Задача

Программа hash_q.cpp создает хэш-таблицу (см. задачу в конце теории) и печатает элементы хеша. Необходимо:

1. Разобраться в ее работе.

2. Посмотреть, что как отработает программа если на вход ей подать список из 21 человек, а размер хэша сделать 21 и 30. Объяснить результаты работы программы.

3. Написать функцию выборки элемента из хеша.

```
char* ChoiceHash(char fio[])
// fio — массив в котором передается ключ (фамилия и имя)

return // информация о телефоне, который соответствует данному ключу (fio), если элемента нет, то пусто
```

4. Написать функцию удаления элемента из хеша.

```
int DelElem (char fio[])
// fio — массив в котором передается ключ (фамилия и имя) */

return // 1 — элемент удален; 0 — элемента нет (не удален)
```

Хеш (ассоциативные массивы)

Работа с линейными списками в программировании реализуются при помощи двух способов: последовательное и связное представление. Выбор способа представления зависти от операций, которые используются для решения поставленной задачи. Однако, часто возникают задачи (операции) с линейными списками, когда по значению какого-либо полю элемента списка необходимо получить всю информацию об элементе. Как не трудно видеть, и последовательное и связное представление, в этом случае потребует производить поиск данного элемента в списке. Приведем два примера:

1. В интернете, адресация к тому либо другому ресурсу производится при помощи IP-адресов, состоящий из 4 групп, кажадя из которой содержащит до 3 цифр. Естественно, запоминать 12 цифр человеку трудно, поэтому каждому такому IP-адресу ставится в соответствие символическое имя (хост): так имени сайта Санкт-Петербургского государственного университета spbu.ru — соотвествует IP-адрес 195.70.197.41. Таким образом, здесь имеется линейный список, каждый элемент которого содержит IP-адрес и его символическое имя. Основная операция при работе с данным списком будет получения по имени соответствующего IP-адреса и на оборот, по IP-адресу символическое имя.

2. Пусть имеется список студентов университета. Элемент списка содержит:

1. ФИО.
2. Дата рождения.
3. Номер курса.
4. Адрес прописки.

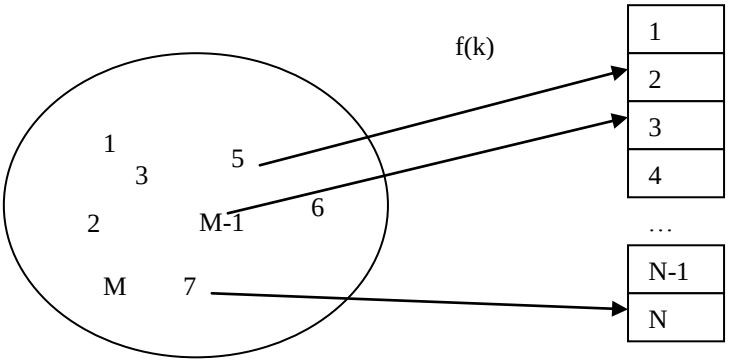
и т.д.

Естественно, возникают задачи, когда по ФИО (предполагаем, что совпадений нет) нужно получить всю информацию о студенте.

Для решения такого типа задач используются, так называемые ассоциативные массивы. Индексами в таких массивах выступает значение поля элемента списка, и они называются ключами. Предполагается, что не могут быть два элемента с одинаковыми ключами. Так, если для рассматриваемого линейного списка определить ассоциативный массив с именем CatalogStudent, то для получения информации о студенте «Сидоров Иван Иванович» необходимо будет написать: CatalogStudent(«Сидоров Иван Иванович»).

Безусловно, использование таких «ассоциативных массивов» (Д.Кнут называет их «памятью с содержательной адресацией»), весьма удобно, но как эффективно реализовать работу с такой структурой? Поскольку, ключи являются уникальными, то естественно и их кодирование в двоичном виде — уникально, следовательно, можно пробовать использовать обычные одномерные массивы для реализации ассоциативных массивы. Индексами в таких одномерных массивах выступали бы ключи, т.к. любой набор нулей и единиц можно трактовать как целое число. Однако даже на компьютере с очень большой оперативной памятью, в общем случае, нельзя разместить небольшой ассоциативный массив, если использовать для его представления обычный одномерный массив. Например в приведенном, для массива с ключами, являющимися именами студентов, имя «Сидоров Иван Иванович» (не самое длинное!) потребует 21 байт для кодирования, и следовательно размерность массива должна будет превосходить $(2^8)^{21} \sim 10^{50}$! А студентов, на 2011 в Санкт-Петербургском государственном университете в 2011 году около 32 тысяч.

Очевидно, что память следует использовать более эффективно и существует различные реализации ассоциативных массивов. Одна из наиболее известных реализаций получила название хеш:



Предположим, что мы нашли функцию i = f(k), отражающую множество всех значений ключей k в отрезок натурального ряда [1,N], такой, что N ≤ M, где M — полное количество ключей. Тогда по

произвольному ключу k мы найдем малое целое значение i, которое может быть использовано в качестве индекса, и, следовательно, все данные можно таким образом разместить в массиве небольшой размерности N и далее работать с его элементами. Такой метод называется *хешированием*, а функция f — *хеш-функцией*.

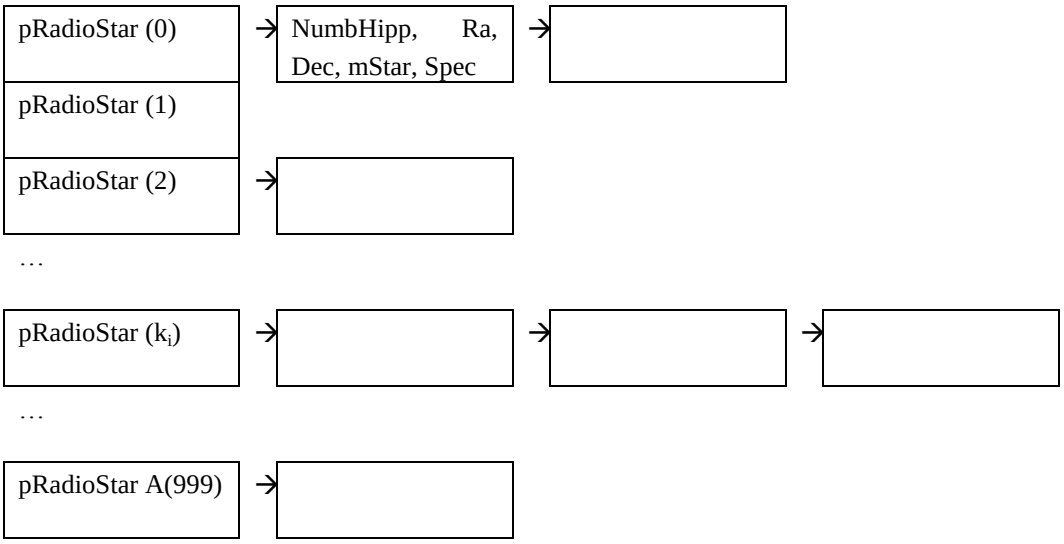
На практике будут встречаться случаи, когда разным значениям ключей будут соответствовать одинаковые значения функции. Такая ситуация называется *коллизией*, и существуют стандартные способы ее разрешения (например, элемент хеша может быть сделан головой списка записей, содержащих такие данные).

Разберем, на примере, как создать хеш:

Пусть необходимо выбрать из каталога Hipparcos (118 218 звезд) информацию (номер звезды, звездная величина, склонение, прямое восхождение, спектр) о 1000 радио звезд и иметь возможность получать информацию, в этой выборке, по номеру звезды каталога Hipparcos.

Введем следующую структуру:

```
RadioStar *pRadioStar[1000]; // массив указателей
struct RadioStar {
  int NumbHipp;
  double Ra, Dec;
  float mStar;
  char Spec[10];
}
```



Индекс массива pRadioStar должен быть связан с номером звезды из каталога Hipparcos, т.е. необходимо задать некую функцию (f), которая сопоставила бы номер радиозвезды из каталога Hipparcos индексу в массиве. Для взаимооднозначного соответствия размерность массива должна была быть равна кол-ву звезд каталога Hipparcos, что привело бы к нерациональному использованию оперативной памяти машины.

В данном случае будем вычислять индекс массива, как остаток от деления номера звезды из каталога Hipparcos на 1000. Тогда будут получаться целые числа от 0 до 999. Естественно, есть большая вероятность, что для некоторых радиозвезд остатки от деления их номеров звезд из каталога Hipparcos на 1000 будут равны. В этих случаях говорят, что возникла коллизия и необходим механизм его разрешения. В нашем случае, решение производится при помощи линейного списка типа стек.

Выбор хеш функции

От выбора хеш функции, зависит время доступа к элементу. На практике наиболее часто встречаются следующие три метода задания хеш-функции:

а) Метод деления;

б) Метод умножения;

в) Универсальный метод.

а) Метод деления

В данном методе при использовании открытой адресации ключ делится на размер хеша, а при использовании метода цепочки на размер массива указателей (A) и в качестве ключа берется остаток от деления:

$$F(k)=k \bmod m$$

Рекомендуется m (размер хеш таблицы или массива) брать простым числом и далеко отстоящем от степени двойки.

б) Метод умножения

В данном методе хеш-функция задается следующим образом:

$$F(k)=[m*(k*A \bmod 1)],$$

где $0<A<1$.

В качестве A рекомендуется брать число $(\sqrt{5}-1)/2 \approx 0{,}618033987$, а m рекомендовано задавать равным степени 2.

Для номера звезды 101 087, A=0,618033987… и m=1000 вычислим ключ:

$F(101\ 087)=[1000*(101\ 087*0{,}618033987 \bmod 1)]=204$

в) Универсальный метод

Два предыдущих метода хеширования имеют следующий недостаток: зная функцию хеширования, или случайно, можно задать так данные, что выборка элемента будет производиться за линейное время, т.е. все элементы хеша будут иметь одинаковые значения ключей.

Универсальный метод предолевает этот недостаток, выбирая хеш функцию случайным образом. В качестве одной из таких функций можно использовать следующую:

$$F(X) = \sum_{i=0}^{n} (a_i \cdot x_i) \bmod m$$

где X — ключ, с последовательностью бит $x_1, x_2, ..., x_n$;

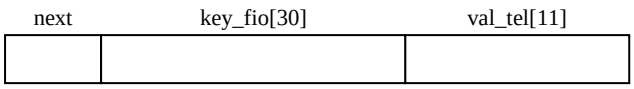
a_i — случайные числа: $0 <= a_i <= m-1$

Число m рекомендуется выбирать простым.

ЗАДАЧА

Рассмотрим реализацию хэша с прямой адресацией более подробно.
Хэш представлен массивом *hash[NN], где, скажем, NN=149 (простое число).
Элементы массива – указатели на структуру record, имеющую 3 поля:

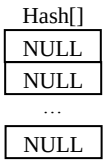
*next — указатель на элемент структуры record;
key_fio[30] — строка из 30 символов — ключ
(ключ определяет индекс массива);
val_tel[11] — строка из 11 символов (значение).



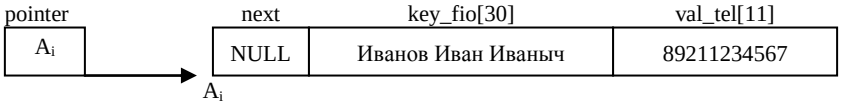
В записях хранятся пары (ключ, значение). Пусть ключи — строки длиной 30 (фамилия и имя), значения — строки длиной 11 (номер мобильного телефона).

Алгоритм

```
#define NN=30011 // простое число
#define N 30000 // кол-во элементов хэша
// ******* Описание структуры для элемента хэша *******
struct record {
    char key_fio[30];  // строки длиной 30 (фамилия и имя)
    char val_tel[11];  // строки длиной 11 (номер мобильного телефона).
    struct record *next;
 };
// ******************************************************

static struct record *hash[NN]; // Хэш представлен массивом указателей

// ******* Описание рабочих перменных  *******
int i, j;
struct record *pointer, *mpointer;
// ******************************************************
```

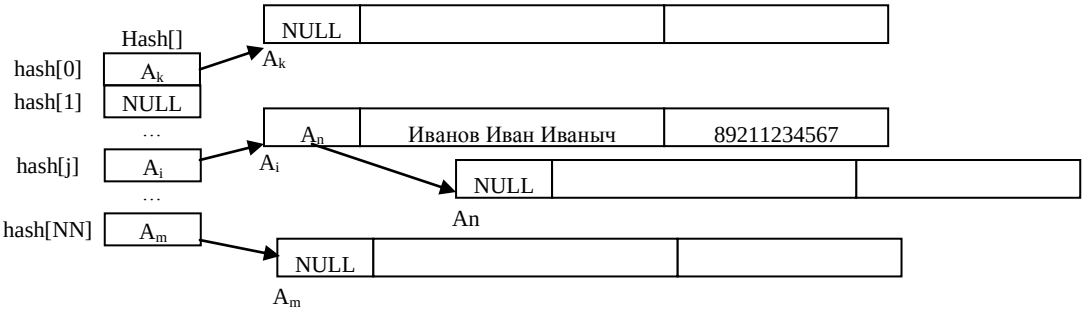



```
int main()
 {
   for (i=0; i<NN; i++) hash[i] = NULL; // Инициализируем хэш (пустой хэш)

// Цикл ввода данных и занесение их в хэш
   for (i=0; i<N; i++)  {
         pointer = malloc (sizeof(struct record)); // выделяем память под элемент списка
         (*pointer).next = NULL;
         scanf ("%s %s", &((*pointer).key_fio), &((*pointer).val_tel)); // ввод данных (фамилия, телефон)
```




```
// по ключу (фамилии и имени получаем) получаем индекс массива hash
    j = keyfunc ((*pointer).key_fio); // keyfunc — функция хеширования выдает число от 0 до NN-1

// заносим информацию в хеш
    if (hash[j] == NULL)
      hash[j] = pointer;
```


```
    else
// если элемент хеша (массива) занят, то разрешаем коллизию, создаем линейный список
       { mpointer = hash[j];
         while ((*mpointer).next != NULL) {mpointer = (*mpointer).next;}
         (*mpointer).next = pointer;
       }
   }
}
```



В качестве функция хеширования (keyfunc) можно использовать следующую:

```
int keyfunc(char s[])
{
   int len = strlen(s);
   int i;
   int key_hash;
   key_hash=0;
  for (i=0;i<len-1;i++) {
    key_hash=(key_hash+(int) s[i])*256; // предполагаем, что у нас очень!!! большие целые числа
  }
  key_hash=key_hash+(int) s[len-1];
  key_hash = fmod (key_hash, NN);
  return key_hash;
}
```