Лекция 13 Хеш-таблицы

13.1. 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.

13.1.1. ***Прямая адресация***. Применяется, когда количество возможных ключей невелико: например, ключи перенумерованы целыми числами из множества $U = \{0, 1, 2, \ldots, m - 1\}$, где m не очень большое целое число. В случае прямой адресации ключ с номером k соответствует элементу **`Index[k]`**. Нередко в качестве ключа используется индекс (номер элемента) массива **`Index`**, а значениями элементов массива **`Index`** являются указатели на соответствующие данные, либо нулевой указатель *null*.

Для реализации прямой адресации необходимо запрограммировать три *словарные операции*: поиск (*Search*(*key*)), добавление данных (*Insert*(*key*, *value*)) и исключение данных (*Delete*(*key*, *value*)). Ясно, что выполнение каждой из трех словарных операций требует время порядка $O(1)$ (т.е. какое-то фиксированное время): *Search*(*key*) – это обращение к элементу *Index*[*key*], *Insert*(*key*, *value*) сводится к операции присваивания *Index*[*key*] = &*value*, а *Delete*(*key*, *value*) – к операции присваивания *Index*[*key*] = *null*.

13.1.2. ***Определение***. $f(n) = O(g(n))$ означает, что с ростом n (при $n \to \infty$) отношение $f(n)/g(n)$ остается ограниченным. Если же $\lim\limits_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$, то $f(n) = o(g(n))$.

$f(n) = O(1)$ означает, что $f(n)$ является ограниченной функцией для всех n, т.е. для всех n $|f(n)| \le C$, это C можно использовать в качестве значения $f(n)$.

13.1.3. Основной недостаток прямой адресации – таблица *Index* занимает слишком много места, если множество всевозможных ключей U достаточно велико (m большое целое число). Тем более что обычно число используемых ключей << числа всевозможных ключей, так что большая часть памяти расходуется зря.

13.2. ***Хеширование*** тоже позволяет обеспечить среднее время операций с данными $T_{ср}(n) = O(1)$ и тоже за счет использования таблицы *Index*. Однако в этом случае таблица содержит места не для всех возможных ключей. Будет показано, что хеш-таблица использует намного меньшую память, чем $|U|$ (мощность множества всех ключей). Она 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 (при ее составлении ключами являются идентификаторы).

13.2.1. При прямой адресации элементу с ключом *key* отводится строка таблицы с номером *key*, 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*.

13.2.2. Если хеш-значения ключей key_1 и key_2 совпадают ($hash(key_1) == hash(key_2)$), говорят, что случилась *коллизия*. Конечно, хотелось бы выбрать хеш-функцию, для которой коллизии исключены, но это возможно лишь тогда, когда все возможные значения ключей заранее известны. В общем же случае коллизии неизбежны, так как $|U| > m$.

13.2.3. 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. Будет показано, что в случае использования двустороннего списка, среднее время выполнения каждой из трех словарных операций – поиска (*Chained-Hash-Search*(*key*)), добавления данных (*Chained-Hash-Insert*(*key*, *value*)) и исключения данных (*Chained-Hash-Delete*(*key*, *value*)) – 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.

13.2.4. Примером хеш-таблицы с цепочками является записная книжка с алфавитом. Хеш-таблица с цепочками успешно применяется в компиляторах при формировании таблицы имен.

13.3. Устройство простой хеш-таблицы (реализация хеширования с цепочками).

13.3.1. Задается некоторое фиксированное число *m* (Типичные значения *m* от 100 до 1,000,000).

13.3.2. Создается массив **`Index[m]`** указателей начал двунаправленных списков (цепочек), который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения *null*.

13.3.3. Задается хеш-функция *hash*(), которая получает на вход ключи и выдает значение от 0 до $m - 1$.

13.3.4. При добавлении пары (*key*, *value*) (*Insert*(*key*, *value*)) вычисляется $h = hash(key)$ и пара добавляется в список **`*Index[h]`**: выполняется поиск по этому списку и если в нем имеется пара с ключом *key*, значение заменяется на *value*; если пары с ключом *key* в списке нет, то пара (*key*, *value*) добавляется в конец списка.

13.3.5. При удалении, либо поиске пары (*key*, *value*) (*Delete*(*key*, *value*), либо *Search*(*key*)) вычисляется $h = hash(key)$ и происходит удаление, либо поиск пары (*key*, *value*) в списке **`*Index[h]`**.

13.4. Анализ хеширования с цепочками.

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

13.4.1.1. Коэффициент заполнения α позволяет судить о качестве хеш-функции: пусть $M = \frac{1}{m}\sum_{i=0}^{m} |\,*Index[i]\,|$ – средняя длина списков; если *hash*(*key*) – «хорошая» хеш-функция, то дисперсия $D = \frac{1}{m}\sum_{i=0}^{m-1} \left(M - |\,*Index[i]\,|\right)^2 \le \alpha$.

13.4.1.2. В частности, условие (13.4.1.1) исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список и поиск в этом списке из *n* элементов имеет среднее время $\Theta(n)$. Ясно, что такое хеширование бессмысленно.

13.4.1.3. *Равномерное хеширование*: хеш-функция подобрана таким образом, что каждый данный элемент может попасть в любую из *m* позиций хеш-таблицы с равной вероятностью, независимо от того, куда попали другие элементы.

13.4.1.4. В случае равномерного хеширования условие (13.4.1.1) выполняется и *средняя длина каждого из m списков хеш-таблицы с коэффициентом заполнения α равна α.*

13.4.1.5. В случае равномерного хеширования среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка α, так как поиск сводится к просмотру одного из списков, среднее время просмотра которого равно α.

13.4.1.6. Поскольку среднее время вычисления хеш-функции равно $\Theta(1)$, то среднее время выполнения каждой из словарных операций с учетом вычисления хеш-функции равно $\Theta(1 + \alpha)$.

Таким образом, доказана следующая

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, включая время на вычисление хеш-функции, будет равно* $\Theta(1 + \alpha)$.

13.4.2. ***Теорема***. *При равномерном хешировании среднее время успешного поиска в хеш-таблице с коэффициентом заполнения α есть* $\Theta(1 + \alpha)$.

13.4.2.1. ***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.

13.4.2.2. Доказательство несложное, но требует знания основ теории вероятностей (это видно и из п°. 13.4.2.1).

13.4.3. Из теорем 13.4.1 и 13.4.2. следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть $O(1)$, т.е. оценивается некоторым постоянным (своим для каждой операции) значением.

13.5. Методы построения хеш-функций.

13.5.1. Общие требования

13.5.2. Построение хеш-функции ***методом деления с остатком***. Хеш-функция *hash*(*key*) определяется соотношением

$$\boldsymbol{hash(key) = key \ \% \ m}.$$

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*.

13.5.3. Построение хеш-функции ***методом умножения***. Пусть количество хеш-значений равно *m*. Выберем и зафиксируем вещественную константу v $0 < v < 1$; положим

$$\boldsymbol{hash(key) = \lfloor m(key \cdot v \ \% \ 1) \rfloor}$$

$key \cdot v \ \% \ 1$ – 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. Тогда, 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:
В качестве значения *hash*(*key*) возьмем первые p битов r_0 (умножение на $m = 2^p$ и отбрасывание младших разрядов). Согласно Д. Кнуту (Искусство программирования, том 3) выбор $v = \left(\sqrt{5} - 1\right)/2 = 0.6180339887...$ является удачным.

13.6. 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;
struct htype *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 указатель на него, если нет возвращаем NULL */

struct htype *search(int k) {
  int h;
  struct htype *p;

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

  /* поиск ключа k */
  if(index[h] != NULL) {
     p = index[h];
     do {
        if(p -> key == k) return p;
        else p = p -> next;
     } while (p != NULL);
     return NULL;
}

/* 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;
}
```

13.7. Дополнения.

Insert(*key*, *value*)
Delete(*key*, *value*)