Курс «Алгоритмы и алгоритмические языки»

Лекция 21

Хеш-таблицы

- ***Хеширование*** позволяет обеспечить среднее время операций с данными $T_{ср}(n) = O(1)$ 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: $U \to \{0, 1, 2, \ldots, m - 1\}$ – *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

- 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 двунаправленных 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-функции: программы

```
#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];
```

Хеш-функции: программы

```
#define MAX 2000  /* размер хеш-таблицы */

static inline void hash (int key) {
  return key % 701;
}
/* 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 ((p = index[h]) != NULL)
    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)
    return NULL;
  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
   предшествующий элемент либо NULL, если цепочка становится пустой */
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»:

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

- 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 **`NULL`**, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку).

Хеширование с открытой адресацией: программы

```
struct htype {
  int key;          /* ключ */
  int val;          /* значение элемента данных */
} *p, *index[MAX];

/* 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(el) ((el) || (el) == 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(el) ((el) || (el) == 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 (!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;
}
```

Хеш-функции для открытой адресации

- *Линейная последовательность* проб.
  Пусть $hash'$: $U \to \{0, 1, \ldots, m - 1\}$ – обычная хеш-функция.
  Функция $hash(k, i) = (hash'(k) + i) \bmod m$
  определяет *линейную последовательность проб*.
- При линейной последовательности проб начинают с ячейки *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:
  - 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.

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

  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