НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика» (основной уровень), 2-й курс, 3-й семестр.

# Методические указания

*Тематическое занятие 22*

# *Ассоциативные массивы, хеш-таблицы.*

## Содержание

# Ассоциативный массив

## *Сложность поиска элемента*

Рассмотрим задачу поиска заданного элемента в уже имеющейся структуре данных, заполненной $n$ элементами.

Сравним асимптотические оценки временно́й сложности решения этой задачи для рассмотренных ранее структур данных.

Для массива с произвольным распределением значений элементов сложность поиска в среднем линейна $O(n)$. В отсортированном массиве сложность поиска становится логарифмической $O(\log_2 n)$, например, при использовании метода деления пополам (дихотомии).

Линейный список (как односвязный, так и двусвязный) дает линейную сложность поиска $O(n)$ вне зависимости от сортировки элементов списка.

Использование двоичного дерева поиска позволяет увеличить быстроту нахождения заданного элемента, и сложность поиска в среднем равна $O(\log_2 n)$. В произвольном дереве в худшем случае (например, когда дерево вырождается в список) сложность поиска линейна $O(n)$. Поэтому для ускорения поиска применяют балансировку деревьев, которая обеспечивает логарифмическую сложность $O(\log_2 n)$ как в среднем, так и в худшем случаях.

## *Поиск константной сложности в массиве*

Оказывается, возможно добиться такого ускорения поиска, которое приведет к константной сложности поиска элемента $O(1)$.

Пусть необходимо провести поиск заданного числа $x$ среди $n$ неповторяющихся целых чисел, значения которых известны и не выходят за пределы отрезка $[a; b]$. Причем количество всех возможных значений из этого отрезка равно $(b-a)+1=m$. Пусть оно превышает количество самих чисел $m>n$.

Пример (для небольших значений $n$ и $m$):

$n=9,\quad a=23,\quad b=35,\quad m=(b-a)+1=(35-23)+1=13,$

известные $n$ чисел из интервала $[a; b]$, расположенные в порядке возрастания:

$$\{\mathbf{24;\ 25;\ 26;\ 28;\ 29;\ 31;\ 32;\ 34;\ 35}\}.$$

Для получения быстрого решения этой задачи создадим массив из $m$ целых чисел и заполним все его элементы одинаковым значением, причем таким, которое не входит в отрезок $[a; b]$, например, значением $0$.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Теперь поместим в этот массив все имеющиеся $n$ чисел так, чтобы индексы элементов массива совпадали со значениями самих этих чисел.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 24 | 25 | 26 | 0 | 28 | 29 | 0 | 31 | 32 | 0 | 34 | 35 |

Поиск заданного числа $x$ в таком массиве выполняется в одно действие – нужно проверить содержимое ячейке с индексом, равным числу $(x-a)$. Если в ячейке находится значение $0$, то заданное число в массиве отсутствует. Если в ячейке содержится число $x$, то поиск прошел успешно.

Временна́я сложность поиска в построенной структуре данных постоянна для любого значения $n$ и равна единице.

## *Идея ассоциативного массива (словаря)*

Фактически в описанном выше примере среди пар целых чисел **(индекс, значение)** выполняется поиск по индексу. В обычном массиве значения элементов могут повторяться, но индекс всегда является уникальным.

Часто составляющими подобных пар являются не целые числа, а значения других типов, например, строки (слова в словаре), структуры (записи в телефонном справочнике) и т.д. В общем случае обе составляющие пары могут не являться целыми числами, но одно из них является уникальным (его обычно помещают на первое место в паре) и называется ***ключом***.

***Ассоциативный массив*** позволяет хранить пары **(ключ, значение)**, причем ключ является уникальным. Для конкретной пары $(key, value)$ говорят, что значение $value$ **ассоциировано** с ключом $key$. Например, для базы автомобильных номеров ключом $key$ может служить уникальный регистрационный номер автомобиля, а значением $value$ – марка автомобиля:

| Рег. номер ($key$) | Марка ($value$) |
|---|---|
| А123ВЕ77 | Ford Focus |
| Р987СТ50 | BMW X7 |
| М456НО62 | Toyota Corolla |
| К543ТХ40 | BMW X7 |

В языках программирования ассоциативный массив – это обычный массив, в котором в качестве индексов (ключей) можно использовать не только целые числа, но и значения других типов. Например, для строковых ключей:

```
array["key"] = "value";     /* Не поддерживается в языке Си. */
```

Подобные строковые ассоциативные массивы называют ***словарями***.

Для приведенного примера таблицы $(key, value)$ заполнение ассоциативного массива могло бы выглядеть так:

```
/* Не поддерживается в языке Си: */
a["А123ВЕ77"] = "Ford Focus";
a["Р987СТ50"] = "BMW X7";
a["М456НО62"] = "Toyota Corolla";
a["К543ТХ40"] = "BMW X7";
```

Некоторые языки программирования поддерживают ассоциативные массивы (или словари), но в языке Си (и Си++) в качестве **индексов массива** можно использовать **только целые числа**.

Для языков, которые не имеют встроенных средств работы с ассоциативными массивами (или словарями), существуют реализации в виде библиотек. Например, в стандартной библиотеке **STL** языка Си++ соответствующая структура данных называется `map` – отображение.

## *Операции ассоциативного массива*

В ассоциативных массивах (словарях) поддерживаются три основные операции:

- вставки (добавления) пары – insert($key$, $value$);
- поиска пары по ключу – find($key$) или search($key$);
- удаления пары по ключу – remove($key$) или delete($key$).

Эти основные операции могут дополняться другими, например, поиск пар с минимальным и максимальным значениями ключа.

# Хеш-функции и хеш-таблицы

## *Разреженный массив*

Продолжим рассматривать задачу поиска заданного числа $x$ в имеющейся совокупности из $n$ чисел. Поскольку значения целых чисел в совокупности не повторяются (т.е. уникальны), то они сами могут являться ключами $k$.

Пусть размер диапазона возможных значений этих ключей $k \in [a; b]$ существенно превышает их количество $n$: $(b-a)+1 = m \gg n$. Например:
$n = 9$, $a = 123$, $b = 64122$, $m = (b-a)+1 = 64000$,
а $n$ ключей $k$, расположенные в порядке возрастания, равны:

$\{$**296; 4137; 8419; 12372; 18159; 39265; 48652; 50294; 58437**$\}$.

Тогда для хранения незначительного количества данных придется использовать массив, подавляющее большинство ячеек которого заполнено значениями $0$:

| 0 | 1 | 2 | ... | 172 | 173 | 174 | ... | 4013 | 4014 | 4015 | ... | 63999 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **0** | **0** | **...** | **0** | **246** | **0** | **...** | **0** | **4137** | **0** | **...** | **0** |

Такой массив называют ***сильно разреженным***.

Хранение чисел в таком массиве организовано нерационально, поскольку занимает **большое количество памяти**. При таком способе хранения исходных данных оперативной памяти компьютера вообще может оказаться недостаточно.

## *Хеширование: терминология*

Попробуем разместить все $n$ чисел исходной совокупности в массиве с фиксированным небольшим количеством элементов $m$. Возьмем значение $m$ из предыдущего примера: $m = 13$.

Теперь количество элементов массива $m$ меньше, чем количество возможных значений ключей $k$. Распределение всех $n$ ключей в таком массиве называется ***хешированием***, а сам одномерный массив из $m$ элементов называется ***хеш-таблицей***.

Чтобы разместить $n = 9$ чисел в хеш-таблице (массиве) из $m = 13$ элементов необходимо для каждого ключа $k$ определить его индекс ячейки, в которую он будет помещен. Для этого вначале вычисляют значение ***хеш-адреса***, принадлежащее отрезку $[0; m-1]$. (В общем случае хеш-адрес может не совпадать с индексом ячейки хеш-таблицы, как будет показано далее.)

Функция, которая для каждого ключа $k$ вычисляет его хеш-адрес из отрезка $[0; m-1]$, называется ***хеш-функцией***: $h(k) \in [0; m-1]$.

## *Хеш-функция*

В общем случае хеш-функция должна отвечать двум (довольно противоречивым) требованиям:

- хеш-функция должна **распределять ключи** по ячейкам хеш-таблицы как можно более **равномерно**;
- хеш-функция должна **легко вычисляться**.

В рассматриваемом примере ключи $k$ являются неотрицательными целыми числами. В этом случае хеш-функция может иметь вид:

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

– остаток от деления $k$ на $m$, который всегда находится на отрезке $[0; m–1]$. Причем для более равномерного распределения ключей $m$ следует выбирать **простым** числом.

Если ключи – символы некоторого алфавита $c$, то сначала их можно пронумеровать целыми неотрицательными числами $\mathrm{ord}(c)$, а затем так же применить для их номеров операцию остаток от деления на простое число:

$$h(c) = \mathrm{ord}(c) \bmod m.$$

Если ключами являются символьные строки – последовательности из $s$ символов $c_0c_1c_2…c_{s–1}$, тогда

- в качестве хеш-функции можно использовать простейшую (но не лучшую) функцию:

$$h(c_i) = \left(\sum_{i=0}^{s-1} \mathrm{ord}(c_i)\right) \bmod m;$$

- существенно лучший вариант, когда значение хеш-функции вычисляют с помощью алгоритма вида:

```
h = 0;
for (i=0; i<=s-1; ++i)
    h = (h*C + ord(c[i]))%m;
```

где `C` – константа, бóльшая, чем любое из значений $\mathrm{ord}(c_i)$.

Для более равномерного распределения желательно, чтобы хеш-функция зависела **от всех битов ключа**, а не только от некоторых из них.

Хеш-функции могут быть и другими – программисту нужно выбрать такую хеш-функцию, которая наилучшим образом соответствует решаемой задаче и отвечает приведенным выше требованиям.

## *Коллизии*

Для рассматриваемого примера $m = 13$ – простое число, поэтому в качестве хеш-функции следует выбрать:

$$h(k) = k \bmod m.$$

Вычислим хеш-адреса (значения хеш-функции) для каждого ключа $k$ из примера:

| $k$ | $h(k)$ |
|---|---|
| 296 | 10 |
| 4 137 | 3 |
| 8 419 | 8 |

| $k$ | $h(k)$ |
|---|---|
| 12 372 | 9 |
| 18 159 | 11 |
| 39 265 | 5 |

| $k$ | $h(k)$ |
|---|---|
| 48 652 | 6 |
| 50 294 | 10 |
| 58 437 | 2 |

Теперь попытаемся разместить ключи $k$ в ячейках массива (хеш-таблицы) используя в качестве индексов полученные хеш-адреса:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 58437 | 4137 | 0 | 39265 | 48652 | 0 | 8419 | 12372 | ? | 18159 | 0 |

Для двух ключей (**296** и **50 294**) значения хеш-адресов совпадают (и равны **10**), но их невозможно разместить в одной и той же ячейке хеш-таблицы. Ситуация, когда два или несколько ключей хешируются в одну ячейку таблицы, называется ***коллизией***.

Коллизии обязательно происходят, если количество ячеек хеш-таблицы $m$ меньше, чем количество ключей $n$: $m < n$.

В **самом худшем** случае хеш-адреса всех ключей могут совпадать, то есть все ключи могут хешироваться в одну ячейку хеш-таблицы.

При удачном выборе размера хеш-таблицы $m$ и хеш-функции $h(k)$ коллизии возникают редко. Но любая схема хеширования должна иметь механизм разрешения коллизий.

# Способы хеширования

Существуют два основных способа хеширования, которые различаются механизмом разрешения коллизий:

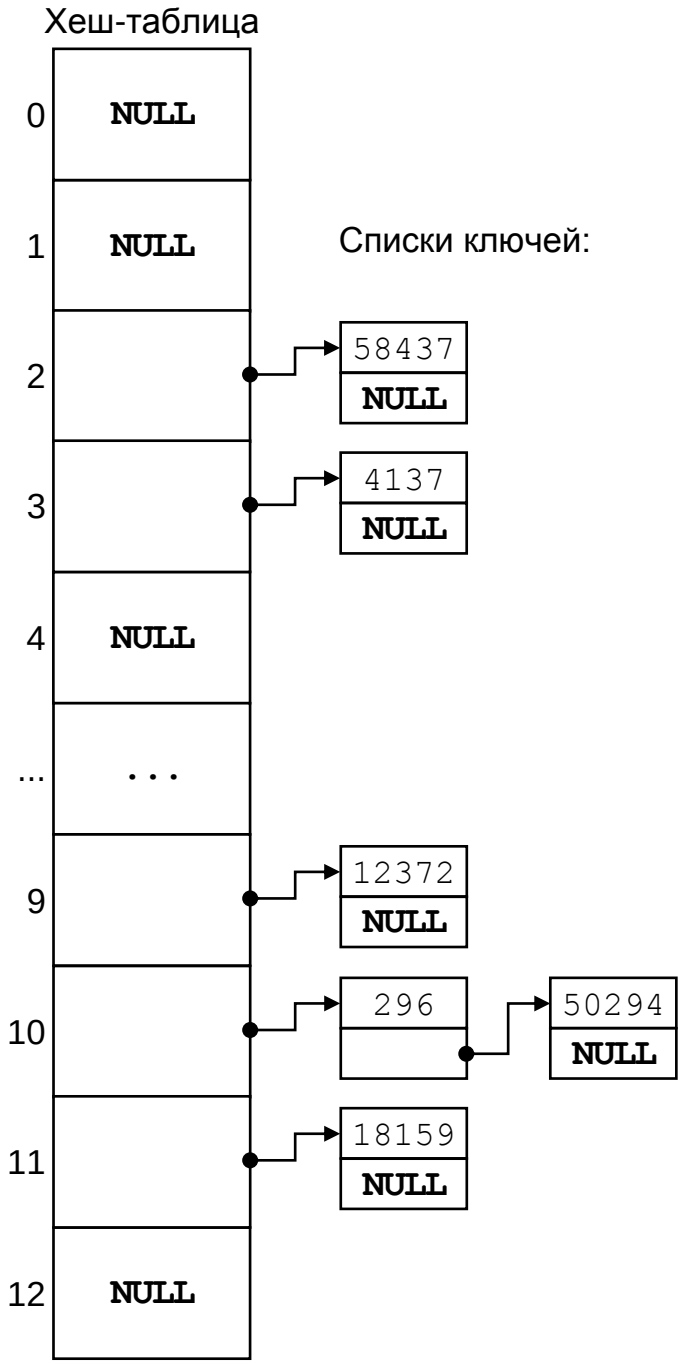
- ***открытое*** хеширование (хеширование ***с раздельными цепочками***);
- ***закрытое*** хеширование (хеширование ***с открытой адресацией***).

## *Открытое хеширование*

При открытом хешировании ключи хранятся в цепочках – **списках** (одно- или двусвязных), присоединенных к ячейкам хеш-таблицы. Каждый такой список содержит все ключи, хешированные в соответствующую ему ячейку.

При этом в каждой ячейке хеш-таблицы хранится не значение ключа, а указатель на присоединенный к ней список (или нулевой указатель, если такой список отсутствует). То есть сама хеш-таблица является **массивом указателей**.

Схема памяти открытого хеширования с односвязными списками для рассмотренного примера:




Обычно ключи помещаются в каждый список в произвольном порядке, поскольку отсортированный список не дает существенных преимуществ в скорости поиска. Вставка (добавление) нового ключа обычно делается в конец списка, но возможны и другие варианты.

Поиск заданного ключа $x$ при открытом хешировании проходит в два этапа:

1) вычисление значения хеш-функции для искомого элемента $h(x)$;
2) поиск элемента в списке, присоединенном к ячейке хеш-таблицы с хеш-адресом, равным вычисленному значению хеш-функции $h(x)$.

## *Коэффициент заполнения*

При открытом хешировании в общем случае эффективность поиска зависит от длины списков. А длина списков зависит от $n$, $m$ и качества используемой хеш-функции $h(\cdot)$.

Если хеш-функция распределяет $n$ ключей по $m$ ячейкам хеш-таблицы практически равномерно, то в каждом списке будет содержаться

$$\alpha = \frac{n}{m}$$

ключей. Это отношение называется ***коэффициентом заполнения***.

В рассмотренном примере открытого хеширования $\alpha = 9/13 \approx 0{,}69$.

Желательно, чтобы коэффициент заполнения приближался к единице. Очень малое значение коэффициента заполнения свидетельствует о множестве пустых ячеек в хеш-таблице и неэффективном использовании памяти, очень большое – длинные списки и продолжительное выполнение поиска.

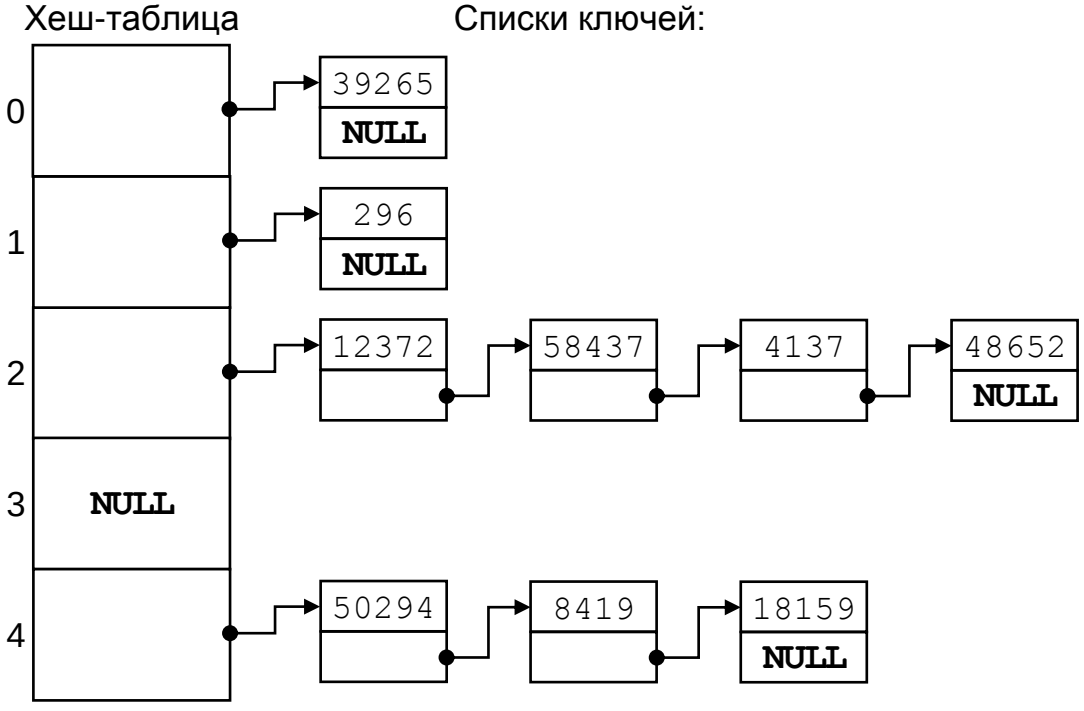
Для хеш-таблиц с раздельными цепочками при разумных значениях коэффициента заполнения временна́я сложность поиска в среднем константна $O(1)$. В самом худшем случае (при цепочке длиной $n$) сложность поиска линейна $O(n)$.

## *Выбор размера хеш-таблицы*

Существенное влияние на эффективность открытого хеширования оказывает выбранный программистом размер хеш-таблицы – значение $m$.

С одной стороны, при использовании остатка от целочисленного деления в качестве хеш-функции для равномерного распределения ключей $m$ должно являться простым числом. С другой стороны, значение $m$ должно обеспечивать близость к единице коэффициента заполнения $\alpha$.

Например, если бы в рассмотренном ранее примере было выбрано значение $m$=5, то схема памяти была бы следующая:




Ключи помещены в каждый список в произвольном порядке, например, в порядке ввода значений ключей с клавиатуры.

Хотя здесь $m$ тоже является простым числом, но распределение получается неравномерным и эффективность поиска ниже, чем при $m=13$.

В качестве значения $m$ рекомендуют выбирать простое число, **немного большее** $n$. Например, первое или второе простое число, следующее за $n$ в последовательности натуральных чисел.

## *Закрытое хеширование*

В случае закрытого хеширования все ключи хранятся в хеш-таблице, без использования списков. Поэтому такой способ хеширования возможен только при

$$m \geqslant n.$$

Для разрешения коллизий могут применяться различные правила. Например, ***линейное исследование***, когда в случае коллизии следующие ячейки проверяются одна за другой. Если следующая ячейка пуста, то ключ вносится в нее; если заполнена – проверяется ячейка, следующая за ней. Если при проверке достигается конец хеш-таблицы, то поиск переходит к первой ячейке (по принципу циклического массива).

Для рассмотренного примера, если ключи вносятся в хеш-таблицу в порядке возрастания их значений:

**..., 296, ..., 18159, ..., 50294, ...,**

то при возникновении коллизии:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 58437 | 4137 | 0 | 39265 | 48652 | 0 | 8419 | 12372 | 296 | 18159 | 50294 |

Из двух ключей с совпадающими хеш-адресами (равными **10**) первый ключ (**296**) помещаются в ячейку хеш-таблицы с индексом **10**, а второй ключ (**50 294**) – в следующую незаполненную ячейку (в которой находится значение $0$) с индексом **12** (хотя индекс ячейки не совпадает с хеш-адресом ключа).

Заполнение хеш-таблицы при линейном исследовании зависит от **порядка внесения** в нее ключей. Если в рассмотренном примере ключи вносятся в хеш-таблицу в следующем порядке:

**..., 296, ..., 50294, ..., 18159, ...,**

то возникновение коллизии приводит к тому, что ячейка хеш-таблицы с индексом **11** становится занята. Ключ **18 159** приходится внести в следующую пустую ячейку (с индексом **12**), хотя хеш-адрес этого ключа равен **11**:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 58437 | 4137 | 0 | 39265 | 48652 | 0 | 8419 | 12372 | 296 | 50294 | 18159 |

Для линейного исследования по мере заполнения хеш-таблицы эффективность поиска ухудшается. При полном заполнении хеш-таблицы, вставка ключей становится невозможной, и хеширование приходится проводить заново в новую хеш-таблицу бóльшего размера, увеличивая $m$.

# Сравнение структур данных

Сравним различные структуры данных по оценкам **временно́й сложности** выполнения двух основных операций: **поиска** и **вставки (добавления)** элементов:

| **Структура данных** | | **ПОИСК** | | **ВСТАВКА** | |
|---|---|---|---|---|---|
| | | **средняя** | **худшая** | **средняя** | **худшая** |
| **Массив** (динамический) | произвольный | $O(n)$ | | $O(n)$ | |
| | сортированный | $O(\log_2 n)$ | | | |
| **Список** (одно- или двусвязный) | произвольный | $O(n)$ | | $O(1)$ | |
| | сортированный | | | $O(n)$ | |
| **Дерево** (двоичное) | произвольное | $O(\log_2 n)$ | $O(n)$ | $O(\log_2 n)$ | $O(n)$ |
| | балансированное | $O(\log_2 n)$ | | $O(\log_2 n)$ | |
| **Хеш-таблица** | | $O(1)$ | $O(n)$ | $O(1)$ | $O(n)$ |

Хеш-таблицы позволяют быстро проводить поиск и вставку с константной сложностью $O(1)$ в среднем, но проигрывают в худших случаях балансированному двоичному дереву (Б-дереву), для которого сложность основных операций – логарифмическая $O(\log_2 n)$.