## *Лекция 14 Хеш-таблицы*

14.1. Программы (двусвязный список).

```
#include <stdbool.h>
#define MAX 2000    /* 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 */
} *p, *index[MAX];

/* инициализация хеш-таблицы */
void init(void) {
  int i;
  for(i = 0; i < MAX; i++) {
     index[i] = NULL; /* хеш-таблица (массив начал цепочек) */
  }
}

/* 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 */
/* предшествующий элемент (если таковой имеется), либо NULL, если */
/* цепочка становится пустой */
bool search(int k, struct htype **p) {
  int h;
  struct htype *q, *qp;

  /* вычисление хеш-адреса */
  h = k % 701; /* хеш-адрес */

  /* поиск цепочки ключа k */
  if(index[h] != NULL) {
     q = index[h];
     /* поиск ключа k в цепочке */
     do {
        /* ключ k найден: возвращаем true и указатель */
        if(q -> key == k) {*p = q; return true;}
        else {qp = q; q = q -> next;}
     } while (q != NULL);
     /* ключ k в цепочке не найден: возвращаем false и указатель на */
     /* последний элемент цепочки (через параметр **p) */
     *p = qp;
  }
  /* цепочки для ключа k в массиве index нет: возвращаем false и */
  /* указатель NULL */
  else *p = NULL;
  return false;
```

```
}

/* Порождение нового элемента цепочки и возврат указателя на него */
struct htype *new() {
  struct htype *p;
  p = malloc(sizeof(struct htype)); //выделение памяти
  p -> key = -1;
  p -> val = 0;
  p -> next = NULL;
  p -> prvs = NULL;
  return p;
}

/* Добавление новой пары (key, value) */
insert(int k, int v) {
  struct htype *p, *q;
  int h;
  /* Если элемент с ключом k уже имеется в цепочке, изменяем его */
  /* значение на v */
  if(search(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 != NULL) {
          p->next = q;
          q->prvs = p;
     }
     else {
          h = k % 701;
          index[h] = q;
     }
  }
}

/* Исключение пары (key, value) */
delete(int k, int v) {
  struct htype *p;
  if(search(k, &p)) {
     if(p->prvs != NULL) p->prvs->next = p->next;
     else {
          h = k % 701;
          index[h] = p->next;
     }
     if(p->next != NULL) p->next->prvs = ->prvs;
     free(p);
  }
```

```
}
```

14.2. Хеширование с *открытой адресацией*.

14.2.1. Все записи хранятся в самой хеш-таблице: каждая ячейка таблицы (массива длины $m$, ячейки имеют номера 0, 1, …, $m$ – 1) содержит либо хранимый элемент, либо *nil*. Указатели вообще не используются, что приводит к сохранению места и ускорению поиска. Таким образом, *коэффициент заполнения* $\alpha = n/m$ не больше 1.

14.2.2. ***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»:

$$hash: U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Функцию *hash* нужно выбрать такой, чтобы в последовательности проб

$$\langle hash(k, 0), hash(k, 1), \ldots, hash(k, m-1)\rangle$$

каждый номер ячейки 0, 1, …, $m$ – 1 встретился только один раз. Если при поиске мы добираемся до ячейки, содержащей *nil*, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку). Функция **search**:

```
int search(struct trec *Table, int key) {
     int i, j;
     i = 0
     do {
          j = hash(key, i);
          if(Table[j] == key) return j;
          i++;
     }while(Table[j] == nil || i == m);
     return nil;
}
```

14.2.3. ***Добавление*** (**insert**): ищем первое свободное место (*nil*).

```
int insert(struct trec *Table, int key) {
     int i, j;
     i = 0
     do {
          j = hash(key, i);
          if(Table[j] == nil) {
               Table[j] == key;
               return j;
          }
          else i++;
}while(i == m);
```

14.2.4. ***Удаление*** (**delete**). При удалении найти удаляемый ключ нетрудно (**search**), но заменить его на *nil* нельзя, так как **search** перестанет работать (дойдет до этого *nil* и остановится). Поэтому вводят *nil′* (для **insert** он *nil*, для **search** – нет. Вообще это не здорово, так как время поиска из-за этих *nil′* увеличивается.

14.3. Хеш-функции для открытой адресации.

14.3.1. Линейная последовательность проб. Пусть

$hash'\colon U \to \{0, 1, \ldots, m-1\}$

обычная хеш-функция. Функция

$$hash(k, i) = (hash'(k) + i) \bmod m$$

определяет *линейную последовательность проб*.

При линейной последовательности проб начинают с ячейки $Table[h'(k)]$, а потом перебирают ячейки таблицы подряд: $Table[h'(k) + 1]$, $Table[h'(k) + 2]$, … (после $Table[m-1]$ переходят к $Table[0]$). Ясно, что последовательность проб полностью определяется первой ячейкой ($Table[h'(k)]$). Поэтому реально существует всего лишь $m$ различных последовательностей проб.

Серьезный недостаток – тенденция к образованию *кластеров* (длинных последовательностей занятых ячеек, идущих подряд), что удлиняет поиск: если в таблице из  ячеек все ячейки с четными номерами заняты, а ячейки с нечетными номерами – свободны, то среднее число проб при поиске элемента отсутствующего в таблице равно 1,5. Если же те же $m/2$ занятых ячеек идут подряд, то согласно оценке для последовательного поиска, среднее число проб равно $(m/2)/2 = m/4$ (гораздо больше). Тенденция к образованию кластеров объясняется тем, что если $k$ заполненных ячеек идут подряд, то вероятность того, что при очередной вставке в таблицу будет использована ячейка, непосредственно следующая за ними, есть $(k+1)/m$ (она пропорциональна «толщине слоя»), а вероятность использования ячейки, предшественница которой тоже свободна, всего лишь $1/m$. Таким образом, хеширование с использованием линейной последовательности проб далеко не равномерное.

14.3.2. 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 $Table[h'(k)]$, а потом ячейки просматриваются не подряд, а по более сложному закону. Метод работает значительно лучше, чем линейный. Чтобы при просмотре таблицы *Table* использовались все ее ячейки, значения $m$, $c_1$ и $c_2$ следует брать не произвольными, а подбирать специально. Следующий алгоритм обеспечивает хеширование с квадратичной последовательностью проб:

1) находим $i \leftarrow hash'(k)$; полагаем $j \leftarrow 0$;

2) проверяем $Table[i]$; если она свободна, заносим в нее запись и выходим из алгоритма, если нет – переходим к шагу 3).

3) полагаем $j \leftarrow (j+1) \bmod m$, $i \leftarrow (i+j) \bmod m$ и возвращаемся к 2).

14.3.3. Двойное хеширование – один из лучших методов открытой адресации.

$$hash(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$$

где $h_1(k)$ и $h_2(k)$ – обычные хеш-функции.

14.4. 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 ≤ 1,387, а при 90% – ≤ 2,559. При поиске отсутствующего элемента и при добавлении нового элемента оценка среднего числа проб $\frac{1}{1-\alpha}$.