УДК 004.023+004.622 DOI: 10.31114/2078-7707-2022-4-26-31

Алгоритм построения быстрых хеш-функций, основанных на замещении символов

М.И. Дябин[1], А.В. Решетников[2]

[1]ООО «Каскад», г. Москва, dyabin@mail.ru

[2]ООО «Аккорд», г. Москва, a_reshetnikov@hush.com

***Аннотация* — Рассматриваются хеш-функции, основанные на замещении символов алфавита целыми неотрицательными числами. Основным параметром таких функций является отображение *T*, осуществляющее замещение; основное назначение функций – выполнение статического хеширования; главные преимущества – высокая скорость работы и простота их реализации. Предлагается алгоритм, выполняющий оптимизацию параметра *T* с целью уменьшения количества коллизий для заданного словаря. Приводятся примеры эффективных хеш-функций, основанных на замещении символов.**

***Ключевые слова* — быстрая хэш-функция, статическое хеширование, замещение символов алфавита.**

I. Введение

Понятие хеш-функции [1, раздел 6.4] хорошо известно в теории программирования. Область их применения разнообразна, в том числе хеш-функции используются для оптимизации программ, которым в процессе работы требуется произвести синтаксический анализ какого-либо длинного текста: в тех ситуациях, когда обрабатываемый текст может содержать часто повторяющиеся слова, для них целесообразно вычислить значения хешей и занести все эти значения в специальную таблицу. Действительно, поиск слова в словаре может быть существенно ускорен, если удачно подобранная хеш-функция позволяет просматривать не весь словарь целиком, а лишь ту его часть, которая содержит список всех слов с фиксированным значением хеша – общего для них и для того слова, которое требуется найти.

При разработке компиляторов, интерпретаторов, верификаторов и других программных систем – таких, которые предназначены для обработки текстовых данных, использующих некий формальный язык, содержащий ограниченный набор ключевых слов – возникает задача построения хеш-функции таким образом, чтобы с её помощью как можно эффективнее производилось распознавание ключевых слов заданного языка. Особенностью словаря в данном случае является то, что он содержит, как правило, небольшое количество слов, и набор этих слов фиксирован и известен уже на этапе разработке программы.

Задачи такого рода называются задачами *статического хеширования*. Хорошо известен один из методов их решения, представленный в статье [2].

В данной работе для задач статического хеширования мы рассматриваем новый метод решения: предлагается использовать для этой цели хеш-функции, основанные на *замещении символов* в словах некоторыми числами. Основным преимуществом нашего метода является простота его реализации; мы приведём фрагменты кода, реализующие некоторые из рассматриваемых хеш-функций.

При использовании замещения символов в простейшем случае предлагается каждому символу x из алфавита заданного языка поставить в соответствие то или иное число $T(x)$; тогда хеширование произвольной строки s может быть произведено следующим образом: каждый элемент $x \in s$ сначала замещается соответствующим ему числом $T(x)$, а затем над полученными числами производится какая-либо быстрая операция f. Например, если в качестве f мы возьмём сложение по модулю 2^M, то тем самым полностью определим функцию, которая в данной работе будет обозначаться через h_0 (более строгое её определение будет приведено далее).

Ясно, что скорость вычисления значения $h_0(s)$ весьма высока: для получения этого значения требуется лишь извлечь из памяти $|s|$ целых чисел ($|s|$ – длина строки s), выполнить их арифметическое сложение и, если это необходимо, взять остаток от деления получившейся суммы на 2^M. (Число M – это не что иное, как количество значащих битов в вычисленном хеше.) Однако, в случае неудачно подобранного соответствия T – оно является параметром функции h_0 – общая скорость работы системы может проседать из-за возникновения *хеш-коллизий* – то есть совпадений значений $h_0(s)$ и $h_0(t)$ для двух различных строк s и t, взятых из словаря заданного языка. В связи с этим возникает важный вопрос:

Каким образом можно подобрать параметр T так, чтобы для заданного словаря снизить число хеш-коллизий до приемлемого уровня?

Настоящая работа предлагает один из возможных ответов на этот вопрос. Мы рассмотрим алгоритм, способный в случае фиксированного словаря проводить

оптимизацию параметра T не только для функции h_0, но и для более сложных хеш-функций. Эффективность данного алгоритма мы продемонстрируем на конкретном примере. Подчеркнём, что предлагаемый алгоритм не обязательно снижает число коллизий до теоретически возможного минимума.

Другой способ уменьшения числа коллизий состоит в том, чтобы перейти от функции h_0 к какой-либо более сложной хеш-функции. В самом деле: если в строке s поменять местами какие-либо два символа – обозначим новую строку через s', то $h_0(s) = h_0(s')$ – то есть, функция h_0 не способна обнаруживать в строках различия такого рода. Мы предложим ещё несколько хеш-функций – а именно, h_{SHR}, h_{ROL}^{xor} и несколько других – обладающих почти такой же скоростью вычисления, как и h_0, но способных обеспечивать гораздо меньшее число хеш-коллизий.

II. Хеш-функция h_0

Начнём с того, что приведём математическое определение функции h_0, о которой говорилось во введении.

Пусть X – алфавит некоторого формального языка, то есть произвольное множество, элементы которого мы будем называть *символами* данного алфавита. *Строкой* длины n называется любая последовательность символов алфавита, содержащая ровно n элементов. *Пустая строка* – это строка длины 0. Множество всех строк конечной длины, составленных из символов алфавита X, будем обозначать через X^*; пустая строка также принадлежит множеству X^*. На практике, конечно же, X всегда является конечным множеством.

Напомним, что задача статического хеширования словаря $L \subseteq X^*$ состоит в том, чтобы построить какое-либо отображение h множества X^* в множество целых чисел, удовлетворяющее следующим условиям:

1) для любой строки $s \in X^*$ вычисление значения $h(s)$ должно производиться достаточно быстро;
2) желательно, чтобы разным словарным строкам $s \in L$ соответствовали разные значения $h(s)$.

Отображение h при этом называется *хеш-функцией*, а число $h(s)$ – *хешем* строки s.

Зафиксируем целое положительное число M и какое-либо отображение T множества X в множество целых неотрицательных чисел; таким образом, каждому элементу $x \in X$ будет поставлено в соответствие определённое число $T(x)$. Пусть $s = x_1 x_2 ... x_n$ – строка, для которой требуется найти значение $h_0(s)$. По определению положим

$$h_0(s) = \left(\sum_{i=1}^{n} T(x_i) \right) \bmod 2^M \qquad (1)$$

– остаток от деления суммы на число 2^M.

Из соотношения (1) видно, что функция h_0 не чувствительна к порядку следования символов в строке s. Коллизии неизбежны при использовании h_0, если в словаре есть строки, отличающиеся лишь порядком следования символов. Тем не менее, даже такая, казалось бы, примитивная хеш-функция, если её параметры M и T выбраны надлежащим образом, способна обеспечить весьма эффективное хеширование. Продемонстрируем это на примере.

Рассмотрим словарь, состоящий из списка команд процессора Intel 8086. Данный словарь включает в себя 116 строк, которые представлены в табл. 1 слева от стрелок. Справа от стрелок представлены их 8-битные хеши (т.е. $M = 8$), вычисленные с помощью функции h_0; в процессе вычисления следующее отображение использовано в качестве параметра T:

A 0x01	B 0xBF	C 0x9D	D 0xE6	E 0x5D	F 0x90	G 0x13
H 0xE7	I 0x31	J 0xF3	K 0x4A	L 0x19	M 0x34	N 0x8E
O 0xFA	P 0x93	Q 0xF0	R 0xF1	S 0xD6	T 0xB1	U 0xF4
	V 0x00	W 0x15	X 0x38	Y 0x18	Z 0x27	

(В нижних строчках ячеек указаны числа, которые отображение T ставит в соответствие буквам, указанным в верхних строчках.)

Из табл. 1 видно, что все строки s, за исключением «AAD» и «DAA», имеют различные значения $h_0(s)$. Единственная коллизия между «AAD» и «DAA» неизбежна, поскольку эти строки совпадают друг с другом с точностью до перестановки символов.

III. Хеш-функции h_F и h_F^{xor}

Перейдём к рассмотрению функций, способных (потенциально) обеспечить более эффективное хеширование, чем функция h_0. Их будет проще задать, если предварительно переписать определение для h_0 с использованием рекуррентных соотношений.

Выберем какие-либо параметры M и T функции h_0 и рассмотрим произвольную строку $s = x_1 x_2 ... x_n$ над алфавитом X. Положим

$$y_0 = 0; \quad y_i = y_{i-1} + T(x_i) \qquad (2)$$

для всех $i \le n$. Очевидно, что

$$h_0(s) = y_n \bmod 2^M. \qquad (3)$$

Таблица 1

Список команд процессора Intel 8086 (слева от стрелок) и их хеши (справа от стрелок), вычисленные с помощью функции h_0. Единственная коллизия наблюдается между строками «AAD» и «DAA», которые принципиально не различимы для данной хеш-функции

AAA → 0x03	JA → 0xF4	JPO → 0x80	RCL → 0xA7
AAD → 0xE8	JAE → 0x51	JS → 0xC9	RCR → 0x7F
AAM → 0x36	JB → 0xB2	JZ → 0x1A	REP → 0xE1
AAS → 0xD8	JBE → 0x0F	LAHF → 0x91	REPE → 0x3E
ADC → 0x84	JC → 0x90	LDS → 0xD5	REPNE → 0xCC
ADD → 0xCD	JCXZ → 0xEF	LEA → 0x77	REPNZ → 0x96
AND → 0x75	JE → 0x50	LES → 0x4C	REPZ → 0x08
CALL → 0xD0	JG → 0x06	LODSB → 0x8E	RET → 0xFF
CBW → 0x71	JGE → 0x63	LODSW → 0xE4	RETF → 0x8F
CLC → 0x53	JL → 0x0C	LOOP → 0xA0	ROL → 0x04
CLD → 0x9C	JLE → 0x69	LOOPE → 0xFD	ROR → 0xDC
CLI → 0xE7	JMP → 0xBA	LOOPNE → 0x8B	SAHF → 0x4E
CMC → 0x6E	JNA → 0x82	LOOPNZ → 0x55	SAL → 0xF0
CMP → 0x64	JNAE → 0xDF	LOOPZ → 0xC7	SAR → 0xC8
CMPSB → 0xF9	JNB → 0x40	MOV → 0x2E	SBB → 0x54
CMPSW → 0x4F	JNBE → 0x9D	MOVSB → 0xC3	SCASB → 0x09
CWD → 0x98	JNC → 0x1E	MOVSW → 0x19	SCASW → 0x5F
DAA → 0xE8	JNE → 0xDE	MUL → 0x41	SHL → 0xD6
DAS → 0xBD	JNG → 0x94	NEG → 0xFE	SHR → 0xAE
DEC → 0xE0	JNGE → 0xF1	NOP → 0x1B	STC → 0x24
DIV → 0x17	JNL → 0x9A	NOT → 0x39	STD → 0x6D
HLT → 0xB1	JNLE → 0xF7	OR → 0xEB	STI → 0xB8
IDIV → 0x48	JNO → 0x7B	OUT → 0x9F	STOSB → 0x16
IMUL → 0x72	JNP → 0x14	POP → 0x20	STOSW → 0x6C
IN → 0xBF	JNS → 0x57	POPA → 0x21	SUB → 0x89
INC → 0x5C	JNZ → 0xA8	POPF → 0xB0	TEST → 0x95
INT → 0x70	JO → 0xED	PUSH → 0x44	XCHG → 0xCF
INTO → 0x6A	JP → 0x86	PUSHA → 0x45	XLATB → 0xC2
IRET → 0x30	JPE → 0xE3	PUSHF → 0xD4	XOR → 0x23

Равенство (3) можно рассматривать как ещё одно определение функции h_0. Согласно данному определению, для того чтобы вычислить $h_0(x_1x_2...x_n)$, можно сначала последовательно вычислить y_1, y_2, ..., y_n по формулам (2), а затем в качестве $h_0(s)$ взять остаток от деления числа y_n на число 2^M.

Далее перейдём от (2) к формулам более общего вида. Наряду с отображением T выберем на множестве целых неотрицательных чисел какое-либо преобразование F данного множества. Введём следующее рекуррентное соотношение:

$$y_0 = 0; \quad y_i = F(y_{i-1}) + T(x_i), \qquad (4)$$

где, как и раньше, через x_i обозначен i-ый символ строки $s = x_1x_2...x_n$, для которой требуется найти значение её хеша. Считаем по определению

$$h_F(x_1x_2...x_n) = y_n \bmod 2^M. \qquad (5)$$

Равенство (5) определяет функцию h_F, параметрами которой служат два отображения – F и T, а также целое положительное число M. Мы говорим, что h_F *основана на замещении символов,* поскольку её основным параметром является отображение T, осуществляющее замещение символов алфавита X.

В рамках данной работы предполагается, что F – фиксированное преобразование (этим объясняется выбор обозначения «h_F»), а поиск параметра T осуществляется с помощью алгоритма, который будет рассмотрен далее. Конкретные примеры преобразований F, при которых функция h_F является достаточно эффективной, будут приведены в следующем разделе.

Если в формуле (4) операцию арифметического сложения заменить операцией побитового исключающего *или* – то есть ввести новое рекуррентное соотношение

$$y_0^{xor} = 0; \quad y_i^{xor} = F\left(y_{i-1}^{xor}\right) \oplus T\left(x_i\right), \tag{6}$$

то аналогичным образом можно определить функцию

$$h_F^{xor}\left(x_1 x_2 ... x_n\right) = y_n^{xor}, \tag{7}$$

которая также основана на замещении символов. В формуле (6) символ «$\oplus$» обозначает операцию поразрядного сложения двух целых чисел по модулю 2. Деление с остатком в данном случае не требуется.

Предостережение 1. Может показаться, что определение (7) игнорирует длину хеша M; но на самом деле это, конечно, не так: параметр M учитывается при выборе отображения F как в функции h_F, так и в функции h_F^{xor}. Действительно: если, например, F – логический сдвиг влево, то M определяет номер разряда, начиная с которого двоичные знаки числа должны игнорироваться после выполнения его сдвига. Важно помнить об этом, подготавливая реализацию какой-либо из функций h_F или h_F^{xor}: на практике должны иметь место равенства

$$y_i = \left(F\left(y_{i-1}\right) \bmod 2^M\right) + T\left(x_i\right);$$

$$y_i^{xor} = \left(F\left(y_{i-1}^{xor}\right) \bmod 2^M\right) \oplus T\left(x_i\right).$$

IV. Хеш-функции h_{SHR}, h_{ROL}^{xor} и h_{ROR}^{xor}

Приведём примеры преобразований F, для которых функции h_F и h_F^{xor} способны эффективно хешировать словари с заранее известными списками строк.

Прежде всего, рассмотрим функцию h_{SHR}, где в качестве F выбрана операция *SHR* – сдвиг числа вправо на 1 бит, или, что то же самое, целочисленное деление числа на 2. Реализация такой функции на языке программирования C может выглядеть следующим образом:

```
unsigned h=0;
for (size_t i=0; s[i]!=0; i++)
{     unsigned c = (unsigned)s[i];
      h = (h >> 1) + table[c];
}
h &= (1u << M) – 1;
```

Здесь переменная *s* представляет собой строку, для которой требуется вычислить хеш h_{SHR}; переменная *table* – это массив, в котором каждый элемент с индексом *x* содержит беззнаковое число $T(x)$. Длина массива *table* по меньшей мере на 1 больше, чем самое большое беззнаковое представление символа, встречающегося в строке *s*. Переменная *M* содержит параметр M функции h_{SHR} – количество битов в вычисляемом хеше; для упрощения реализации сделано предположение, что значение *M* строго меньше[1], чем количество битов, содержащихся в типе *unsigned int*. После выполнения данного кода значение $h_{SHR}(s)$ будет записано в переменную *h*.

Функция h_{SHR} выполняет хеширования словаря из табл. 1 без коллизий при $M = 8$ в случае, когда T – отображение, заданное следующей таблицей:

A	B	C	D	E	F	G
0x4A	0x74	0xBE	0x56	0x18	0x00	0x79
H	I	J	K	L	M	N
0x00	0x71	0x00	0x00	0x20	0xB8	0xA6
O	P	Q	R	S	T	U
0x00	0x38	0x00	0x04	0x9F	0x00	0x4E
	V	W	X	Y	Z	
	0x00	0x00	0x19	0x00	0x78	

Представляет интерес функция h_{ROL}^{xor}: в качестве F она использует операцию *циклического* сдвига числа на 1 бит влево, рассматривая при этом сдвигаемое число как M-битное. Данная функция также хеширует словарь из табл. 1 без коллизий при $M = 8$, если в качестве T выбрано отображение

A	B	C	D	E	F	G
0x54	0xFF	0x3B	0x80	0x61	0x06	0x92
H	I	J	K	L	M	N
0xDE	0x7F	0x39	0xC5	0x15	0x0A	0xB3
O	P	Q	R	S	T	U
0xD7	0xDB	0xDC	0x23	0x71	0x6F	0x0C
	V	W	X	Y	Z	
	0xE4	0x63	0x65	0x16	0xFD	

Реализация функции h_{ROL}^{xor} на языке C может (при тех же допущениях) иметь следующий вид:

[1] В противном случае вместо «1u» требовалось бы писать «1ul».

```
unsigned h=0;
for (size_t i=0; s[i]!=0; i++)
{   unsigned c = (unsigned)s[i];
    h = ((h<<1) | (h>>(M–1))) & ((1u<<M)–1);
    h ^= table[c];
}
```

Двойственным образом определяется и реализуется функция h_{ROR}^{xor}, использующая в качестве F циклический сдвиг вправо ROR на 1 бит.

С математической точки зрения функции h_{ROL}^{xor} и h_{ROR}^{xor} интересны тем, что их значения могут быть вычислены без использования рекуррентных соотношений следующим образом:

$$h_{ROL}^{xor} = \bigoplus_{i=1}^{n} \left[T(x_i) \text{ ROL } (n-i) \right]; \qquad h_{ROR}^{xor} = \bigoplus_{i=1}^{n} \left[T(x_i) \text{ ROR } (n-i) \right], \tag{8}$$

где через $x \text{ ROL } a$ и $x \text{ ROR } a$ обозначены результаты циклических сдвигов M-битного числа x влево и вправо, соответственно, на a битов.

Эксперименты показывают, что h_{ROL} и h_{ROR} – тоже весьма эффективные хеш-функции, однако, *для них, вообще говоря, не выполняются соотношения, аналогичные (8)*.

Предостережение 2. Возьмём в качестве F операцию SHL, выполняющую для M-битового числа его логический сдвиг влево на 1 бит. Функция h_{SHL} обладает тем недостатком перед функцией h_{SHR}, что всякий раз после выполнения сдвига влево какого-либо числа y полученное значение $y << 1$ необходимо подвергать взятию от него остатка от деления на 2^M: в противном случае появление лишних битов в числе $y << 1$ может привести к результату, отличному от $h_{SHL}(s)$, а также нарушить кроссплатформенность реализации функции.

Предостережение 3. Согласно экспериментам, хеш-функции h_{SHR}^{xor} и h_{SHL}^{xor} не столь эффективны, как хеш-функции h_{SHR} и h_{SHL}.

V. Алгоритм оптимизации параметра T

Пусть $T: X \to Y$ – любое отображение. Введём следующее обозначение: если $\alpha \in X$ и $\beta \in Y$ – произвольные элементы, то обозначим через $T_{\alpha \mapsto \beta}$ отображение такое, что $T_{\alpha \mapsto \beta}(x) = T(x)$ для всех $x \in X \setminus \{\alpha\}$ и $T_{\alpha \mapsto \beta}(\alpha) = \beta$. Другими словами, если T задано некоторой таблицей *table*, то после выполнения команды $table[\alpha] := \beta$ та же таблица будет задавать отображение $T_{\alpha \mapsto \beta}$.

Опыт показывает, что для функций h_F и h_F^{xor} параметр T следует подбирать итерационно следующим образом.

Шаг 1. Изначально в качестве T можно взять любое отображение (например, проще всего положить $T(x) = 0$ для всех x).

Шаг 2. Если текущее значение T позволяет избежать коллизий (либо если при заданных начальных условиях известно минимальное теоретически возможное число коллизий и это число достигается при текущем значении параметра T), то поиск оптимального значения T завершён.

Шаг 3. Если среди всех отображений вида $T_{\alpha \mapsto \beta}$ есть такое отображение $T_{\alpha_0 \mapsto \beta_0}$, при котором число коллизий уменьшается, следует положить $T := T_{\alpha_0 \mapsto \beta_0}$ и перейти к шагу 2.

Шаг 4. Иначе следует провести рандомизацию параметра T и перейти к шагу 2.

Все таблицы для T, представленные в данной работе, были получены с использованием данного алгоритма. При поиске параметра T для функции h_{SHR} понадобилось всего лишь 16 итераций, при этом ни разу не был выполнен шаг 4 (изначально было выбрано $T(x) = 0$ при всех x).

Шаг 4 требует отдельного внимания. Рандомизация таблицы означает генерацию псевдослучайного вектора достаточно большой размерности. Желательно при этом, чтобы алгоритм рандомизации был детерминирован: при одних и тех же начальных условиях желательно, чтобы в конце концов алгоритм всегда находил одно и то же отображение T. Такой детерминированный алгоритм генерации псевдослучайных векторов представлен в работе [3]. В той же статье подробно обсуждаются общие способы решения данной проблемы.

VI. Выводы

Были рассмотрены функции h_F и h_F^{xor} при различных значениях параметра F. На примере словаря, состоящего из команд микропроцессора Intel 8086, было показано, что h_{SHR} и h_{ROL}^{xor} при надлежащем выборе значения T способны провести статическое хеширование данного словаря без коллизий, а функция h_0 способна снизить число коллизий до единственной коллизии, которую в данном случае невозможно исключить принципиально. Также был предложен алгоритм, позволяющий находить T в случае произвольного словаря.

Отображение T осуществляет замену символов алфавита целыми неотрицательными числами. Исполь-

зование отображений такого вида с целью построения хеш-функций ранее было предложено П.К.Пирсоном в работе [4]. Введённая им функция при попытке хешировать с её помощью словарь из табл. 1 всякий раз приводила к возникновению нескольких десятков коллизий, даже после оптимизации параметра T по описанному выше алгоритму.

Таким образом, если размер словаря не слишком велик, то, скорее всего, предложенный в данной работе алгоритм сможет за приемлемое время подобрать для него параметр T, при котором хеширование данного словаря с помощью функций h_0, h_{SHR}, h_{ROL}^{xor} или h_{ROR}^{xor} оказывается весьма эффективным. Авторы не исключают того, что одновременный поиск параметров F и T для функций h_F и h_F^{xor} в принципе мог бы обеспечить ещё более высокое качество статического хеширования. Но для выполнения такого поиска авторам алгоритм неизвестен.

ЛИТЕРАТУРА

[1] Knuth D.E. The Art of Computer Programming. V. 3. Searching and Sorting. Second Edition. Reading, Massachusetts: Addison – Wesley, 1973. xiv + 782 pp. (Кнут Д.Э. Искусство программирования. Том 3. Сортировка и поиск. 2-е изд. Пер. с англ. М.: И. Д. «Вильямс», 2007. 824 с.)

[2] Fredman M.L., Komlós J., Szemerédi E. Storing a Sparse Table with O(1) Worst Case Access Time // Journal of the ACM. 1984. V. 31. № 3. P. 538–544.

[3] Дябин М.И., Решетников А.В., Саксонов Е.А. Быстрый метод генерации псевдослучайных векторов большой размерности для тестирования систем на кристалле // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2021. № 4.С. 86–91.

[4] Pearson P.K. Fast Hashing of Variable-Length Text Strings // Communications of the ACM. 1990. V. 33. № 6. P. 677–680.

An Algorithm of Building Fast Hash Functions Based on Replacement of Symbols

M.I. Dyabin[1], A.V. Reshetnikov[2]

[1]"Kaskad" Ltd., Moscow, dyabin@mail.ru

[2]"Accord" Ltd., Moscow, a_reshetnikov@hush.com

***Abstract* — Let there be given a formal language X. Let T be a mapping which replaces the symbols of X by non-negative integer numbers. For any sequence $s = x_1x_2\ldots x_n$ of elements of X, suppose that $y_0 = 0$, $y_i = (y_{i-1}$ div $2) + T(x_i)$ for all i. Let us define $h_{SHR}(s) = (y_n \bmod 2^M)$, where M is a positive integer number, say $M = 8$. So h_{SHR} is an example of a function based on replacement of alphabetical symbols. M and T are the parameters of the function. If both parameters are chosen correctly, h_{SHR} becomes very effective for hashing a given dictionary statically; an example is given in this paper. There are some other functions, based on replacement of symbols, which are also quite suitable for static hashing. The main problem is how to find an appropriate value for T. A possible solution is offered. The core idea is that if it is possible to fix T in exactly one element in order to reduce the number of hash collisions, this should be done; otherwise T should be randomized.**

***Keywords* — fast hash function, static hashing, replacement of alphabetic symbols.**

REFERENCES

[1] Knuth D.E. The Art of Computer Programming. V. 3. Searching and Sorting. Second Edition. Reading, Massachusetts: Addison – Wesley, 1973. xiv + 782 pp.

[2] Fredman M.L., Komlós J., Szemerédi E. Storing a Sparse Table with O(1) Worst Case Access Time // Journal of the ACM. 1984. V. 31. № 3. P. 538–544.

[3] Дябин М.И., Решетников А.В., Саксонов Е.А. Быстрый метод генерации псевдослучайных векторов большой размерности для тестирования систем на кристалле // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2021. Выпуск 4. С. 86-91. doi:10.31114/2078-7707-2021-4-86-91 (in Russian).

[4] Pearson P.K. Fast Hashing of Variable-Length Text Strings // Communications of the ACM. 1990. V. 33. № 6. P. 677–680.