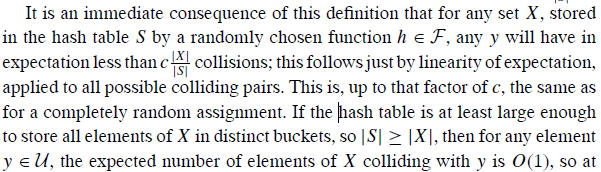
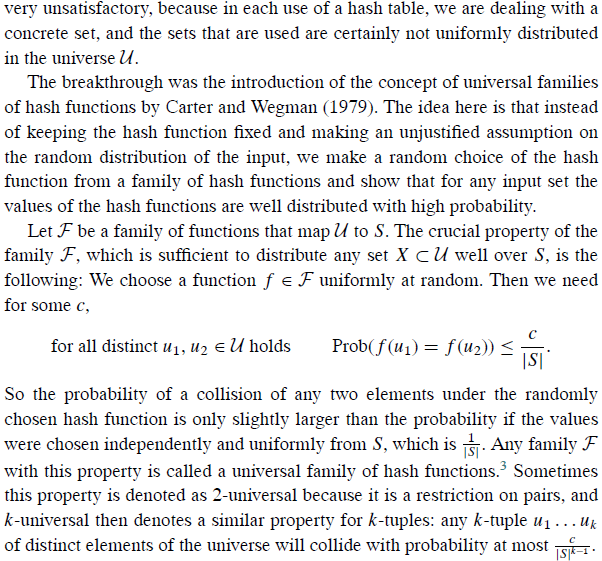
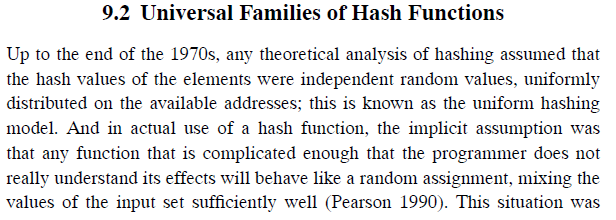
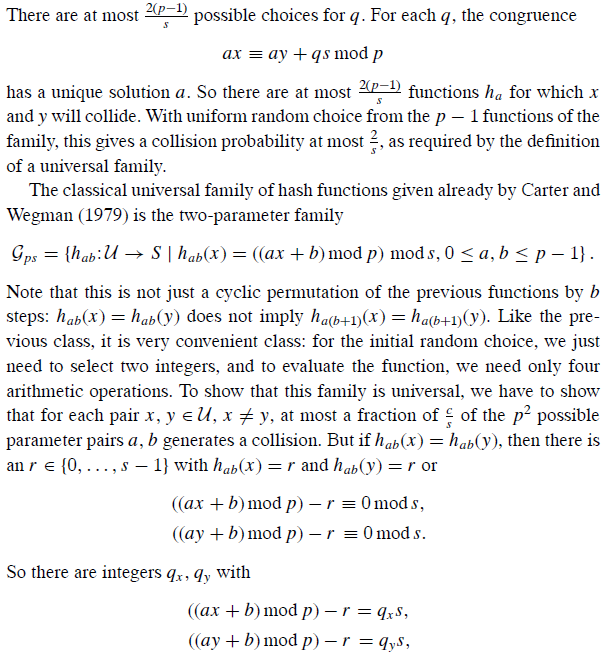
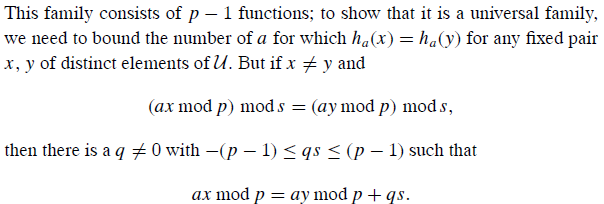
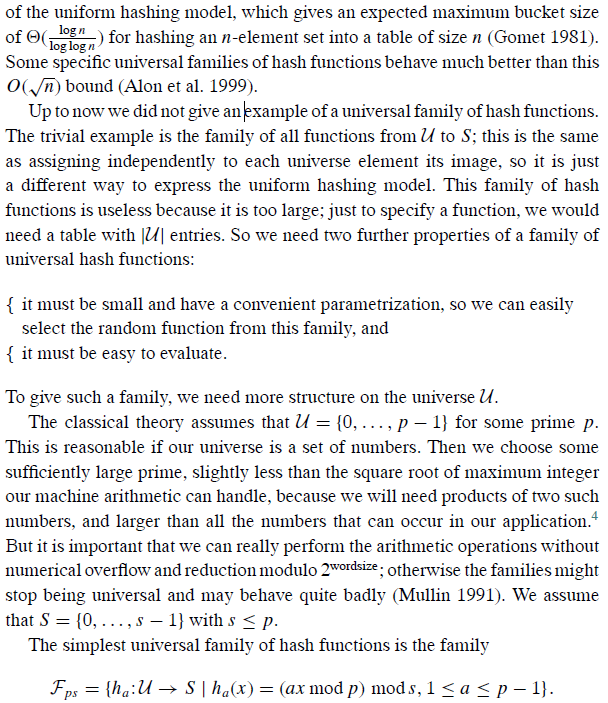
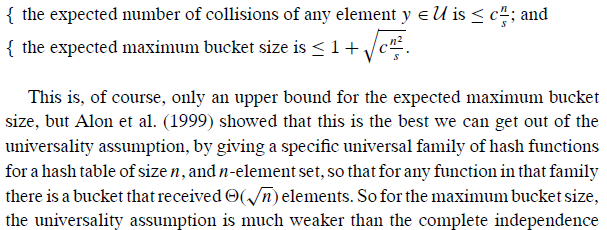
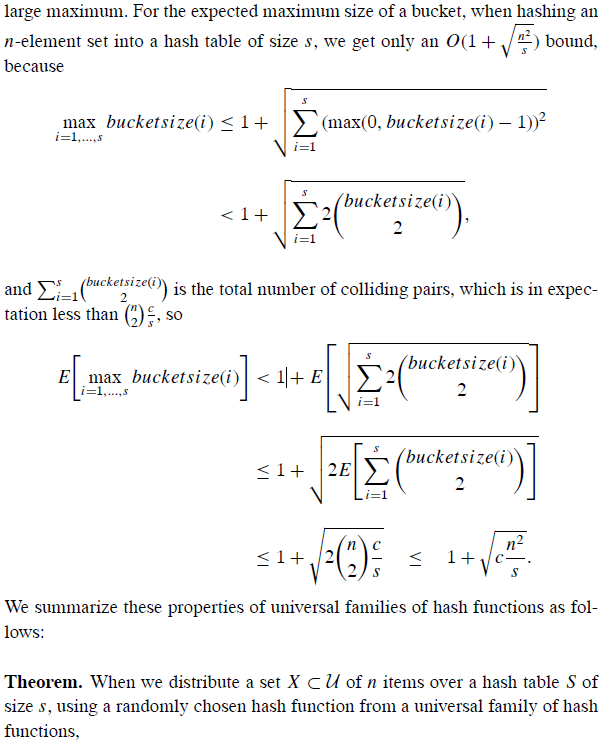
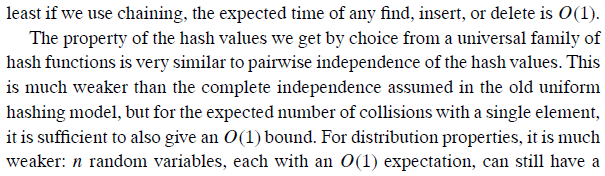
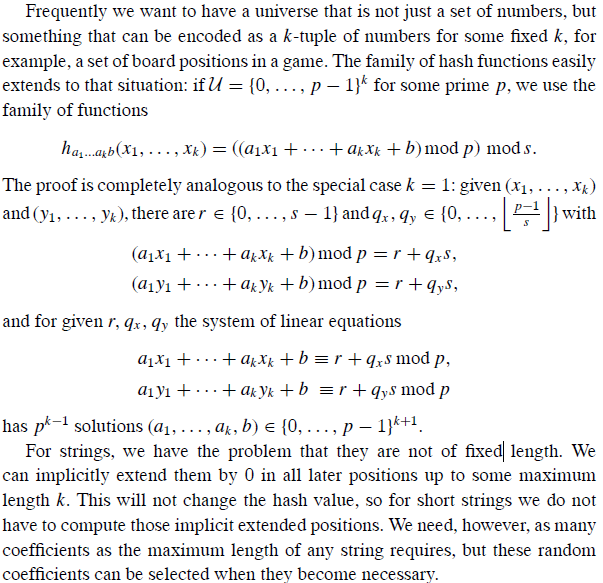
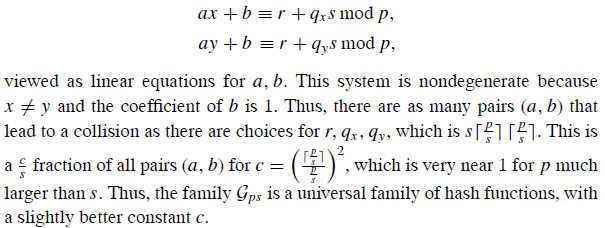
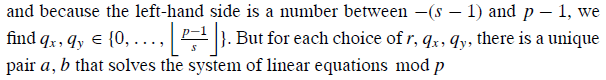
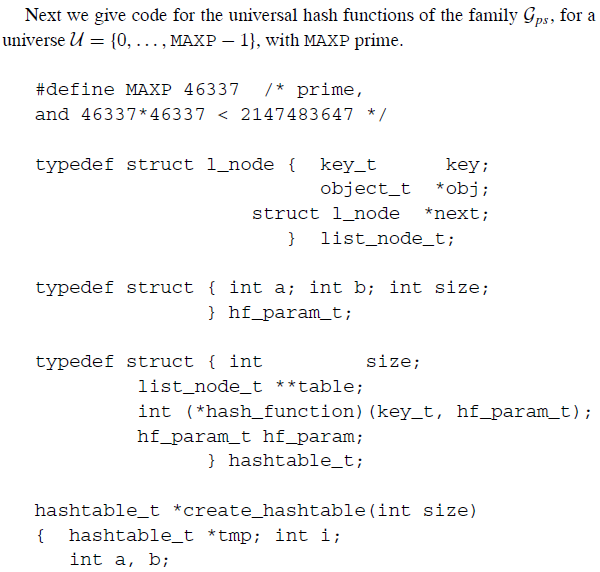
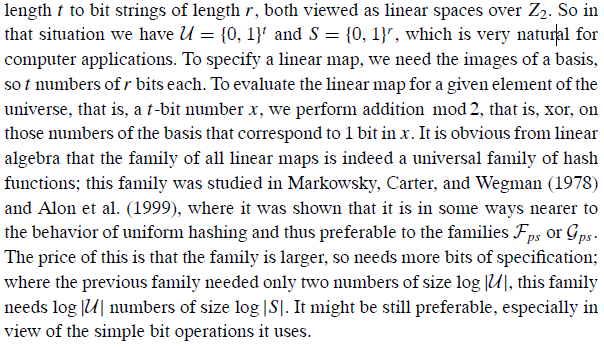
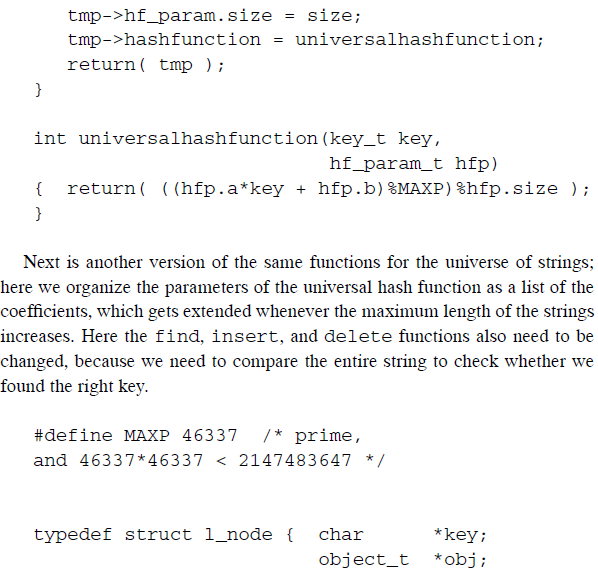
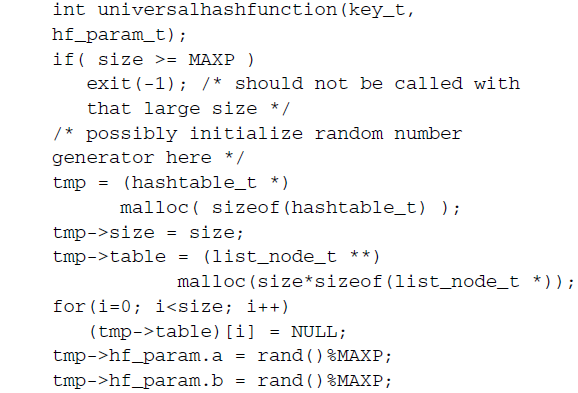
**Нужно исправить перевод текста:**

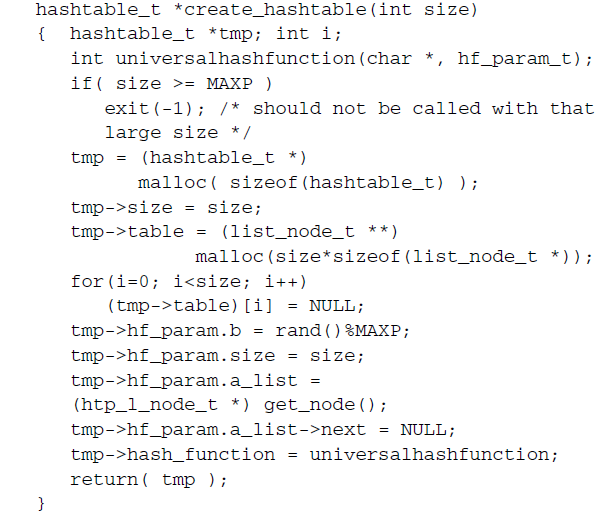
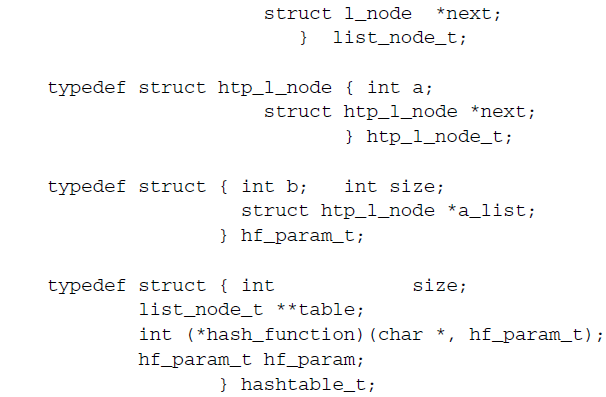


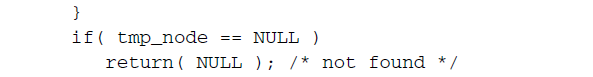
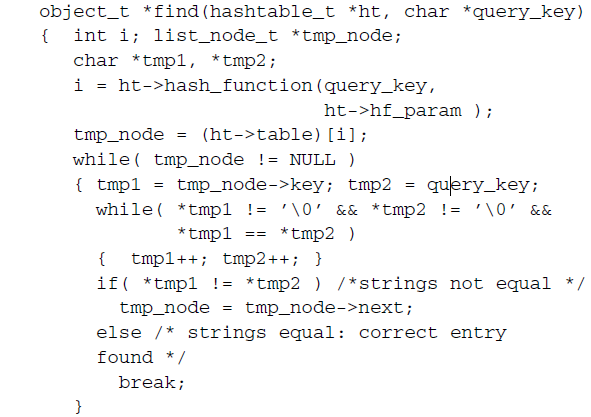
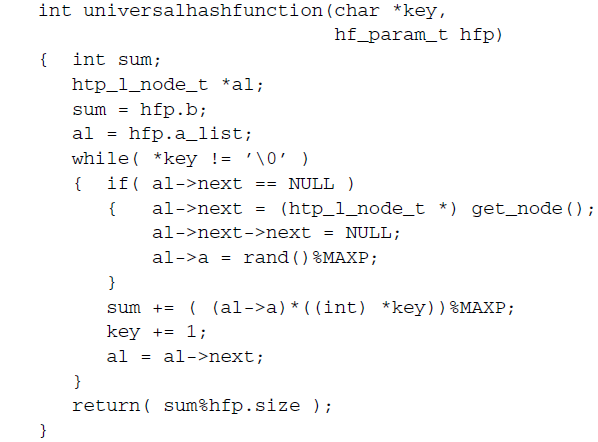


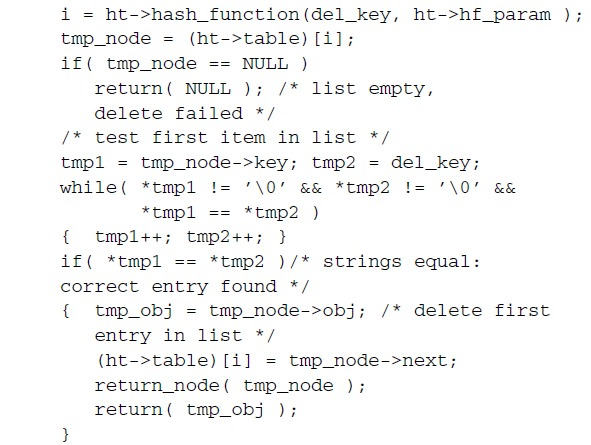
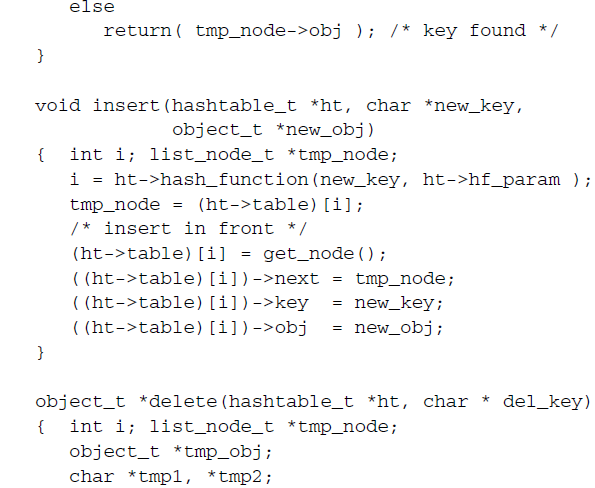


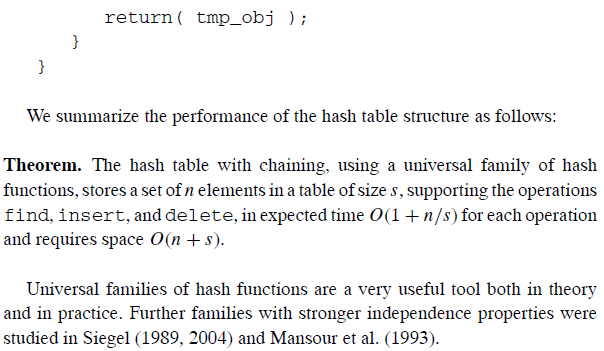
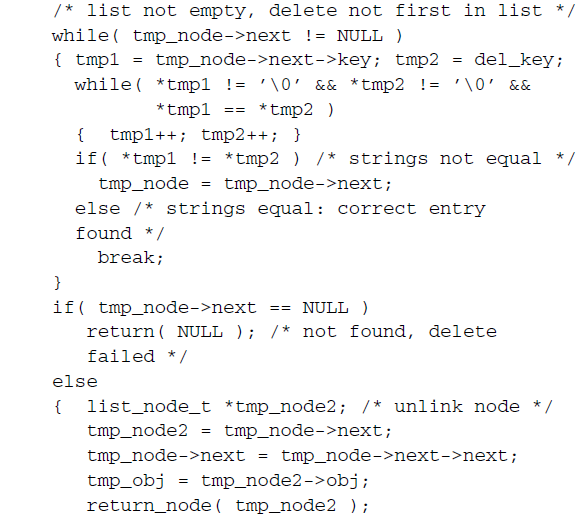
C:\Users\gnoma\Desktop\14.PNG











**Перевод:**

Универсальные семейства хэш-функций.

До конца 1970-х годов любой теоретический анализ хэширования предполагал, что хэш-значения элементов являются независимыми случайными величинами, равномерно распределенными на доступных адресах; это известно как единая модель хэширования. А в фактическом использовании хэш-функций, неявно предполагалось, что любая функция, которая сложна настолько, что программист не понимает ее последствий, будет вести себя в виде случайных операций, достаточно хорошо смешивая значения входных наборов. (Pearson 1990). Эта ситуация была очень неудовлетворительна, потому что в каждом использовании хэш-таблицы, мы имеем дело с конкретным набором, а множества, которые используются, конечно, не равномерно распределены во Вселенной U.

Прорывом стало введение понятия универсальных семейств хеш-функций ученых Carter и Wegman (1979). Идея состоит в том, что вместо того, чтобы хранить хэш-функцию фиксированной и делать неоправданные предположения о случайном распределении на входе, мы делаем случайный выбор хэш-функции из семейства хэш-функций и показываем, что для любого входного набора значений хэш-функции распределены с высокой вероятностью.

Пусть F есть семейство функций, отображающих U к S. Решающим свойством семейства F, которое является достаточным для распространения любого набора X с U свыше S, заключается в следующем: Выберем функцию равномерно наугад. Затем нам нужно для некоторых с,

для любых различных u1, u2, принадлежащих к U

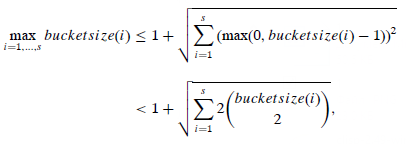
Prob(f(u1) = f(u2))<= С /|S|.

Таким образом, вероятность столкновения любых двух элементов, случайно выбранных хеш-функцией, лишь немного больше, чем вероятность того, если значения были бы выбраны независимо и равномерно из S, которая равна 1 /|S|. Любое семейство F с этим свойством называется универсальным семейством хэш-функций. Иногда это свойство обозначается как 2-универсальное, потому что это ограничение на пары, а k-универсальное затем обозначает аналогичное свойство для k-кортежей: любой k-кортеж u1. . .uk различных элементов вселенной столкнётся с вероятностью не более .

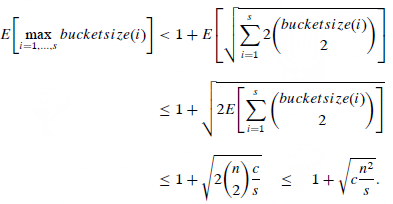
Это непосредственно вытекает из определения, что для любого множества X, хранящемся в хэш-таблице S случайно выбранной функции h, принадлежащей F, любой у будет иметь в ожидании меньше, чем c\*|X|/|S|столкновений; это следует только по линейности ожидания, применяется ко всем возможным встречным парам. Если хэш-таблица, по крайней мере достаточно большая для хранения всех элементов X в разных сегментах, соответственно |S|>=|X|, то для любого элемента у, принадлежащего к U, ожидаемое число элементов X столкновения с у является O(1), так по крайней мере, если мы используем цепочки, ожидаемое время нахождения, вставки или удаления является O (1).

Свойство хеш-значения, которые мы получаем по своему выбору от универсального семейства хеш-функций очень похоже на попарные независимости хэш-значения. Это намного слабее, чем полная независимость, предполагается в старой модели равномерного хеширования, но для ожидаемого числа столкновений с одним элементом достаточно, чтобы также дать O (1). Для распределения свойств, это гораздо слабее: n случайных величин, каждая с ожиданием O(1), все еще может иметь большой максимум.

Для ожидаемого максимального размера из сегмента, при хэшировании n-элементного множества в хэш-таблице размером S, мы получаем только , потому что



И C:\Users\gnoma_000\Desktop\2.PNG является общим количеством встречных пар, которое находится в ожидании меньше чем , таким образом:



Мы суммировали эти свойства универсальных семейств хеш-функций следующим образом:

**Теорема.** Когда мы распределяем ряд X с U из n элементов по хэш-таблице S размером s, используя случайно выбранную хэш-функцию из универсального семейства хэш-функций,

{Ожидаемое число столкновений любого элемента у, принадлежащему к U является <= c\*n/s; и

{Ожидаемый максимальный размер сегмента C:\Users\gnoma_000\Desktop\2.PNG

Это, конечно, только верхняя граница среднего для ожидаемого максимального размера сегмента, но Alon et al. (1999) показал, что это лучшее, что может выйти из универсальности предположения, давая конкретному универсальному семейству хэш-функции для хэш-таблицы размера n, и n-элементного множества, так что для любой функции в этой семье есть сегмент, который получил (n) элементов. Поэтому для максимального размера сегмента, универсальность предполагается намного слабее, чем полная независимость единой модели хеширования, которая дает ожидаемый максимум размера сегмента C:\Users\gnoma_000\Desktop\2.PNGдля хэширования n-элементного множества в таблицу размера n (Gomet 1981). Некоторые конкретные универсальные семейства хэш-функций ведут себя гораздо лучше, чем это (Alon et al. 1999).

До настоящего времени мы не можем привести пример универсального семейства хеш-функций. Тривиальным примером является семейство всех функций от U к S; это то же самое, как и присвоение независимо к каждому элементу Вселенной свой образ, так что это просто другой способ выразить единую модель хэширования. Это семейство хеш-функций бесполезно, потому что оно слишком большое; чтобы просто указать функцию, нам потребуется таблица с |U| записей. Так что нам нужно еще два свойства семейства универсальных хэш-функций:

{оно должно быть небольшим и иметь удобную параметризацию, таким образом мы сможем легко выбирать случайную функцию из этого семейства, и

{оно должно быть легко оценено.

Чтобы получить такое семейство, нам нужно больше структур на вселенной U.

Классическая теория предполагает, что U = {0,. . . , р - 1} для некоторого простого р. Это разумно, если наша Вселенная представляет собой набор цифр. Затем выберем некоторое достаточно большое простое число, несколько меньше, чем квадратный корень из максимального целого, нашей машинной арифметикой можно справиться, потому что нам будет нужно произведения двух таких чисел и больше, чем все числа, которые могут возникнуть в нашем приложении.

Но важно, что мы можем реально выполнять арифметические операции без численного сокращения overflowand по модулю ; в противном случае семейства могут перестать быть универсальными и могут вести себя довольно плохо (Mullin 1991). Мы предполагаем, что S = {0,. . . , s - 1} с s <= р.

Простейшим универсальным семейством хеш-функций является семейство:

C:\Users\gnoma_000\Desktop\1.PNG

Эта семья состоит из р - 1 функций; показывая, что она является универсальной семьёй, мы должны ограничивать количество "а", для которого hа(х) = hа(у) для любых фиксированных х, у пар, у различных элементов из U. Но если х ≠ у и

C:\Users\gnoma_000\Desktop\1.PNG

То есть c –(p – 1) <= qs <= (p – 1) такое, что

C:\Users\gnoma_000\Desktop\1.PNG

Существует не более возможных вариантов для q. Для каждого q, конгруэнтность

C:\Users\gnoma_000\Desktop\1.PNG

имеет уникальное решение *a*. Таким образом существует не более функций ha, для которых х и у столкнуться. При равномерном случайном выборе из р - 1 функций семейства вероятность столкновения не более 2/s, как это требует определение универсального семейства.

Классическое универсальное семейство хэш-функций, заданных уже Картером и Вэгманом (1979) является двух-параметрическим семейством

C:\Users\gnoma_000\Desktop\1.PNG

Обратите внимание, что это не просто циклическая перестановка предыдущих функций из В шагов: hab (х) = hab (y) не подразумевает ha(b+1) (x) = ha(b+1)(y). Как и предыдущий класс, это очень удобный класс: для первоначального случайного выбора, нам просто нужно выбрать два целых числа и оценить функцию, нам нужно только четыре арифметических операций. Чтобы показать, что это семейство является универсальным, мы должны показать, что для каждой пары х, у 𝟄 U, х ≠ у, в большинстве c/s из возможны параметры пар а, b порождающие столкновения. Но если hab(x) = hab (y), то существует r 𝟄 {0, . . .,s – 1} c hab (x) = r и hab (y) = r или

C:\Users\gnoma_000\Desktop\1.PNG

Таким образом, существуют целые qx, qy c

C:\Users\gnoma_000\Desktop\1.PNG

и потому, что левая часть представляет собой число от –(s – 1) и p - 1, находим C:\Users\gnoma_000\Desktop\1.PNG. Но для каждого выбора r, qx, qy, существует единственная пара a, b, которая решает систему линейных уравнений по модулю р

C:\Users\gnoma_000\Desktop\1.PNG

рассматриваемые как линейные уравнения для a, b. Эта система является невырожденной, потому что х ≠ у и коэффициент b равен 1. Таким образом, существует столько пар (а, b), которые приводят к столкновению, поскольку есть выбор для r, qx, qy , который равен C:\Users\gnoma_000\Desktop\1.PNG Это с/s часть всех пар (а, b) при с =C:\Users\gnoma_000\Desktop\1.PNG, которая расположена очень близко к 1 для р гораздо больше, чем s. Таким образом, семейство Gps является универсальным семейством хэш-функций, с несколько улучшенной константой с.

Часто мы хотим, чтобы Вселенная была не просто набором цифр, а тем, что может быть закодировано как k-набор чисел для некоторых фиксированных k, например, набор позиций на доске в игре. Семейство хэш-функций легко распространяется и на эту ситуацию: если U = { для некоторого простого р, мы используем семейство функций

C:\Users\gnoma_000\Desktop\1.PNG

Доказательство полностью аналогично специальному случаю при k = 1: учитывая (x1, ..., хк) и (y1, ..., ук), имеем r 𝟄 {0,. . . , s - 1} и qx, qy 𝟄 {0,. . . , C:\Users\gnoma_000\Desktop\1.PNG} c

C:\Users\gnoma_000\Desktop\1.PNG

и для данных r, qx, qy системы линейных уравнений

C:\Users\gnoma_000\Desktop\1.PNG

имеют решения C:\Users\gnoma_000\Desktop\1.PNG

Для строк имеется проблема - они не имеют фиксированной длины. Мы можем неявно расширить их на 0 во всех дальнейших позициях до некоторого максимального k. Это не изменит значения хэш-функции, поэтому для коротких строк нам не придется вычислять эти неявные расширенные позиции. Нам нужно, впрочем, как и многим коэффициентам, максимальная длина любой строки, но эти случайные коэффициенты можно выбрать, когда в них возникнет необходимость.

Другое универсальное семейство хэш-функций, что является одновременно простым в реализации и хорошей производительности в семействе всех линейных отображений из битовых строк длиной t до битовых строк длиной r, оба рассматриваются как линейные пространства над Z2. Таким образом, в этой ситуации мы имеем U = T и S = ​​, что очень естественно для компьютерных приложений. Чтобы задать линейное отображение, нам нужно образы основе, таким образом t количество бит каждого R . Для оценки линейной карты для данного элемента вселенной, то есть, по-битовое число х, мы выполняем сложения по модулю 2, то есть исключающее, на те числа из основы, которые соответствуют одному биту х. Очевидно, из линейной алгебры, что семейство всех линейных отображений действительно универсальное семейство хэш-функций; эта семейство было изучено в Марковском, такими учеными как Carter, и Wegman и Alon et al. (1999), где было показано, что это в некотором смысле приближались к поведению равномерного хэширования и, следовательно, предпочтительнее были семейства Fps или Gps. Ценой этого является то, что семья больше, поэтому нужно больше битов спецификации; где предыдущей семье нужны были только два числа размера log|U|, этому семейству необходимо log |*U*| чисел размера log |*S*|. Это может быть еще более предпочтительным, особенно с учетом простых битовых операций, которые оно использует. Ниже мы приводим код для универсальной хэш-функции семьи Gps и для вселенной U = {0,. . . , MAXP - 1}, с MAXP простым.

#define MAXP 46337 /\* prime,

and 46337\*46337 < 2147483647 \*/

typedef struct l\_node { key\_t key;

object\_t \*obj;

struct l\_node \*next;

} list\_node\_t;

typedef struct { int a; int b; int size;

} hf\_param\_t;

typedef struct { int size;

list\_node\_t \*\*table;

int (\*hash\_function)(key\_t, hf\_param\_t);

hf\_param\_t hf\_param;

} hashtable\_t;

hashtable\_t \*create\_hashtable(int size)

{ hashtable\_t \*tmp; int i;

int a, b;

int universalhashfunction(key\_t,

hf\_param\_t);

if( size >= MAXP )

exit(-1); /\* should not be called with

that large size \*/

/\* possibly initialize random number

generator here \*/

tmp = (hashtable\_t \*)

malloc( sizeof(hashtable\_t) );

tmp->size = size;

tmp->table = (list\_node\_t \*\*)

malloc(size\*sizeof(list\_node\_t \*));

for(i=0; i<size; i++)

(tmp->table)[i] = NULL;

tmp->hf\_param.a = rand()%MAXP;

tmp->hf\_param.b = rand()%MAXP;

tmp->hf\_param.size = size;

tmp->hashfunction = universalhashfunction;

return( tmp );

}

int universalhashfunction(key\_t key,

hf\_param\_t hfp)

{ return( ((hfp.a\*key + hfp.b)%MAXP)%hfp.size );

}

Далее идет еще один вариант той же функции для вселенной строк;

Здесь мы приводим параметры универсальной хэш-функции как список коэффициентов, который продлевается всякий раз, когда максимальная длина строки увеличивается. Здесь нахождение, вставка и удаление функции также должны быть изменены, потому что мы должны сравнивать всю строку, чтобы проверить нашли ли мы правильный ключ.

#define MAXP 46337 /\* prime,

and 46337\*46337 < 2147483647 \*/

typedef struct l\_node { char \*key;

object\_t \*obj;

struct l\_node \*next;

} list\_node\_t;

typedef struct htp\_l\_node { int a;

struct htp\_l\_node \*next;

} htp\_l\_node\_t;

typedef struct { int b; int size;

struct htp\_l\_node \*a\_list;

} hf\_param\_t;

typedef struct { int size;

list\_node\_t \*\*table;

int (\*hash\_function)(char \*, hf\_param\_t);

hf\_param\_t hf\_param;

} hashtable\_t;

hashtable\_t \*create\_hashtable(int size)

{ hashtable\_t \*tmp; int i;

int universalhashfunction(char \*, hf\_param\_t);

if( size >= MAXP )

exit(-1); /\* should not be called with that

large size \*/

tmp = (hashtable\_t \*)

malloc( sizeof(hashtable\_t) );

tmp->size = size;

tmp->table = (list\_node\_t \*\*)

malloc(size\*sizeof(list\_node\_t \*));

for(i=0; i<size; i++)

(tmp->table)[i] = NULL;

tmp->hf\_param.b = rand()%MAXP;

tmp->hf\_param.size = size;

tmp->hf\_param.a\_list =

(htp\_l\_node\_t \*) get\_node();

tmp->hf\_param.a\_list->next = NULL;

tmp->hash\_function = universalhashfunction;

return( tmp );

}

int universalhashfunction(char \*key,

hf\_param\_t hfp)

{ int sum;

htp\_l\_node\_t \*al;

sum = hfp.b;

al = hfp.a\_list;

while( \*key != ’\0’ )

{ if( al->next == NULL )

{ al->next = (htp\_l\_node\_t \*) get\_node();

al->next->next = NULL;

al->a = rand()%MAXP;

}

sum += ( (al->a)\*((int) \*key))%MAXP;

key += 1;

al = al->next;

}

return( sum%hfp.size );

}

object\_t \*find(hashtable\_t \*ht, char \*query\_key)

{ int i; list\_node\_t \*tmp\_node;

char \*tmp1, \*tmp2;

i = ht->hash\_function(query\_key,

ht->hf\_param );

tmp\_node = (ht->table)[i];

while( tmp\_node != NULL )

{ tmp1 = tmp\_node->key; tmp2 = query\_key;

while( \*tmp1 != ’\0’ && \*tmp2 != ’\0’ &&

\*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 != \*tmp2 ) /\*strings not equal \*/

tmp\_node = tmp\_node->next;

else /\* strings equal: correct entry

found \*/

break;

}

if( tmp\_node == NULL )

return( NULL ); /\* not found \*/

else

return( tmp\_node->obj ); /\* key found \*/

}

void insert(hashtable\_t \*ht, char \*new\_key,

object\_t \*new\_obj)

{ int i; list\_node\_t \*tmp\_node;

i = ht->hash\_function(new\_key, ht->hf\_param );

tmp\_node = (ht->table)[i];

/\* insert in front \*/

(ht->table)[i] = get\_node();

((ht->table)[i])->next = tmp\_node;

((ht->table)[i])->key = new\_key;

((ht->table)[i])->obj = new\_obj;

}

object\_t \*delete(hashtable\_t \*ht, char \* del\_key)

{ int i; list\_node\_t \*tmp\_node;

object\_t \*tmp\_obj;

char \*tmp1, \*tmp2;

i = ht->hash\_function(del\_key, ht->hf\_param );

tmp\_node = (ht->table)[i];

if( tmp\_node == NULL )

return( NULL ); /\* list empty,

delete failed \*/

/\* test first item in list \*/

tmp1 = tmp\_node->key; tmp2 = del\_key;

while( \*tmp1 != ’\0’ && \*tmp2 != ’\0’ &&

\*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 == \*tmp2 )/\* strings equal:

correct entry found \*/

{ tmp\_obj = tmp\_node->obj; /\* delete first

entry in list \*/

(ht->table)[i] = tmp\_node->next;

return\_node( tmp\_node );

return( tmp\_obj );

}

/\* list not empty, delete not first in list \*/

while( tmp\_node->next != NULL )

{ tmp1 = tmp\_node->next->key; tmp2 = del\_key;

while( \*tmp1 != ’\0’ && \*tmp2 != ’\0’ &&

\*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 != \*tmp2 ) /\* strings not equal \*/

tmp\_node = tmp\_node->next;

else /\* strings equal: correct entry

found \*/

break;

}

if( tmp\_node->next == NULL )

return( NULL ); /\* not found, delete

failed \*/

else

{ list\_node\_t \*tmp\_node2; /\* unlink node \*/

tmp\_node2 = tmp\_node->next;

tmp\_node->next = tmp\_node->next->next;

tmp\_obj = tmp\_node2->obj;

return\_node( tmp\_node2 );

return( tmp\_obj );

}

}

Мы суммируем производительность структуры хеш-таблицы следующим образом:

**Теорема.** Хэш-таблица с цепочкой, используя универсальное семейство хеш-функций, сохраняет набор из n элементов в таблице, размер s, поддержку операций: найти, вставить и удалить, в ожидаемое время O (1 + n/s) для каждой операции и требует пространства O (n + s). Универсальные семейства хэш-функций являются весьма полезным инструментом в теории и на практике. Дальнейшие семейства с более сильными независимостями свойств изучались в Siegel (1989, 2004) и Mansour et al. (1993)