Урок 8

Хеш-таблицы

Быстрый поиск и вставка с помощью хеш-таблиц.

Введение

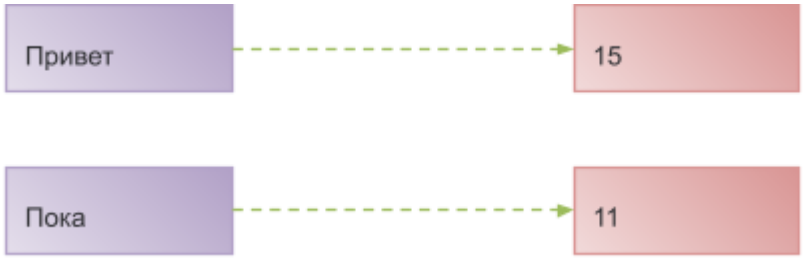
Хеш-таблицы — структура данных, которая реализует операции вставки, удаления и поиска за очень короткое время, практически **O(1)**.

На этом уроке разберемся с хеш-таблицами, хеш-функциями и рассмотрим два основных типа таблиц: цепочками и открытой адресации.

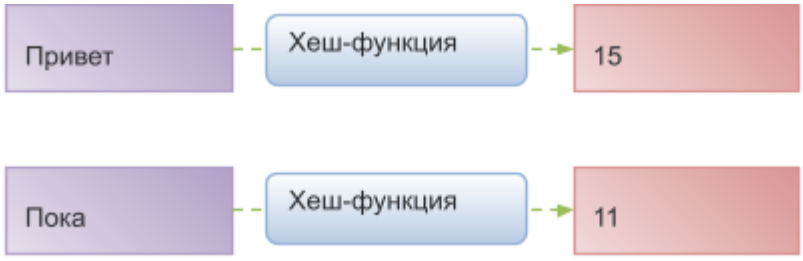
Представим, что работаем в магазине по продаже сантехники. Каждый день приходят клиенты и спрашивают о стоимости товаров. Все цены хранятся в книге, где более 50000 наименований. Если искать нужный товар линейным поиском, по времени это займет **O(n)**. Если представить, что каждое наименование товара проверяется за 0,1 секунду, то это 83 минуты. Если товары в книге будут отсортированы по алфавиту, можно воспользоваться алгоритмом бинарного поиска: тогда время поиска составит $\mathbf{O(\log_2 N)}$, или 2 секунды. Но директор требует минимизировать время на поиск и добавление товаров в книгу, так как покупателей становится все больше. Он хочет, чтобы операции поиска, вставки и удаления выполнялись за время **O(1)**. Для этого подходят хеш-таблицы.

Хеширование

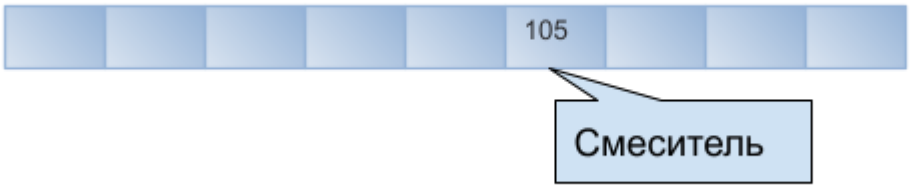
Хеширование — это преобразование массива данных или набора символов в число.



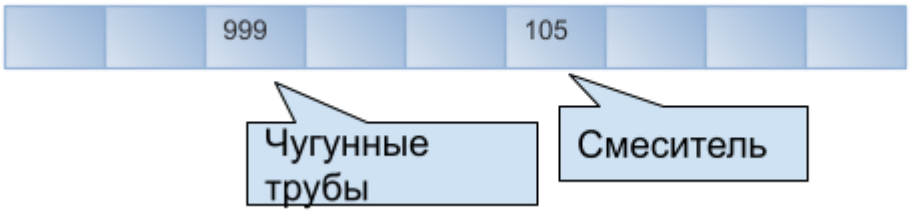
Инструмент, который преобразовывает строки в числа, называется хеш-функцией. Она работает по определенным закономерностям и не выдает случайные числа, иначе была бы бесполезной для поиска. Если передать функции слово «Привет», ответом будет 15, и в следующий раз в аналогичной ситуации — таким же. В идеале разным словам должны соответствовать разные числа. Если хеш-функция на каждое слово выдавала бы одно число, то проще было бы реализовать связанный список.



Создадим модель хеш-таблицы. В ней будем хранить цены на товары. Для хранения выберем массив. Передадим хеш-функции слово «смеситель», на выходе получим 5.



Добавим в таблицу цену на чугунные трубы. Хеш-функция вернула значение 2.



И так заполняется вся таблица.

Теперь можно работать с хеш-таблицей. Когда покупатель спросит про стоимость чугунных труб, нужно будет передать это наименование в хеш-функцию, и она вернет индекс массива, где хранится информация о цене.

Создаем хеш-функцию

Создадим свою хеш-функцию. Она должна преобразовывать строки в числа, и в идеале на каждое слово должно быть уникальное значение. Но на практике это почти невозможно.

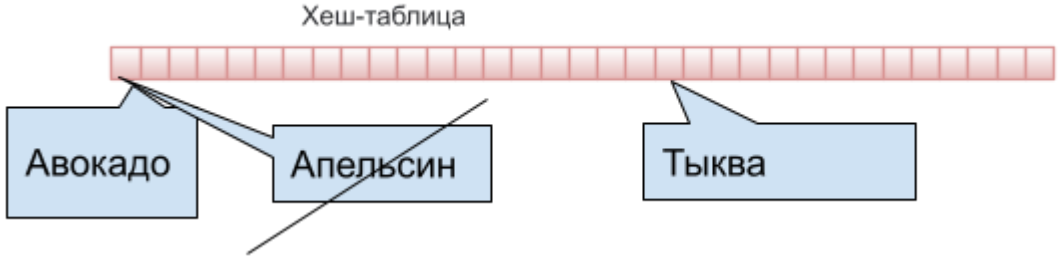
Рассмотрим несколько возможных алгоритмов хеширования. Допустим, наша хеш-таблица хранит стоимость продуктов питания. Наименования состоят из букв русского алфавита. Первое, что нужно сделать — определить размер таблицы. Выделим для нее массив на 33 элемента. Алгоритм хеширования будет генерировать индексы по алфавитному принципу, в зависимости от первой буквы слова. Для буквы «а» — индекс 0, для «б» — 1, для «в» — 2 и так далее. Добавим в таблицу цену на апельсины. Первая буква наименования — «а», так что вставляем цену на апельсины в первый элемент массива с индексом 0.

Теперь добавим в таблицу цену на тыкву.

Апельсин

Тыква

Теперь добавим авокадо. Место под букву «а» уже занято, и если добавить в первый элемент цену на авокадо, то потеряется цена на апельсин.

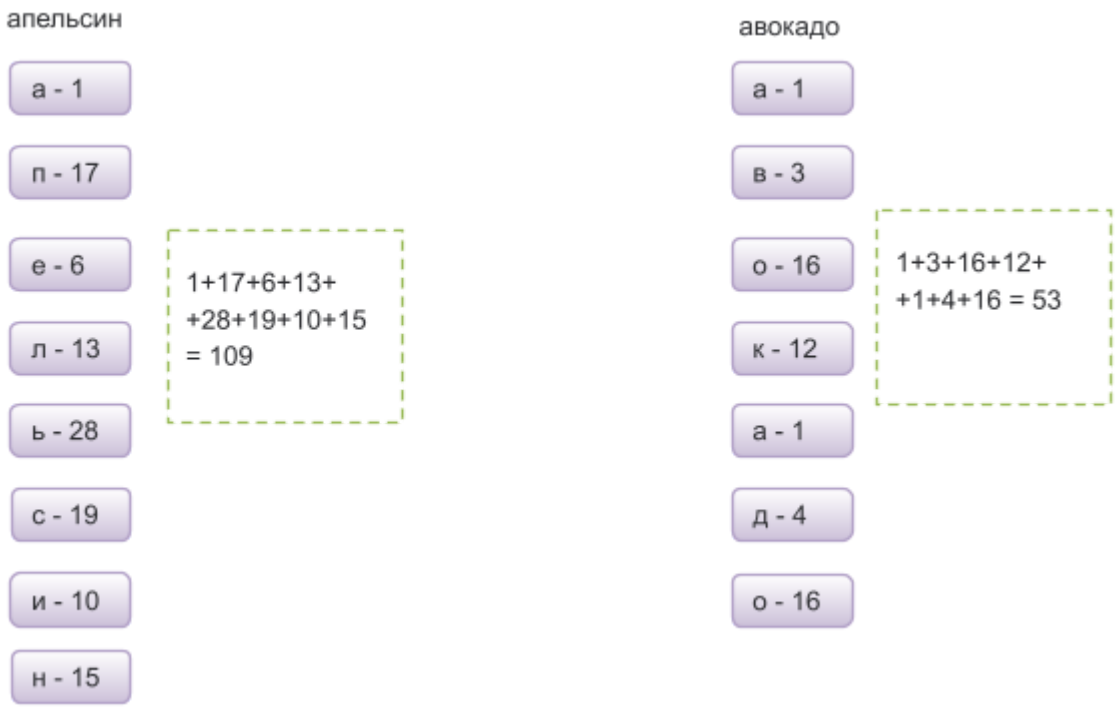


Ситуацию, когда двум ключам назначается один элемент массива, называют коллизией. Количество коллизий может быть разным, все зависит от того, насколько хорошая хеш-функция используется. Наш алгоритм, который записывает значения в массив в алфавитном порядке, — слабая хеш-функция. Существует много способов борьбы с коллизиями, и один из них — хранить в элементе массива связный список. В нашем случае это будет выглядеть так:

Если захотите узнать цену тыквы, эта операция выполнится быстро, но чтобы узнать цену на авокадо, придется выполнять дополнительный поиск внутри связанного списка. Если он небольшой, это не долго, но если в хеш-таблице все продукты будут начинаться на букву «а», то нужно менять хеш-функцию. Иначе алгоритм поиска будет такой же, как для связанного списка.

Суммирование

Рассмотрим пример: есть словарь с 50000 слов. Они ограничиваются длинной в 8 букв русского алфавита. Каждой букве присваивается числовой код: для «а» — 1, для «б» — 2, для «в» — 3. Используя такой подход, закодируем слова «апельсин» и «авокадо».



С использованием алгоритма суммирования слово «апельсин» будет помещено в 109 элемент массива, а «авокадо» — в 53.

Теоретически, последнее слово в нашем словаре будет «яяяяяяяяя». В какую позицию его поместить?

$$33 + 33 + 33 + 33 + 33 + 33 + 33 + 33 = 264$$

Получается, что размер массива, выделенный под хеш-таблицы, равен 264, а слов в словаре 50000. Разделим 50000 на 264 и получим по 189 слов на один элемент массива. Слишком много, скорость работы такой таблицы не будет оптимальной.

Возведение в степень

Попробуем еще одни алгоритм. В обычном числе, которое состоит из нескольких цифр, каждая левая цифра в десять раз больше предыдущей. Например, число 5678 можно разложить как:

$$5 * 10^3 + 6 * 10^2 + 7 * 10^1 + 8 * 10^0 = 5678$$

Так же можно поступить и со словом. Преобразовать каждую букву в число и умножить на 10 в 33 степени, так как в алфавите 33 буквы. Попробуем преобразовать слово «тыква».

$$20 * 33^4 + 29 * 33^3 + 12 * 33^2 + 3 * 33^1 + 1 * 33^0 = 23\,718\,420 + 1\,042\,173 + 13\,068 + 99 + 1 = 24\,773\,761$$

Теперь массив слишком большой и содержит слова типа «аааааааа», которые ничего не обозначают, а у нас в словаре всего 50000 слов.

Нужно оптимизировать алгоритм умножения на степени и уменьшить размер массива для хранения словаря. Если в словаре 50000 слов, и хеш-таблица должна содержать столько же элементов. На самом деле потребуется вдвое больше, но это мы разберем позже. Алгоритм сокращения размера хеш-таблицы будет основываться на вычислении остатка от деления.

Представим, что у нас есть числа в диапазоне от 0 до 199. Их необходимо поместить в хеш-таблицу размером в 10 элементов, которые пронумерованы от 0 до 9. Для расчета индекса в хеш-таблице применим оператор остатка от деления большого числа к размеру хеш-таблицы.

$$S = Ln - RangeSize$$

В формуле **s** — индекс, в который необходимо поместить число; **Ln** — число из диапазона от 0 до 199; **RangeSize** — размер хеш-таблицы.

Используя такой подход, производим сжатие с коэффициентом 20:1. Аналогичным способом можно сжимать размер хеш-таблицы для очень больших чисел, в том числе для словаря.

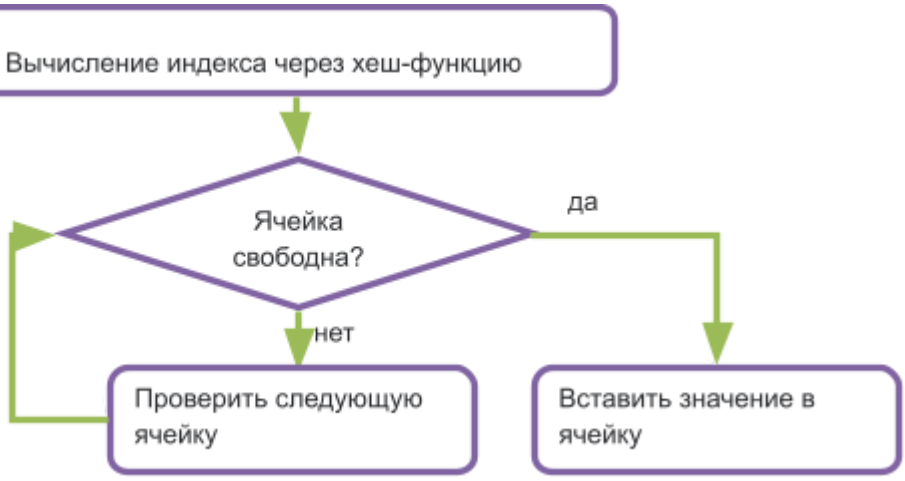
Предполагается, что на каждое слово будет приходиться две ячейки. Зачем это нужно? Вернемся к теме коллизий. Если закодировать с помощью нашего алгоритма слово melioration, то выяснится, что соответствующая ему ячейка хеш-таблицы уже занята словом demystify. Можно использовать связанный список или найти свободную ячейку — ведь для хеш-таблицы специально было выделено в два раза больше ячеек для хранения слов. Такой способ обхода коллизий называется открытой адресацией. А способ, при котором создается связный список, — методом цепочек.

Открытая адресация

Рассмотрим более детально метод открытой адресации. Существует несколько способов его реализации. Разберем линейное и квадратичное пробирование и двойное хеширование.

Линейное пробирование

Метод линейного пробирования заключается в последовательном поиске свободной ячейки. Если ячейка, которая была рассчитана с помощью хеш-функции, занята, выбирается следующая. Если и она занята, переходим к последующей. И так далее, пока не будет обнаружена свободная.



Реализуем методы поиска, вставки и удаления значений из хеш-таблицы. Чтобы вставить значение в хеш-таблицу, сначала необходимо найти ячейку. Рассмотрим поиск в хеш-таблице. В нашем примере таблица будет содержать числовые данные (ключи) от 0 до 999. Для их хранения создана хеш-таблица с размерностью в 60 ячеек. Разрабатываемая хеш-функция должна сжимать диапазон ключей до размера массива.

$$arrayIndex = key \ \% \ 60$$

```
public Item find(int key){
      int hashVal = hashFunc(key);
      while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                  return hashArr[hashVal];
            }
            ++hashVal;
            hashVal%=arrSize;
      }
      return null;

}
```

Функция **find** в начале вызывает метод **hashFunc()**, который хеширует искомый ключ для получения индекса **hashVal**.

```
public int hashFunc(int key){
      return key % arrSize;

}
```

Затем в цикле **while** проверяется, занята ли выбранная ячейка. Если ячейка не пуста, то выясняется, содержит ли она искомый ключ. Если проверка дает положительный результат, возвращается искомый элемент. В ином случае переменная **hashVal** увеличивается, и на новой итерации цикла проверяется следующая ячейка.

При достижении конца таблицы исполняется строка **hashVal%=arrSize**, которая возвращает перебор к началу таблицы.

Для вставки элемента в хеш-таблицу используется похожий механизм, как для определения позиции для вставки ключа. Но вместо конкретного значения он ищет пустую ячейку или удаленный элемент, который равен -1.

```
 public void insert(Item item){
      int key = item.getKey();
      int hashVal = hashFunc(key);
      while (hashArr[hashVal] != null && hashArr[hashVal].getKey() != -1) {
            ++hashVal;
            hashVal%=arrSize;
      }

      hashArr[hashVal] = item;

}
```

Метод **delete**, который удаляет указанное значение, похож на метод **find**. Обнаружив нужный элемент, метод изменяет его значение на -1.

```
public Item delete(int key){
      int hashVal = hashFunc(key);
      while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                  Item temp = hashArr[hashVal];
                  hashArr[hashVal] = nonItem;
                  return temp;
            }
            ++hashVal;
            hashVal%=arrSize;
      }
      return null;

}
```

Полный код программы:

```
class Item{
    private int data;

    public Item(int data){
        this.data = data;
    }

    public int getKey(){
        return this.data;
    }
}

class HashTable{
    private Item[] hashArr;
    private int arrSize;
    private Item nonItem;

    public HashTable(int size){
        this.arrSize = size;
        hashArr = new Item[arrSize];
        nonItem = new Item(-1);
    }

    public void display(){
        for(int i=0;i<arrSize;i++){
            if(hashArr[i] !=null){
                System.out.println(hashArr[i].getKey());
            } else {
                System.out.println("***");
            }
        }
    }

    public int hashFunc(int key){
        return key % arrSize;
    }

    public void insert(Item item){
        int key = item.getKey();
        int hashVal = hashFunc(key);
        while (hashArr[hashVal] != null && hashArr[hashVal].getKey() != -1) {
            ++hashVal;
            hashVal%=arrSize;
        }

        hashArr[hashVal] = item;
    }

    public Item delete(int key){
        int hashVal = hashFunc(key);
        while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                Item temp = hashArr[hashVal];
                hashArr[hashVal] = nonItem;
                return temp;
            }
            ++hashVal;
            hashVal%=arrSize;
```

```
        }
        return null;
    }

    public Item find(int key){
        int hashVal = hashFunc(key);
        while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                return hashArr[hashVal];
            }
            ++hashVal;
            hashVal%=arrSize;
        }
        return null;
    }
}

public class HashApp {

    public static void main(String[] args) throws IOException{
        Item aDataItem;
        int aKey, size, n, keysPerCell;
        // Ввод размеров
        System.out.print("Enter size of hash table: ");
        size = getInt();
        System.out.print("Enter initial number of items: ");
        n = getInt();
        keysPerCell = 10;
        // Создание таблицы
        HashTable theHashTable = new HashTable(size);
        for(int j=0; j<n; j++){
            aKey = (int)(java.lang.Math.random() * keysPerCell * size);
            aDataItem = new Item(aKey);
            theHashTable.insert(aDataItem);
        }
        while(true){
            System.out.print("Enter first letter of ");
            System.out.print("show, insert, delete, or find: ");
            char choice = getChar();
            switch(choice){
                case 's':
                    theHashTable.display();
                    break;
                case 'i':
                    System.out.print("Enter key value to insert: ");
                    aKey = getInt();
                    aDataItem = new Item(aKey);
                    theHashTable.insert(aDataItem);
                    break;
                case 'd':
                    System.out.print("Enter key value to delete: ");
                    aKey = getInt();
                    theHashTable.delete(aKey);
                    break;
                case 'f':
                    System.out.print("Enter key value to find: ");
                    aKey = getInt();
                    aDataItem = theHashTable.find(aKey);
                    if(aDataItem != null){
                        System.out.println("Found " + aKey);
```

```
                    }else
                        System.out.println("Could not find " + aKey);
                    break;
                default:
                    System.out.print("Invalid entry\n");
            }
        }
    }
    public static String getString() throws IOException{
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;
    }
    public static char getChar() throws IOException{
        String s = getString();
        return s.charAt(0);
    }
    public static int getInt() throws IOException{
        String s = getString();
        return Integer.parseInt(s);
    }


}
```

Увеличение размера хеш-таблицы

Хеш-таблица может переполниться. Чтобы этого избежать, можно увеличить размерность массива. Но в Java для этого нужно создать новый массив и скопировать в него значения, так как массивы являются статическими.

Функция хеширования учитывает размер массива, и в новом элементы будут находиться в других ячейках. Чтобы скопировать все значения в новый массив, для каждого элемента нужно выполнить метод **insert**.

Как правило, новый массив, должен быть вдвое больше предыдущего. Размер массива должен быть простым числом, и его вычисление является частью процесса перехеширования.

Создадим метод **getPrime**, который возвращает следующее простое число после текущего, которое и будет новой размерностью массива.

```
private int getPrime(int min){
      for(int i = min+1; true; i++)
            if( isPrime(i))
                  return i;
}
private boolean isPrime(int n){
      for(int j=2; (j*j <= n); j++)
            if( n % j == 0)
                  return false;
      return true;

}
```

Квадратичное пробирование

При использовании линейного пробирования элементы в хеш-таблице объединяются в группы, так как вставляются рядом. Таблица заполняется неравномерно, что снижает ее быстродействие. Чтобы этого избежать, квадратичное пробирование проверяет удаленные друг от друга ячейки.

Для этого необходимо вычислить размер шага. При линейном пробировании, если ячейка **x** занята, то следующая будет **x+1**, далее **x+2** и так далее. При квадратичном пробировании проверяется ячейка **x+1**, **x+4**, **x+9**, **x+16**, **x+25**. Расстояние от исходной позиции — квадрат номера шага.

Проблема групп при квадратичном пробировании решается, но возникает другая: последовательность, в которой ищутся свободные ячейки, одна и та же. Допустим, элементы 184, 302, 420 и 544 хешируются в индекс 7. В этом случае ключ 302 будет смещен на 1 ячейку, 420 — на четыре ячейки, а 544 — на девять. У каждого последующего элемента смещение будет еще больше. Это называется вторичной группировкой. Ее нельзя назвать серьезной проблемой, но на практике применяется более удачное решение.

Двойное хеширование

Для устранения проблем, связанных с первичной и вторичной группировкой, используется метод двойного хеширования. Создаются две хеш-функции, которые будут генерировать разную последовательность для ключей, хешируемых в один и тот же индекс.

Такая задача решается за счет повторного хеширования ключа другой хеш-функцией. Она не должна совпадать с первой функцией, а ее результат никогда не должен быть равен 0. Опыт и время показали, что для этого хорошо подходит функция такого типа:

$$\text{смещение} = \text{константа} - (\text{ключ} \ \% \ \text{константа})$$

где константа — простое число, которое меньше размера массива.

Реализация двойного хеширования:

```
class Item{
    private int data;

    public Item(int data){
        this.data = data;
    }

    public int getKey(){
        return this.data;
    }
}

class HashTable{
    private Item[] hashArr;
    private int arrSize;
    private Item nonItem;

    public HashTable(int size){
        this.arrSize = size;
        hashArr = new Item[arrSize];
        nonItem = new Item(-1);
    }

    public void display(){
```

```
        for(int i=0;i<arrSize;i++){
            if(hashArr[i] !=null){
                System.out.println(hashArr[i].getKey());
            } else {
                System.out.println("***");
            }
        }
    }

    public int hashFunc(int key){
        return key % arrSize;
    }
    public int hashFuncDouble(int key){
        return 5 - key % 5;
    }

    public void insert(Item item){
        int key = item.getKey();
        int hashVal = hashFunc(key);
        int stepSize = hashFuncDouble(key);
        while (hashArr[hashVal] != null && hashArr[hashVal].getKey() != -1) {
            hashVal+=stepSize;
            hashVal%=arrSize;
        }

        hashArr[hashVal] = item;
    }

    public Item delete(int key){
        int hashVal = hashFunc(key);
        int stepSize = hashFuncDouble(key);
        while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                Item temp = hashArr[hashVal];
                hashArr[hashVal] = nonItem;
                return temp;
            }
            hashVal+=stepSize;
            hashVal%=arrSize;
        }
        return null;
    }

    public Item find(int key){
        int hashVal = hashFunc(key);
        int stepSize = hashFuncDouble(key);
        while (hashArr[hashVal] != null) {
            if (hashArr[hashVal].getKey() == key){
                return hashArr[hashVal];
            }
            hashVal+=stepSize;
            hashVal%=arrSize;
        }
        return null;
    }

    private int getPrime(int min){
        for(int i = min+1; true; i++)
            if( isPrime(i))
                return i;
```

```
    }
    private boolean isPrime(int n){
        for(int j=2; (j*j <= n); j++)
            if( n % j == 0)
                return false;
        return true;
    }

}



public class HashApp {

    public static void main(String[] args) throws IOException{
        Item aDataItem;
        int aKey, size, n, keysPerCell;
        // Ввод размеров
        System.out.print("Enter size of hash table: ");
        size = getInt();
        System.out.print("Enter initial number of items: ");
        n = getInt();
        keysPerCell = 10;
        // Создание таблицы
        HashTable theHashTable = new HashTable(size);
        for(int j=0; j<n; j++){
            aKey = (int)(java.lang.Math.random() * keysPerCell * size);
            aDataItem = new Item(aKey);
            theHashTable.insert(aDataItem);
        }
        while(true){
            System.out.print("Enter first letter of ");
            System.out.print("show, insert, delete, or find: ");
            char choice = getChar();
            switch(choice){
                case 's':
                    theHashTable.display();
                    break;
                case 'i':
                    System.out.print("Enter key value to insert: ");
                    aKey = getInt();
                    aDataItem = new Item(aKey);
                    theHashTable.insert(aDataItem);
                    break;
                case 'd':
                    System.out.print("Enter key value to delete: ");
                    aKey = getInt();
                    theHashTable.delete(aKey);
                    break;
                case 'f':
                    System.out.print("Enter key value to find: ");
                    aKey = getInt();
                    aDataItem = theHashTable.find(aKey);
                    if(aDataItem != null){
                        System.out.println("Found " + aKey);
                    }else
                        System.out.println("Could not find " + aKey);
                    break;
                default:
                    System.out.print("Invalid entry\n");
```

```
            }
        }
    }
    public static String getString() throws IOException{
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;
    }
    public static char getChar() throws IOException{
        String s = getString();
        return s.charAt(0);
    }
    public static int getInt() throws IOException{
        String s = getString();
        return Integer.parseInt(s);
    }


}
```

Практическое задание

1. Создать программу, реализующую метод цепочек.

Дополнительные материалы

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 517–540 сс.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 487–516 сс.