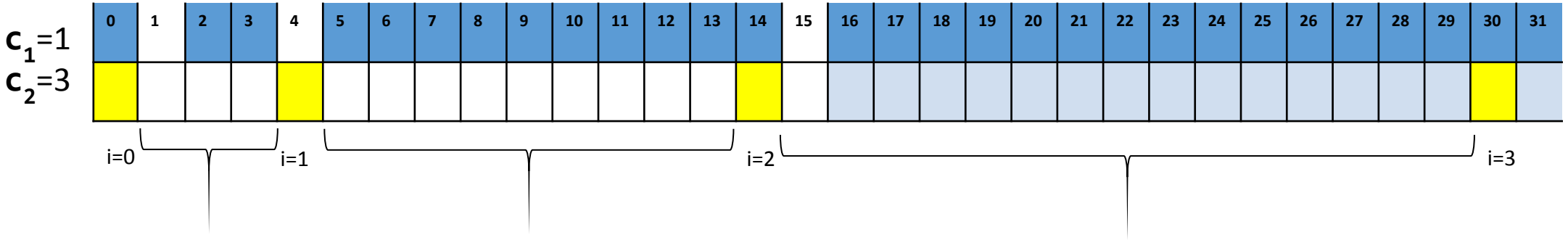
При непрерывном расположении заполненных ячеек увеличивается время добавления нового элемента и других операций.

Квадратичное пробирование:

$$h(x,i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod M,$$

где числа c_1 и c_2 фиксированы (эти значения должны быть тщательно подобраны, чтобы в результате M попыток все ячейки были посещены, например, если M=11, то можно взять c_1=1, c_2=3).

Номер ячейки квадратично зависит от номера попытки, расстояние между ячейками с каждой итерацией увеличивается на константу ($=2 \cdot c_2$).



На практике такой метод часто работает лучше линейного, разбрасывая ключи более равномерно по массиву. Тенденции к образованию кластеров нет, но аналогичный эффект проявляется в форме образования **вторичных кластеров** (ситуация, когда на начальном этапе различные элементы получают одно и тоже хеш-значение, что приводит к тому, что каждый последующий из этих элементов проходит путь предшествующего).

Двойное хеширование:

$$h(x,i) = (h'(x) + h''(x) \cdot i) \bmod M$$

Последовательность проб при работе с ключом x представляет собой арифметическую прогрессию (по модулю M) с первым членом $h'(x)$ и шагом $h''(x)$.

Для того, чтобы последовательность проб покрыла всю таблицу, значение $h'(x)$ должно быть ненулевым и взаимно простым с M.

Для этого можно поступить следующим образом:

- выбрать в качестве M степень двойки, а функцию $h''(x)$ взять такую, чтобы она принимала только нечётные значения;
- выбрать в качестве M простое число и потребовать, чтобы вспомогательная хеш-функция $h''(x)$ принимала значения от 1 до $M - 1$.

Операция вставки элемента

1) Перед вставкой ключа **x** выполняется поиск этого ключа.

   Если такой элемент **x** уже есть в таблице, то вставка не требуется.

2) Затем проверяется принципиальное наличие ячеек, не занятых ключами (счётчик занятых ячеек).

3) Если есть свободные ячейки, то осуществляется пробирование, начиная с того места, в которое указывает хеш-функция, в соответствии с некоторой последовательностью проб, пока не будет найдено свободное место в хеш-таблице.
4) В свободную ячейку заносится **x**, а счётчик занятых ячеек увеличивается на 1.

Например, при линейном пробировании:

h(x,i) = (x+i) mod M, M=10

для последовательности элементов:

7, 29, 67, 38, 25, 27, 18

Хеш-таблица будет иметь следующий вид:

0	1	2	3	4	5	6	7	8	9
38	27	18			25		7	67	29
							67	38	38
27							27	27	27
18	18							18	18

h(7, **0**)=7 - свободна – заносим 7

h(29, **0**)=9 - свободна – заносим 29

h(67, **0**)=7 - занята

h(67, **1**)=8 - свободна – заносим 67

h(38, **0**)=8 - занята

h(38, **1**)=9 - занята

h(38, **2**)=0 - свободна – заносим 38

h(25, **0**)=5 – свободна – заносим 25

и так далее

…

Название «открытая адресация» связано с тем фактом, что положение (адрес) элемента не определяется полностью его хеш-значением.

Операция поиска элемента

Случай 1.

Предположим, сначала что в таблице операция удаления элемента НЕ поддерживается.

Ячейки массива проверяются, в соответствии с той же функцией последовательности проб, которая использовалась при добавлении элементов в хеш-таблицу.

Для ячейки вводится **два** состояния:

empty — ячейка пуста;

key(x) — ячейка содержит ключ **x**.

Поиск останавливается как только будет выполнено одно из условий:

1) найден элемент **x**;
2) достигнута пустая ячейка (элемента **x** в таблице нет);
3) выполнены **M** попыток (таблица полностью заполнена, а элемента - нет).

0	1	2	3	4	5	6	7	8	9
38	27	18	empty	empty	25	empty	7	67	29
=19	=19	=19	=19						=19

$h(x, i) = (x + i) \bmod M$

M=10, x=19

СТОП, элемента x=19
НЕТ

Операция поиска элемента

Случай 2.

Предположим, что в хеш-таблице поддерживается операция удаления элемента.

h(x,i)=(x+i) mod M,
M=10

0	1	2	3	4	5	6	7	8	9
38	27	18	empty	empty	25	empty	7	67	29
=18	=18							=18	=18

СТОП, элемента x=18 НЕТ - **НЕ ВЕРНО**

1) Предположим, что из таблицы сначала удалили элемент **x**=27.

Для удаления элемента x=27 вначале выполняем его поиск и, если ячейка с элементом найдена, то:

1) переводим её в состояние свободной;
2) счётчик занятых ячеек уменьшаем на единицу.

2) Затем выполнили поиск элемента **x**=18.

Поиск останавливается как только будет выполнено одно из условий:

1) найден элемент **x**;
2) достигнута пустая ячейка (элемента **x** в таблице нет);
3) выполнены **M** попыток (таблица полностью заполнена, а элемента - нет).

$h(x, i) = (x + i) \bmod M, \quad M=10$

0	1	2	3	4	5	6	7	8	9
38	27 deleted	18	empty	empty	25	empty	7	67	29
=18	=18	=18						=18	=18

Для ячейки вводятся три состояния:

empty — ячейка пуста;

key(x) — ячейка содержит ключ x;

deleted — ячейка ранее содержала ключ, но он был удалён.

Поиск останавливается как только будет выполнено одно из условий:

1) найден элемент **x**;
2) достигнута пустая ячейка **empty** (элемента **x** в таблице нет);
3) выполнены **M** попыток (таблица полностью заполнена, но элемента **x** в таблице нет).

Теперь при удалении x=27, ячейка с номером 1 перейдет в состояние **deleted**.

Добавление нового элемента можно осуществлять как в ячейку **empty**, так и в ячейку **deleted**.

Недостатки открытой адресации

1. Нетрудно видеть, что при разрешении коллизий методом открытой адресации наличие большого числа **deleted**-ячеек отрицательно сказывается на времени выполнения операции поиска, а значит и других операций.

0	1	2	3	4	5	6	7	8	9
deleted	deleted	deleted	deleted	deleted	9	empty	7	67	deleted

Чтобы исправить ситуацию, после ряда удалений можно перестраивать хеш-таблицу заново, уничтожая удалённые ячейки.

`h(x, i) = (x + i) mod M, M=10, K={9,7,67}`

0	1	2	3	4	5	6	7	8	9
empty	empty	empty	empty	empty	empty	empty	7	67	9

2. Число хранимых ключей не может превышать размер хеш-массива (при заполнении на 70% производительность падает и нужно расширять таблицу);

3. Более строгие требования к выбору хеш-функции: чтобы распределять значения максимально равномерно по корзинам, функция должна минимизировать кластеризацию хеш-значений, которые стоят рядом в последовательности проб; при образовании больших кластеров, время выполнения всех операций может стать неприемлемым даже при том, что заполненность таблицы в среднем невысокая и коллизии редки.

Преимущества открытой адресации

1) экономия памяти, если размер ключа невелик по сравнению с размером указателя

   в методе цепочек приходится хранить в массиве указатели на начала списков, а каждый элемент списка хранит, кроме ключа, указатель на следующий элемент, поэтому на все эти указатели расходуется память;

2) не требуется затрат времени на выделение памяти на каждую новую запись и подход может быть реализован даже на миниатюрных встраиваемых системах, где полноценный аллокатор недоступен;

3) нет лишней операции обращения по указателю (`indirection`) при доступе к элементу;

4) лучшая локальность хранения, особенно с линейной функцией проб

   когда размеры ключей небольшие, это даёт лучшую производительность за счёт хорошей работы кеша процессора, который ускоряет обращения к оперативной памяти;

   однако, когда ключи «тяжёлые» (не целые числа, а составные объекты), то они забивают все кеш-линии процессора, к тому же много места в кеше тратится на хранение незанятых ячеек - можно в массиве с открытой адресацией хранить не сами ключи, а указатели на них, но при этом часть преимуществ при этом будет утрачена.

Хеш-таблицы на практике

Любой подход к реализации хеш-таблицы может работать достаточно быстро на реальных нагрузках.

Время, которое занимают операции с хеш-таблицами, обычно составляет малую долю от общего времени работы программы.

Расход памяти редко играет решающую роль.

Часто выбор между той или иной реализацией хеш-таблицы делается на основании других факторов в зависимости от ситуации.

Коэффициент заполнения (англ. load factor)

Критически важным показателем для хеш-таблицы является коэффициент заполнения— отношение числа ключей, которые хранятся в хеш-таблице, к размеру хеш-таблицы:

$$\alpha = \frac{n}{M}$$

Однако коэффициент заполнения не показывает различия между заполненностью отдельных корзин.

Низкий коэффициент заполнения не является абсолютным благом. Если коэффициент близок к нулю, это говорит о том, что большая часть таблицы не используется и память тратится впустую.

Для оптимального использования хеш-таблицы желательно, чтобы её размер был примерно пропорционален числу ключей, которые нужно хранить.

На практике редко случается, что число ключей фиксировано и можно заранее выставить хорошее значение параметра M. Если ставить его заведомо больше, то много памяти будет потрачено зря (особенно если нужно организовать много хеш-таблиц с небольшим числом ключей в каждой).

Реализация хеш-таблицы общего назначения обязана поддерживать **операцию изменения размера**.

На практике часто используемым приёмом является автоматическое изменение размера.

Когда коэффициент заполнения превышает некоторый порог α_{max}, выделяется память под новую, большую таблицу, все элементы из старой таблицы перемещаются в новую, затем память из-под старой хеш-таблицы освобождается.

Аналогично, если коэффициент заполненности опускается ниже другого порога α_{min}, элементы перемещаются в хеш-таблицу меньшего размера.

Объединение хеш-значений

Предположим, что координаты точек на плоскости хранятся в виде пар целых чисел `(x,y)`, и нужно создать множество точек с использованием хеш-таблицы.

Пусть получены хеш-значения двух координат h(x) и h(y).

Как их объединить, чтобы получить хеш от пары?

Пусть для простоты верхняя граница возможных значений хеш-функции не фиксирована (где-то дальше в реализации хеш будет взят по нужному модулю M). Часто на практике программисты для соединения хешей пишут тривиальные функции, например через операцию побитового исключающего или (xor):

```
def combine(hx, hy):
return hx ^ hy
```

Такой вариант часто работает на практике приемлемо, но не лишён очевидных недостатков. Например, для всех точек с равными координатами x и y хеш-функция будет принимать нулевое значение, и если точек на прямой y = x во входных данных окажется много, производительность будет низкой из-за коллизий.

Также очевидно, что разные точки (x, y) и (y, x), симметричные относительно той же прямой, получат одинаковые хеш-значения. Чтобы подобрать пары, дающие коллизию, было труднее, для объединения хешей используют более сложные функции с обилием «магических» констант и странных операций.

Например, в C++-библиотеке boost используется примерно такая формула:

```
def combine(hx, hy):
 return hx ^ (hy + 0x9e3779b9 + (hx << 6) + (hx >> 2))
```

Часто берут линейную комбинацию двух хеш-значений с, например, большими взаимно простыми коэффициентами.

```
def combine(hx, hy):
 return hx + 1000000007 * hy
```

Основной смысл таких манипуляций — сделать так, чтобы на реально встречающихся в жизни данных коллизии были более редкими. Но контрпример при желании можно подобрать. Лучшего универсального решения в этом деле нет.

Хеширование строк.
Полиномиальное хеширование

Предположим, что на вход поступает строка: $S = s_0, s_1, s_2, \dots, s_n$.

Можно считать, что элементы строки – целые числа от **1** до L, где L – размер алфавита.

Для этого каждый символ строки S заменяем его порядковым номером в алфавите s-’a’+1.

S= ‘abcdb’ → 1,2,3,4,2

Рекомендуют выбирать следующие константы:

$k > L$ (натуральное число, чуть больше размера алфавита);

m – достаточно большое целое число, например 2^{64};

числа **k** и m должны быть взаимно просты, например, если m=2^{64}, то k – нечётное число.

Прямой полиномиальный хеш	**Обратный полиномиальный хеш**
это число $h_{\text{пр.}}(s) = (s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_n \cdot k^n) \bmod m$ (хеширование **слева направо**)	это число $h_{\text{обр.}}(s) = (s_0 \cdot k^n + s_1 \cdot k^{n-1} + s_2 \cdot k^{n-2} + \cdots + s_n) \bmod m$ (хеширование **справа налево**)
Предположим, что алфавит небольшой, $L = 9$. Пусть $k = 10 >$ и $m = 1\ 000\ 007$.	
$h_{\text{пр.}}$(‘abcdb’ → 1,2,3,4,2)=24 321	$h_{\text{обр.}}$(‘abcdb’ → 1,2,3,4,2)=12 342

Две одинаковые строки будут иметь обязательно один хеш, а вероятность коллизий крайне мала.

Основная операция, которую позволяют выполнять хеши – быстрое $(O(1))$ сравнение двух подстрок на равенство.

Вычислить значения функций
$\boldsymbol{h}_{\text{пр.}}(\boldsymbol{s})$ и $\boldsymbol{h}_{\text{обр.}}(\boldsymbol{s})$
можно за линейное от длины строки время ($= \Theta(n)$).

1) **Прямое полиномиальное хеширование**:

$$\boldsymbol{h_{\text{пр.}}(s) = (s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_n \cdot k^n) \bmod m.}$$

Выполним сначала за время $\Theta(n)$ предподсчёт нужных степеней.
Обозначим $\boldsymbol{k[i]} = k^i \bmod \boldsymbol{m}$, тогда:

$$\begin{cases} \boldsymbol{k[0] = 1} \\ \boldsymbol{k[i] = (k[i-1] \cdot k) \bmod m,\ i = 1, \dots, n.} \end{cases}$$

Обозначим $\boldsymbol{h_{\text{пр.}}(0{:}\,i)}$ - хеш для префикса строки $\boldsymbol{s[0..i]}$:

$$\boldsymbol{h_{\text{пр.}}(0{:}\,i) = (\underbrace{s_0 + s_1 \cdot k^1 + s_2 \cdot k^2 + \cdots + s_i \cdot k^i}) \bmod m.}$$

Тогда

$$\begin{cases} \boldsymbol{h_{\text{пр.}}(0{:}\,0) = s_0} \\ \boldsymbol{h_{\text{пр.}}(0{:}\,i) = \left(h_{\text{пр.}}(0{:}\,i-1) + s_i \cdot k[i]\right) \bmod m,\quad i = 1, \dots, n} \end{cases}$$

Так как значение хеша каждого следующего префикса выражается через значение хеша предыдущего префикса, то линейным проходом по всей строке за время $O(n)$ можно вычислить хеши для всех префиксов строки.

2) **Обратное полиномиальное хеширование**:

$$\boldsymbol{h_{\text{обр.}}(s) = (s_0 \cdot k^n + s_1 \cdot k^{n-1} + s_2 \cdot k^{n-2} + \cdots + s_n) \bmod m}$$

$$\begin{aligned} &\boldsymbol{h_{\text{обр.}} = 0} \\ &\boldsymbol{i = 0, \ldots, n} \\ &\qquad \boldsymbol{h_{\text{обр.}} = \left(h_{\text{обр.}} \cdot k + s_i\right) \bmod m,} \end{aligned}$$

0

$i = 0 \quad (0 \cdot k + s_0) \bmod m$

$i = 1 \quad (s_0 \cdot k + s_1) \bmod m$

$i = 2 \quad (s_0 \cdot k^2 + s_1 \cdot k^1 + s_2) \bmod m$

…

$i = n \quad (s_0 \cdot k^n + \ldots + s_{n-1} \cdot k^1 + s_n) \bmod m$

Так как хеш - это значение многочлена, то для многих строковых операций можно быстро пересчитывать хеш результата.

$$h_{\text{пр.}}(\mathrm{x}) = (\mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \mathrm{x}_2 \cdot k^2 + \cdots + \mathrm{x}_n \cdot k^n) \bmod m$$
$$h_{\text{пр.}}(y) = (y_0 + y_1 \cdot k^1 + y_2 \cdot k^2 + \cdots + y_r \cdot k^r) \bmod m$$

Конкатенация двух строк X и $\mathrm{Y} = \mathrm{x}_0, \ldots, \mathrm{x}_n, y_1, \ldots, y_r$.

$$h_{\text{пр.}}(\mathrm{x}y) = (\mathrm{x}_0 + \mathrm{x}_1 \cdot k^1 + \mathrm{x}_2 \cdot k^2 + \cdots + \mathrm{x}_n \cdot k^n + y_0 \cdot k^{n+1} + y_1 \cdot k^{n+2} + \cdots + y_r \cdot k^{n+r}) \bmod m$$

$$h_{\text{пр.}}(\mathrm{x}y) = (\underbrace{(\mathrm{x}_0 + \cdots + \mathrm{x}_n \cdot k^n) \bmod m}_{h_{\text{пр.}}(x)} + k^{n+1} \cdot \underbrace{(y_0 + \cdots + y_r \cdot k^r) \bmod m}_{h_{\text{пр.}}(y)}) \bmod m$$

$$\boxed{h_{\text{пр.}}(xy) = (h_{\text{пр.}}(x) + k^{|x|} \cdot h_{\text{пр.}}(y)) \bmod m}$$

Хеш для подстроки $x[l..r]$

$$h_{\text{пр.}}(l:r) = x_l + x_{l+1} \cdot \boldsymbol{k^1} + \cdots + x_r \cdot \boldsymbol{k^{r-l}}$$

как вычислить быстрее?

Предположим, что $m = 2^{64}$ и значение полинома помещается в 64 бита. Это равносильно тому, что на компьютере можно выполнять все операции над 64-битными числами и не задумываться о том, чтобы брать остаток. Но в C++ для этого необходимо использовать не `int64_t` (обычно, синоним `long long`), а `uint64_t` (`unsigned long long`), потому что переполнение знаковых целочисленных типов в C++ — опасный пример неопределённого поведения.

$$h_{\text{пр.}}(x) = \underbrace{x_0 + x_1 \cdot k^1 + \cdots + x_{l-1} \cdot k^{l-1}}_{} + x_l \cdot \boldsymbol{k^l} + x_{l+1} \cdot \boldsymbol{k^{l+1}} + \cdots + x_{r-1} \cdot \boldsymbol{k^{r-1}} + x_r \cdot \boldsymbol{k^r} + \cdots + x_n \cdot k^n$$

$$h_{\text{пр.}}(l:r) \cdot k^l = h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)$$

$$h_{\text{пр.}}(l:r) = \frac{h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)}{k^l}$$

Из формулы следует, что для вычисления хешей любой подстроки, необходимо знать лишь хеши всех префиксов этой строки. Теперь, чтобы сравнить на равенство две любые подстроки строки S, можно сравнить соответствующие хеши этих подстрок. Так как предполагается, что значение полинома помещается в 64 бита, то считаем, что сравнение двух хешей будет выполнено за $O(1)$.

$$h_{\text{пр.}}(l:r) = \left(\frac{h_{\text{пр.}}(0:r) - h_{\text{пр.}}(0:l-1)}{k^l}\right) \bmod m,$$

Для реализации математической формулы необходимо найти обратный элемент для $p := k^l$ по модулю m.
Для этого нужно вычислить значение функции Эйлера $\varphi(m)$ (= число взаимно простых с m чисел) и возвести p в степень $\varphi(m) - 1$ (https://ru.m.wikipedia.org/wiki/Обратное_по_модулю_число).
В случае полиномиального хеша с $m = 2^{64}$ обратным числу p будет число $p^{2^{63}-1}$.
Вычислить величину $p^{2^{63}-1}$ можно бинарным возведением в степень:

$$p^n = \begin{cases} (p^{n/2})^2, n - \text{чётное} \\ 2 \cdot (p^{n-1/2})^2, n - \text{нечётное} \end{cases}$$

Если предподсчитать степени обратного элемента по выбранному модулю m, то сравнение можно выполнять за O(1).

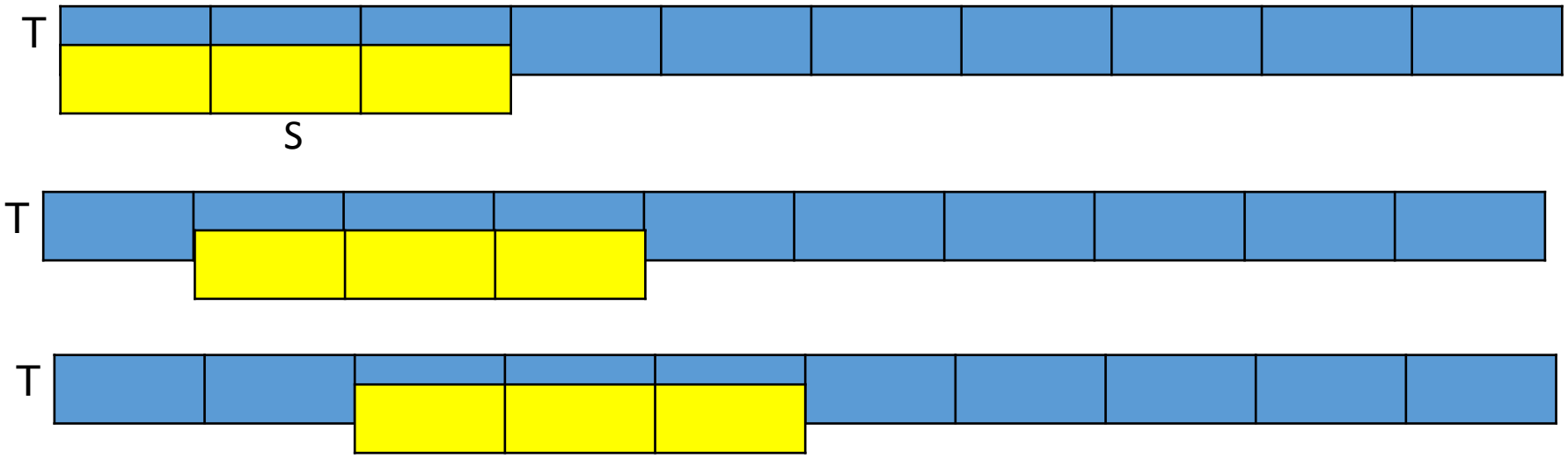
Так как деление достаточно сложная операция, то для сравнения подстрок на равенство на практике используют формулу без деления (приведение к одной степени). Предположим, что $l_2 > l_1$, тогда

$$h_{\text{пр.}}(l_1:r_1) = h_{\text{пр.}}(l_2:r_2)$$

$$k^{l_2-l_1} \cdot (h_{\text{пр.}}(0:r_1) - h_{\text{пр.}}(0:l_1-1)) = (h_{\text{пр.}}(0:r_2) - h_{\text{пр.}}(0:l_2-1))$$

Применение хеширования строк

- ✓ сравнение двух подстрок на равенство;
- ✓ подсчитаем хеши всех префиксов строки S, тогда можно за время $O(\log |S|)$ лексикографически сравнить две строки на больше (меньше): дихотомией ищут такой наибольший индекс i, что префиксы обеих строк длины i равны;
- ✓ за время $O(|T| + |S|)$ можно найти все вхождения подстроки S в текст T (алгоритм Рабина (Rabin)-Карпа (Karp), 1987 год): сначала для всех префиксов T вычислить хеши, также вычислить хеш для всей стоки S; затем двигаясь слева направо по тексту T пробуем накладывать окно длины $|S|$.



Проход по содержимому хеш-таблицы

В процессе программирования может возникнуть необходимость выполнить обход всех элементов структуры данных и, например, распечатать их.

Функция для итерации по содержимому структуры является полезной, поэтому обычно поддерживается в реализациях хеш-контейнеров, с которыми ведётся работа на практике.

В большинстве реализаций проход по хеш-множествам выполняется в произвольном порядке, **не гарантируется какой-либо отсортированности ключей**.

В случае, если внутренняя реализация хеш-таблицы использует метод цепочек, обычно функция обхода выдаёт сначала все элементы первой корзины (с хеш-значением 0) в порядке их следования в цепочке, затем все элементы второй корзины (с хеш-значением 1) и т. д.

Более того, если распечатать элементы хеш-множества, добавить новый ключ, сразу удалить его, вновь распечатать элементы, то порядок может получиться другим. Такое может случиться, если добавление нового ключа привело к перестроению хеш-таблицы с изменением числа M корзин и элементы были перераспределены по корзинам вновь. Не стоит нигде в коде закладываться на порядок итерации по хеш-контейнерам: большинство реализаций в разных языках программирования могут гарантировать только то, что посещены будут все элементы, не важно в каком порядке.

Наоборот, средства итерации по ключам множества, которое построено на базе бинарного поискового дерева, обычно возвращают ключи в порядке возрастания (выполняется внутренний обход дерева). Порядок фиксирован и каждый раз одинаковый.

Часто предсказуемость результата удобна, например, для написания модульных тестов к частям программы. **Таким образом, если порядок итерации важен, возможно, стоит использовать «древесные» структуры данных.**

Хеш-таблицы в C++

Долгое время в языке **C++** не было стандартных реализаций структур данных на основе хеш-таблиц.

Контейнеры **`std::set`** и **`std::map`** из **STL** строятся на основе сбалансированных бинарных поисковых деревьев (во всех популярных реализациях применяются красно-чёрные деревья).

Хеш-таблицы существовали в виде нестандартных расширений (например `stdext::hash_set` в Visual Studio) или внешних библиотек (например `boost`).

Наконец, **в стандарте C++11 в STL официально были добавлены хеш-таблицы**.

Стандарт предусматривает четыре контейнера на основе хеш-таблиц, которые отличаются от своих аналогов на основе деревьев наличием префикса **unordered_** в названии.

std::unordered_set представляет собой динамическое множество;
std::unordered_map — ассоциативный массив.

Существует также два **multi-контейнера**, которые допускают хранение одинаковых ключей.

Стандарт требует, чтобы в построении этих структур данных авторы компиляторов использовали **разрешение коллизий методом цепочек**.

Метод открытой адресации не был стандартизирован из-за внутренних трудностей при удалении элементов. Однако детали реализации хеш-таблиц стандартом не регламентируются.

В качестве хеш-значения в C++ используется число типа **`size_t`**.

Все хеш-контейнеры предоставляют метод **`rehash()`**, который позволяет установить размер хеш-таблицы (число корзин **`M`**).

Метод под названием **`load_factor()`** возвращает текущий коэффициент заполнения.

Рассмотрим более подробно реализацию в компиляторе GCC (GNU Compiler Collection)

Пусть ключи добавляются в **`std::unordered_set`** по одному.

Когда коэффициент заполнения достигает значения 1, происходит перестроение хеш-таблицы:

в качестве нового числа корзин берётся первое простое число из заранее составленного списка, не меньшее удвоенного старого числа корзин (**таким образом, размер таблицы как минимум удваивается и является простым числом**).

Длины отдельных цепочек никак не анализируются (появление одной длинной цепочки не повлечёт за собой операцию перестроения).

Хеш-таблицы в JAVA

Коллекции **HashSet** и **HashMap** реализуются как хеш-таблицы, для разрешения коллизий используется **метод цепочек**.

Для хеширования целых чисел применяется функция следующего вида:

```
int hash(int h) {
h ^= (h >>> 20) ^ (h >>> 12);
return h ^ (h >>> 7) ^ (h >>> 4);
}
```

операция **>>>** — беззнаковый сдвиг вправо: биты смещаются вправо, число слева дополняется нулями;

операция **^** — поразрядное сложение по модулю 2, исключающее «или»;

Затем в классе коллекции результат функции hash берётся по модулю числа корзин, по которым раскладываются элементы.

Число корзин, оно же число различных значений хеш-функции M, в Java **всегда выбирается как некоторая степень числа 2:**

чтобы деление на M можно было заменить операцией битового сдвига вправо, так как современные процессоры выполняют инструкцию деления целых чисел существенно медленнее, чем битовые операции;

В версии **Java 8**
разработчики озаботились вопросом **устойчивости коллекций**, использующих хеширование, **к коллизиям**.

В исходном коде библиотеки Java можно найти константу:

```
istatic final int TREEIFY_THRESHOLD = 8;
```

В случае, если новый ключ попадает в корзину, в которой уже лежат как минимум восемь других ключей, библиотека **преобразует связный список** для данной корзины в **бинарное сбалансированное поисковое дерево**.

Получается гибридная структура:

- ✔ корзины для тех хеш-значений, где **ключей мало**, хранятся **списками**;
- ✔ корзины, где **ключей** накопилось **много**, хранятся в виде **деревьев**.

Хеш-таблицы в PYTHON

Встроенный тип **dict** — ассоциативный массив, словарь —широко используется в языке.

Он реализован в виде хеш-таблицы, где коллизии разрешаются **методом открытой адресации.**

> Разработчики предпочли метод открытой адресации методу цепочек ввиду того, что он позволяет значительно сэкономить память на хранении указателей, которые используются в хеш-таблицах с цепочками.

Интерпреттором CPython поддерживается опция командной строки -R, которая активирует на старте случайный выбор начального значения (англ. seed), которое затем используется для вычисления хеш-значений от строк и массивов байт.

Криптографические хеш-функции

В криптографии множество возможных ключей бесконечно, и любой блок данных является ключом (в принципе, произвольный массив байт можно рассматривать как двоичную запись некоторого числа).

Хеш-функция h(x) называется **криптографической**, если она удовлетворяет следующим требованиям:

- **необратимость**: для заданного значения хеш-функции **c** должно быть сложно определить такой ключ **x**, для которого **h(x) = c**;
- **стойкость к коллизиям первого рода**: для заданного ключа **x** должно быть вычислительно невозможно подобрать другой ключ **y**, для которого **h(x) = h(y)**;
- **стойкость к коллизиям второго рода**: должно быть вычислительно невозможно подобрать пару ключей **x** и **y**, имеющих одинаковый хеш.

Криптографические хеш-функции обычно не используются в хеш-таблицах, потому что они сравнительно медленно вычисляются и имеют большое множество значений. Зато такие хеш-функции широко применяются в системах контроля версий, системах электронной подписи, во многих системах передачи данных для контроля целостности.

Примерами криптографических хеш-функций являются алгоритмы MD5, SHA-1, SHA-256.

Так, метод SHA-1 ставит в соответствие произвольному входному сообщению некоторую 20-байтную величину, т. е. результат вычисления SHA-1 принимает одно из 2^{160} различных значений.

Пример вычисления SHA-1 от ASCII-строки, где результат записан в шестнадцатеричной системе счисления:

SHA-1("The quick brown fox jumps over the lazy dog") = 0x2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

В настоящий момент коллизии для MD5 и SHA-1 обнаружены, поэтому методы постепенно выходят из широкого использования. Более новые алгоритмы семейства SHA-2 считаются существенно более стойкими к коллизиям. Тем не менее следует понимать, что коллизии есть обязательно, потому что нельзя биективно отобразить бесконечное множество в конечное. Вопрос только в том, насколько трудно эти коллизии отыскать.

За годы развития вычислительной техники сформировалась наука о том, как практически строить хеш-функции.

> Разработаны всевозможные правила на тему того, какие биты ключа x стоит взять, на сколько позиций выполнить поразрядный сдвиг, как применить операцию исключающего «или», на что умножить, чтобы получить хеш, который бы выглядел как случайный.

Чтобы найти примеры практических хеш-функций, можно взять какую-нибудь стандартную библиотеку. Эти функции часто ничего не гарантируют: коллизии, конечно, есть.

Любую фиксированную хеш-функцию можно всегда «сломать» — придумать ситуацию, когда там будут одни сплошные коллизии.

Попробуем тогда внести элемент случайности при выборе хеш-функции?

Универсальное хеширование

(англ. universal hashing)

Добавить рандомизацию:

внести элемент случайности, чтобы не было фиксированного «плохого» случая.

Идея не брать какую-либо одну конкретную хеш-функцию, а, например, организовать программу так, чтобы при каждом запуске хеш-функция выбиралась заново.

Тогда одним и тем же входом «сломать» программу не получится, и можно будет рассуждать о длине цепочки с точки зрения теории вероятностей: говорить о том, какая длина цепочки в среднем, какая у неё дисперсия и т.п.

Такой подход называют **универсальным хешированием.**

Хеш-функция выбирается

> случайным образом из некоторого сгенерированного специальным образом **универсального множества хеш-функций**, которое гарантирует, что при случайном выборе хеш-функции из него вероятность коллизии для двух различных элементов $< \frac{1}{M}$

В **1977** году Дж.Л. **Картером** (J.L. Carter) и М.Н. **Вегманом** (M.N. Wegman) предложен один из способ построения универсального семейства хеш-функций для целочисленных ключей.

$$H_{a,b}(x) = \big((a \cdot x + b) \bmod p\big) \bmod M$$

x – ключ;

M – размер таблицы (произвольное число, не обязательно простое);

p – выбрать достаточно большое простое число, такое, что все ключи лежат в диапазоне от 0 до $p - 1$ ($p > M$);

a – выбирается из множества $\{1,2,\ldots, p-1\}$
b – выбирается из множества $\{0,1,\ldots, p-1\}$

постулат Бертрана гласит, что существует достаточно близкое к N простое число: $N \le p < 2N$

$p \cdot (p-1)$ способ выбрать хеш-функцию из семейства

Оценки

добавление элемента – усреднённо O(1);

поиск элемента – математичекое ожидание O(1).

Совершенное хеширование

(англ. *perfect hashing*)

Пусть **S** — фактическое множество ключей в хеш-таблице, и у этого множества размер **n**, где **n<M**.

Тогда существует **хеш-функция**, которая не даёт коллизий: если элементов меньше, чем слотов, то их всегда можно отобразить без коллизий. Это хорошая хеш-функция, её называют **совершенной**.

Эта функция всегда есть и зависит от набора ключей.

Даже если предположить, что набор ключей статический и известен заранее (т.е. все ключи сразу поступили, размещены в таблице и новые ключи туда более не будут более добавлены), есть трудность в том, как эффективно задать эту хеш-функцию.

Таким образом, подход заключается в построении **хеш-функции**, которая является:

- **простой**, т.е. достаточно константного объёма памяти, чтобы её хранить;
- **быстрой**, т.е. требуется константное время на вычисление;
- **совершенной,** т.е. коллизии отсутствуют.

Совершенное двухуровневое хеширование

Уровень 1

хеширование цепочками в M ячеек, возможны коллизии

выполнять

✔ выбрать хеш-функцию для таблицы порядка M

(функция берётся из универсального множества хеш-функций); $S = \sum_{0 \le i \le M-1} L_i^2$

✔ вычислить $P(S \le 3 \cdot M) \ge 1/2$

Уровень 2

повторять (S>3·M)

для каждой цепочки строится небольшая вторичная хеш-таблица, чтобы гарантировать отсутствие коллизий на этом уровне

выполнять

выбрать хеш-функцию для таблицы порядка $(L_i)^2$

(из универсального множества хеш-функций);

пока (есть коллизии)

$$P(\text{хеш}-\text{функция для таблицы порядка } k^2 \text{ на } k \text{ элементах не даёт коллизий}) \ge \frac{1}{2}$$

<u>В худшем случае поиск</u> элемента выполняется за **O(1)**

<u>Математическое ожидание построения таблицы</u> **O(n)**

$$H_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod M$$

x – ключ;

M – размер таблицы (произвольное число, не обязательно простое);

p - достаточно большое простое число, такое, что все ключи лежат в диапазоне от *0* до *p-1 (p>M);*

a - выбирается из множества {1,2,…,p-1}

b - выбирается из множества {0,1,…,p-1}

K={10,22,37,40,60, 70,75}

Уровень 1.

$h_{3,42}(x) = ((3 \cdot x + 42) \bmod 101) \bmod 9$

0	**10**
1	
2	**60,75**
3	
4	
5	**70**
6	
7	**37,40,22**
8	
9	

$S=1^2+2^2+1^2+3^2=15$

$15<3\cdot9$,

переход ко 2-у уровню

Уровень 2.

$h_i(x) = ((a_i \cdot x + b_i) \bmod 101) \bmod m_i$

0	m_0=1	a_0=0	b_0=0	10							
1											
2	m_2=4	a_2=10	b_2=18	60	75						
3											
4											
5	m_5=1	a_5=0	b_5=0	70							
6											
7	m_7=9	a_7=23	b_7=88	40			37				22
8											
9											

ФПМИ БГУ

Общие задачи в iRunner для закрепления навыков

0.5. Хеш-таблица (разрешение коллизий метом открытой адресации)

ФПМИ БГУ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Спасибо за внимание