

**Курс «Алгоритмы и алгоритмические языки»
1 семестр 2013/2014**

Лекция 22

Хеш-таблицы

- ◆ Словарные операции: *добавление*, *поиск* и *удаление* элементов по их ключам.
- ◆ Организуется таблица ключей: массив **Index[m]** длины **m**, элементы которого содержат значение ключа и указатель на данные (информацию), соответствующие этому ключу.
 - ◆ **Прямая адресация.** Применяется, когда количество возможных ключей невелико: например, ключи перенумерованы целыми числами из множества $U = \{0, 1, 2, \dots, m - 1\}$, где m не очень большое число.
 - ◆ В случае прямой адресации ключ с номером k соответствует элементу **Index[k]**. Этот ключ обычно не записывается в элемент массива, т.к. совпадает с индексом.
 - ◆ Все три словарные операции выполняются за время порядка $O(1)$.
 - ◆ Основной недостаток прямой адресации – таблица *Index* занимает слишком много места, если множество всевозможных ключей U достаточно велико (m большое целое число).

Хеш-таблицы

- ◇ **Хеширование** тоже позволяет обеспечить среднее время операций с данными $T_{\text{cp}}(n) = O(1)$ и тоже за счет использования таблицы *Index*.
- ◇ Хеш-таблица использует память объемом $\Theta(|K|)$, где $|K|$ – мощность множества использованных ключей (правда это оценка в среднем, а не в худшем случае, да и то при определенных предположениях).
- ◇ Пример использования хеширования – таблица идентификаторов программы, составляемая компилятором.
- ◇ В случае хеш-адресации элементу с ключом *key* отводится строка таблицы с номером $hash(key)$, где $hash: U \rightarrow \{0, 1, 2, \dots, m - 1\}$ – хеш-функция. Число $hash(key)$ называется *хеш-значением* ключа *key*.
- ◇ Если хеш-значения ключей key_1 и key_2 совпадают ($hash(key_1) == hash(key_2)$), говорят, что случилась *коллизия*. Выбрать хеш-функцию, для которой коллизии исключены, возможно лишь тогда, когда все возможные значения ключей заранее известны. В общем же случае коллизии неизбежны, так как $|U| > m$.

Хеш-таблицы

- ◇ Простейший способ обработки коллизий – сцепление элементов с одинаковыми значениями хеш-функции: все такие элементы сцепляются в список, а в хеш-таблицу помещается указатель на первый элемент этого списка. В пределах каждого такого списка осуществляется последовательный поиск.
- ◇ В случае использования двусвязного списка среднее время выполнения каждой из трех словарных операций будет иметь порядок $O(1)$. Основная трудность – в поиске по списку, но коллизий не очень много и $hash(key)$ можно выбрать так, чтобы списки были достаточно короткими.
- ◇ Примером хеш-таблицы с цепочками является записная книжка с алфавитом.

Хеш-таблицы

- ◆ Устройство простой хеш-таблицы (реализация хеширования с цепочками).
 - ◆ Задается некоторое фиксированное число m (типичные значения m от 100 до 1,000,000).
 - ◆ Создается массив **Index[m]** указателей начал двусвязных списков (цепочек), который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения *NULL*.
 - ◆ Задается хеш-функция $hash()$, которая получает на вход ключи и выдает значение от 0 до $m - 1$.
 - ◆ При добавлении пары ($key, value$) вычисляется $h = hash(key)$ и пара добавляется в список **Index[h]**.
 - ◆ При удалении либо поиске пары ($key, value$) вычисляется $h = hash(key)$ и происходит удаление либо поиск пары ($key, value$) в списке **Index[h]**.

Хеш-таблицы



Анализ хеширования с цепочками.

- ◆ Пусть **Index[m]** – хеш-таблица с m позициями, в которую занесено n пар (*key*, *value*). Отношение $\alpha = n/m$ называется *коэффициентом заполнения* хеш-таблицы.

- ◆ Коэффициент заполнения α позволяет судить о качестве хеш-функции:

пусть $M = \frac{1}{m} \sum_{i=0}^{m-1} |Index[i]|$ – средняя длина списков;

если $hash(key)$ – «хорошая» хеш-функция, то

дисперсия $D = \frac{1}{m} \sum_{i=0}^{m-1} (M - |Index[i]|)^2 \leq \alpha$.

- ◆ Это условие исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список и поиск в этом списке из n элементов имеет среднее время $\Theta(n)$.

Хеш-таблицы

- ◆ Анализ хеширования с цепочками.
 - ◆ *Равномерное хеширование*: хеш-функция подобрана таким образом, что каждый данный элемент может попасть в любую из t позиций хеш-таблицы с равной вероятностью, независимо от того, куда попали другие элементы.
 - ◆ Условие из предыдущего слайда выполняется и *средняя длина каждого из t списков хеш-таблицы с коэффициентом заполнения α равна α* .
 - ◆ Среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка α , так как поиск сводится к просмотру одного из списков.
 - ◆ Поскольку среднее время вычисления хеш-функции равно $\Theta(1)$, то среднее время выполнения каждой из словарных операций с учетом вычисления хеш-функции равно $\Theta(1 + \alpha)$.

Хеш-таблицы

- ◇ **Теорема.** Пусть T – хеш-таблица с цепочками, имеющая коэффициент заполнения α , причем хеширование равномерно. Тогда при поиске элемента, **отсутствующего** в таблице, будет просмотрено в среднем α элементов таблицы, а время поиска, включая время на вычисление хеш-функции, будет равно $\Theta(1 + \alpha)$.
- ◇ **Теорема.** При равномерном хешировании среднее время **успешного** поиска в хеш-таблице с коэффициентом заполнения α есть $\Theta(1 + \alpha)$.
 - ◆ **Замечание.** Теорема не сводится к предыдущей, так как в предыдущей теореме оценивалось среднее число действий, необходимых для поиска случайного элемента, равновероятно попадающего в любую из ячеек таблицы.
 - ◆ В этой теореме сначала рассматривается случайно выбранная последовательность элементов, добавляемых в таблицу (на каждом шаге все значения ключа равновероятны и шаги независимы); потом в полученной таблице выбираем элемент для поиска, считая, что все ее элементы равновероятны.
- ◇ Из теорем следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть $O(1)$.

Методы построения хеш-функций

- ◇ Построение хеш-функции **методом деления с остатком**.
 - ◆ Хеш-функция $hash(key)$ определяется соотношением **$hash(key) = key \% m$** .
 - ◆ При правильном выборе m такая хеш-функция обеспечивает распределение, близкое к равномерному.
 - ◆ Правильный выбор m : в качестве m выбирается достаточно большое простое число, далеко отстоящее от степеней двойки.
 - ◆ Например, если устраивает средняя длина списков 3, а число записей, доступ к которым нужно обеспечить с помощью хеш-таблицы ≈ 2000 , то можно взять $m = 2000/3 \approx 701$. Тогда $hash(key) = key \% 701$.
 - ◆ Недостаток: в качестве m нельзя брать степень двойки, так как если $m = 2^p$, то $hash(key)$ – это просто p младших битов числа key .

Методы построения хеш-функций

- ◆ Построение хеш-функции **методом умножения**.
 - ◆ Пусть количество хеш-значений равно m . Выберем и зафиксируем вещественную константу v , $0 < v < 1$; положим $hash(key) = \lfloor m(\text{frac}(key \cdot v)) \rfloor$
 $\text{frac}(key \cdot v)$ – дробная часть числа $key \cdot v$.
 - ◆ Достоинство метода умножения в том, что качество хеш-функции слабо зависит от выбора m . Обычно в качестве m выбирают степень двойки, так как в этом случае умножение на m сводится к сдвигу.
 - ◆ **Пример.** Пусть в используемом компьютере длина слова равна w битам и ключ key помещается в одно слово.
 - ◆ Если $m = 2^p$, то вычисление $hash(key)$ можно выполнить следующим образом: умножим key на w -битовое целое число $v \cdot 2^w$; получится $2w$ -битовое число r_0 .
В качестве значения $hash(key)$ возьмем старшие p битов “дробной” части числа $r_0 / 2^w$ ($r_0 \% 2^w$ или обнуление w старших разрядов, потом умножение на $m = 2^p$).
 - ◆ Согласно Д. Кнуту выбор $v = (\sqrt{5} - 1) / 2 = 0.6180339887\dots$ является удачным.

Хеш-функции: программы

```
#define MAX 701    /* размер хеш-таблицы */
struct htype {
    int key;        /* ключ */
    int val;        /* значение элемента данных */
    struct htype *next; /* указатель на следующий элемент
                        цепочки */
    struct htype *prvs; /* указатель на предыдущий элемент
                        цепочки */
};
struct htype *index[MAX];
```

Хеш-функции: программы

```
#define MAX 701    /* размер хеш-таблицы */
static inline int hash (int key) {
    return key % MAX;
}
/* инициализация хеш-таблицы */
void init (void) {
    int i;
    for (i = 0; i < MAX; i++)
        index[i] = NULL; /* массив начал цепочек */
}
```

Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k:
   если элемент с ключом k найден, возвращаем указатель
   на него, если нет, возвращаем NULL */
struct htype *search (int k) {
    int h;
    struct htype *p;
    /* вычисление хеш-адреса */
    h = hash (k);
    /* поиск ключа k */
    if (index[h]) {
        p = index[h];
        do {
            if (p->key == k)
                return p;
            else
                p = p->next;
        } while (p);
    }
    return NULL;
}
```

Хеш-функции: программы

```
/* Порождение нового элемента цепочки и возврат указателя  
на него */
```

```
struct htype *new (void) {  
    struct htype *p;  
    p = (struct htype *) malloc (sizeof (struct htype));  
    if (!p)  
        abort ();  
    p->key = -1;  
    p->val = 0;  
    p->next = NULL;  
    p->prvs = NULL;  
    return p;  
}
```

Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k: если элемент с ключом k найден, возвращаем значение true и указатель на найденный элемент; если элемент не найден, возвращаем значение false и указатель на последний элемент либо NULL, если цепочка пустая */
```

```
static bool search_internal (int k, struct htype **r) {  
    struct htype *p, *q;  
  
    if ((p = index[hash (k)]) != NULL) {  
        do {  
            if (p->key == k) {  
                *r = p;  
                return true;  
            }  
            else  
                q = p, p = p->next;  
        } while (p);  
        *r = q;  
    } else  
        *r = NULL;  
    return false;  
}
```

Хеш-функции: программы

```
/* Добавление новой пары (key, value) */
void insert (int k, int v) {
    struct htype *p, *q;
    /* Если элемент с ключом k уже имеется в цепочке,
       изменяем его значение на v */
    if (search_internal (k, &p))
        p->val = v;
    else {
        /* Если элемента с ключом k в цепочке нет */
        /* порождение и инициализация нового элемента цепочки */
        q = new ();
        q->key = k;
        q->val = v;
        /* Включение порожденного элемента в цепочку */
        if (p) {
            p->next = q;
            q->prvs = p;
        } else
            index[hash (k)] = q;
    }
}
```

Хеш-функции: программы

```
/* Исключение пары (key, value) */
void delete (int k, int v) {
    struct htype *p;
    if (search_internal (k, &p)) {
        if (p->prvs)
            p->prvs->next = p->next;
        else
            index[hash (k)] = p->next;
        if (p->next)
            p->next->prvs = p->prvs;
        free (p);
    }
    /* иначе ничего не нашли, удалять не нужно */
}
```

Хеширование с открытой адресацией

- ◇ Все записи хранятся в самой хеш-таблице: каждая ячейка таблицы (массива длины m) содержит либо хранимый элемент, либо `NULL`. Указатели вообще не используются, что приводит к сохранению места и ускорению поиска.
- ◇ Таким образом, коэффициент заполнения $\alpha = n/m$ не больше 1.
- ◇ **Поиск (search)**: мы определенным образом просматриваем элементы таблицы, пока не найдем искомый или не убедимся, что искомый элемент отсутствует.
- ◇ Просматриваются не все элементы (иначе это был бы последовательный поиск), а только некоторые согласно значению хеш-функции, которая в этом случае имеет два аргумента – ключ и «номер попытки»:

$$\text{hash}: U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

- ◇ Функцию *hash* нужно выбрать такой, чтобы в последовательности проб $\langle \text{hash}(k, 0), \text{hash}(k, 1), \dots, \text{hash}(k, m - 1) \rangle$ каждый номер ячейки $0, 1, \dots, m - 1$ встретился только один раз.
- ◇ Если при поиске мы добираемся до ячейки, содержащей `NULL`, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку).

Хеширование с открытой адресацией: программы

```
#define m 1999
struct htype {
    int key;           /* ключ */
    int val;          /* значение элемента данных */
} *index[m];

/* Поиск элемента */
struct htype *search (int k) {
    int i = 0, j;

    do {
        j = hash (key, i);
        if (index[j] && index[j]->key == k)
            return index[j];
    } while (!index[j] || ++i == m);
    return NULL;
}
```

Хеширование с открытой адресацией: программы

```
/* Добавление элемента */
int insert (int k, int v) {
    int i = 0, j;

    do {
        j = hash (key, i);
        if (index[j] && index[j]->key == k) {
            index[j]->val = v;
            return j;
        }
    } while (!index[j] || ++i == m);
    /* Таблица может оказаться заполненной */
    if (i == m)
        return -1; /* Или расширим index */
    index[j] = new ();
    index[j]->key = k, index[j]->val = v;
    return j;
}
```

Хеширование с открытой адресацией: программы

```
/* Внутренний поиск: вернем индекс массива */
static int search_internal (int k) {
    int i = 0, j;

    do {
        j = hash (key, i);
        if (index[j] && index[j]->key == k)
            return j;
    } while (!index[j] || ++i == m);
    return -1;
}

/* Внешний поиск легко реализуется через внутренний */
struct htype *search (int k) {
    int j = search_internal (k);
    return j >= 0 ? index[j] : NULL;
}
```

Хеширование с открытой адресацией: программы

```
/* Удаление элемента */  
void delete (int k) {  
    int j;  
  
    j = search_internal (k);  
    if (j < 0)  
        return;  
    /* Нельзя писать index[j] = NULL!  
        Будут потеряны ключи, возможно, находящиеся  
        за удаляемым ключом (с тем же хешем). */  
    ???  
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)

/* Удаление элемента */
void delete (int k) {
    int j;

    j = search_internal (k);
    if (j < 0)
        return;
    /* Нельзя писать index[j] = NULL! */
    free (index[j]);
    index[j] = SHADOW;
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(e1) ((e1) || (e1) == SHADOW)

static int search_internal (int k) {
    int i = 0, j;

    do {
        j = hash (key, i);
        if (!ISEMPTY (index[j]) && index[j]->key == k)
            return j;
    } while (!index[j] || ++i == m);
    return -1;
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(e1) ((e1) || (e1) == SHADOW)

/* Добавление элемента */
int insert (int k, int v) {
    int i = 0, j;

    do {
        j = hash (key, i);
        if (!ISEMPTY (index[j]) && index[j]->key == k) {
            index[j]->val = v;
            return j;
        }
    } while (! ISEMPTY (index[j]) || ++i == m);

    /* Таблица может оказаться заполненной (много вставок/удалений) */
    if (i == m)
        return -1; /* Или расширим index */
    index[j] = new ();
    index[j]->key = k, index[j]->val = v;
    return j;
}
```

Хеш-функции для открытой адресации

- ◇ *Линейная последовательность проб.*
Пусть $hash': U \rightarrow \{0, 1, \dots, m - 1\}$ – обычная хеш-функция.
Функция $hash(k, i) = (hash'(k) + i) \bmod m$
определяет *линейную последовательность проб*.
- ◇ При линейной последовательности проб начинают с ячейки $index[h'(k)]$, а потом перебирают ячейки таблицы подряд: $index[h'(k) + 1]$, $index[h'(k) + 2]$, ... (после $index[m - 1]$ переходят к $index[0]$).
- ◇ Существует лишь m различных последовательностей проб, т.к. каждая последовательность однозначно определяется своим первым элементом.

Хеш-функции для открытой адресации

- ◆ Серьезный недостаток – тенденция к образованию *кластеров* (длинных последовательностей занятых ячеек, идущих подряд), что удлиняет поиск:
 - ◆ Если в таблице все четные ячейки заняты, а нечетные ячейки свободны, то среднее число проб при поиске отсутствующего элемента равно 1,5.
 - ◆ Если же те же $m/2$ занятых ячеек идут подряд, то среднее число проб равно $(m/2)/2 = m/4$.
- ◆ Причины образования кластеров: если k заполненных ячеек идут подряд, то:
 - ◆ вероятность того, что при очередной вставке в таблицу будет использована ячейка, непосредственно следующая за ними, есть $(k + 1)/m$ (пропорционально «толщине слоя»),
 - ◆ вероятность использования конкретной ячейки, предшественница которой тоже свободна, всего лишь $1/m$.
- ◆ Таким образом, хеширование с использованием линейной последовательности проб далеко не равномерное.
- ◆ Возможное улучшение: добавляем не 1, а константу c , взаимно простую с m (для полного обхода таблицы).

Хеш-функции для открытой адресации

- ◇ Квадратичная последовательность проб:
 $hash(k, i) = (hash'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$,
 c_1 и $c_2 \neq 0$ – константы.
- ◇ Пробы начинаются с ячейки $index[h'(k)]$, а потом ячейки просматриваются не подряд, а по более сложному закону. Метод работает значительно лучше, чем линейный.
- ◇ Чтобы при просмотре таблицы $index$ использовались все ее ячейки, значения m , c_1 и c_2 следует брать не произвольными, а подбирать специально. Если обе константы равны единице:
 - ◆ находим $i \leftarrow hash'(k)$; полагаем $j \leftarrow 0$;
 - ◆ проверяем $index[i]$:
 - если она свободна, заносим в нее запись и выходим из алгоритма,
 - если нет – полагаем $j \leftarrow (j + 1) \bmod m$,
 $i \leftarrow (i + j) \bmod m$ и повторяем текущий шаг.

Хеш-функции для открытой адресации

- ◇ *Двойное хеширование* – один из лучших методов открытой адресации.
$$\text{hash}(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$
где $h_1(k)$ и $h_2(k)$ – обычные хеш-функции.
- ◇ Дополнительная хеш-функция $h_2(k)$ генерирует хеши, взаимно простые с m .
- ◇ Если основная и дополнительная функция существенно независимы (т.е. вероятность совпадения их хешей обратно пропорциональна квадрату m), то скучивания не происходит, а распределение ключей по таблице близко к случайному.
- ◇ *Оценки.* Среднее число проб для равномерного хеширования оценивается при успешном поиске как $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.
При коэффициенте заполнения 50% среднее число проб для успешного поиска $\leq 1,387$, а при 90% – $\leq 2,559$.
- ◇ При поиске отсутствующего элемента и при добавлении нового элемента оценка среднего числа проб $\frac{1}{1-\alpha}$.

Хеширование других данных

◆ Хеширование идентификаторов в компиляторе

```
hashval_t
htab_hash_string (const PTR p)
{
    const unsigned char *str = (const unsigned char *) p;
    hashval_t r = 0;
    unsigned char c;

    while ((c = *str++) != 0)
        r = r * 67 + c - 113;

    return r;
}
```

◆ Хеширование ключа переменной длины: в GCC используется

<http://burtleburtle.net/bob/hash/evahash.html>