**РАЗДЕЛ 7. БАЗОВЫЕ АЛГОРИТМЫ ПОИСКА**

Поиск – это один из важнейших алгоритмов обработки структур данных. Алгоритмы поиска предназначены для проверки наличия элемента или извлечения элемента из любой структуры данных, в которой он хранится. Более строго задачу поиска можно сформулировать следующим образом: найти один или несколько элементов в множестве, причем искомые элементы должны обладать определенным свойством. Это свойство может быть абсолютным или относительным. Относительное свойство характеризует элемент по отношению к другим элементам: например, минимальный элемент в множестве чисел.

Существует огромное количество алгоритмов поиска. Их сложность варьируется от простых алгоритмов поиска методом последовательного сравнения, в чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска. Особого внимания заслуживают алгоритмов, основанные на представлении базового набора данных в другой, более подходящей для поиска форме, и которые применяются в реально действующих приложениях обработки массивов информации в огромных базах данных.

Для решения задачи поиска также как и для задач сортировки не существует единого алгоритма, который бы лучше подходил для всех случаев. Некоторые из алгоритмов выполняются быстрее других, но для их работы требуется дополнительная оперативная память. Другие выполняются очень быстро, но их можно применять только для заранее отсортированных массивов. В то же время анализ алгоритмов поиска несколько отличается от алгоритмов сортировки. В частности, для них не существует проблемы устойчивости. При этом могут возникать ситуации, требующие введения новых критериев сложности и ее оценка [intellect.icu].

В целом, все алгоритмы сводятся к выполнению следующих шагов [intellect.icu]:

1) установление свойства элементов исходного набора; в большинстве случаев такими свойствами являются значения элементов;

2) сравнение значения элемента с эталонным свойством (для абсолютных свойств) или сравнение свойств двух элементов (для относительных свойств);

3) прохождение по элементам множества.

В принципе, алгоритмы поиска отличаются между собой именно методами перебора и стратегии поиска. Исходя из типа поисковой операции, эти алгоритмы обычно классифицируются в двух категориях:

*Последовательный поиск*: при этом список или массив выполняется последовательно, и каждый элемент проверяется.

*Интервальный поиск*: Эти алгоритмы специально разработаны для поиска в отсортированных структурах данных. Эти типы поисковых алгоритмов намного эффективнее, чем линейный поиск, поскольку они многократно нацелены на центр структуры поиска и делят пространство поиска пополам.

В данном разделе рассмотрим основные алгоритмы поиска в структурах данных. Для более глубокого понимания практического использования алгоритмов в разделе приводятся их DRAKON-диаграммы и оценка временной и пространственной сложности.

**6.1. Линейный поиск данных**

а). Линейный поиск неотсортированных данных

В случае линейного поиска элемента в неупорядоченном наборе данных, например, в массиве или в срезе простейшим алгоритмом является просмотр всех элементов до тех пор пока не будет найдено искомое значение (рис.6.1).

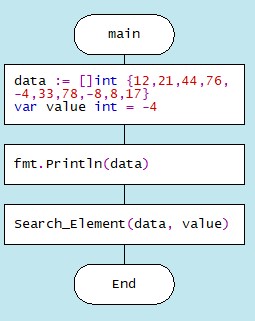
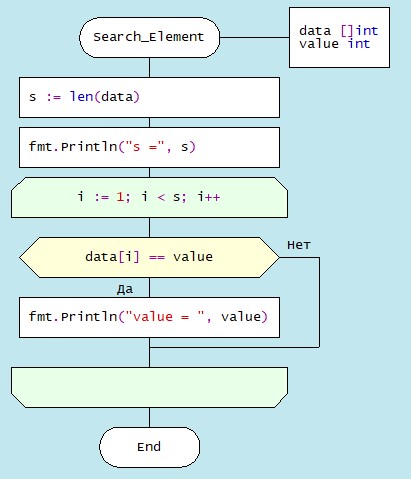
 

Fig.6.1 DRAKON-диаграмма алгоритма линейного поиска

Этот алгоритм не очень эффективен, однако он работает на произвольных коллекциях.

*Временнáя сложность*: O(n). В худшем случае (нужный элемент находится в последней позиции) для поиска элемента нужно пройти все элементы среза. Здесь «n» — размер среза. дополнительная память.В принципе возможен и другой наихудший случай – отсутствие нужного элемента.

*Пространственная сложность*: O(1). Для размещения среза дополнительной памяти не требуется.

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Search Element) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

б). Линейный поиск отсортированных данных в срезе

Если элементы набора данных отсортированы по возрастанию, либо по убыванию, то поиск нужного элемента будет намного эффективнее, чем в неупорядоченном линейном поиске. Поскольку во многих случаях не нужно проходить весь список. Например, когда в результате прохода по возрастающему отсортированному списку обнаруживается элемент с большим значением, дальнейший поиск прекращается. Такой подход экономит время и повышает производительность. На рис. 6.2. показана DRAKON-диаграмма такого алгоритма.

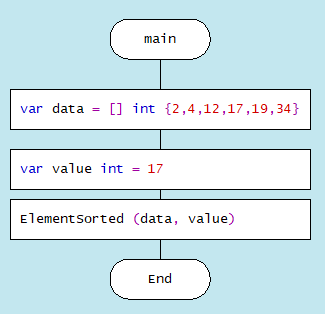
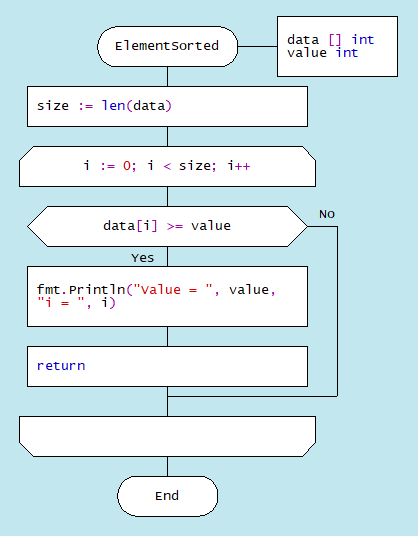
 

Рис. 6.2. DRAKON-диаграмма алгоритма поиска данного отсортированного среза

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Data Sorted) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

**6.2. Двоичный поиск данных в отсортированном срезе**

Алгоритм двоичного поиска реализуется следующим образом:

1. Определяется значение элемента в середине структуры данных. Полученное значение сравнивается с искомым значением.
2. Если искомое значение меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент с искомым значением или не станет пустым интервал для поиска.

DRAKON-диаграмма алгоритма бинарного поиска представлена на рис. 6.3. (модуль main() аналогичен предыдущему алгоритму):

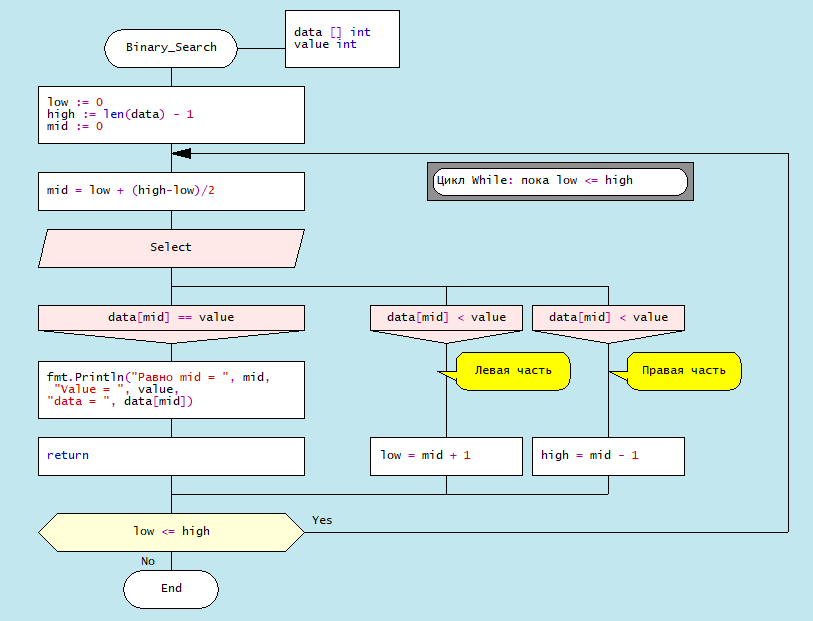


Рис. 6.3. DRAKON-диаграмма алгоритма бинарного поиска

*Временная сложность* алгоритма бинарного поиска принадлежит классу O(log n). Способ, которым следует это интерпретировать, заключается в том, что асимптотический рост времени, затрачиваемого функцией на выполнение, данного входного набора размера n, не будет превышать log n.

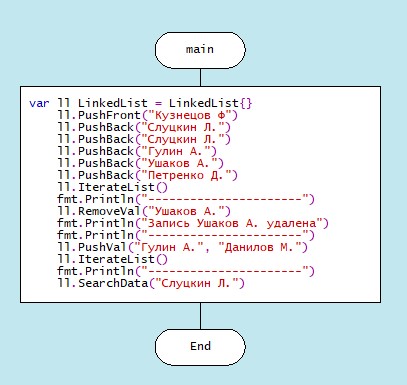
Пространственная сложность: O(1). То есть дополнительного пространства не требуется.

|  |  |
| --- | --- |
| Пути доступа к программным файлам (Binary Sorted) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

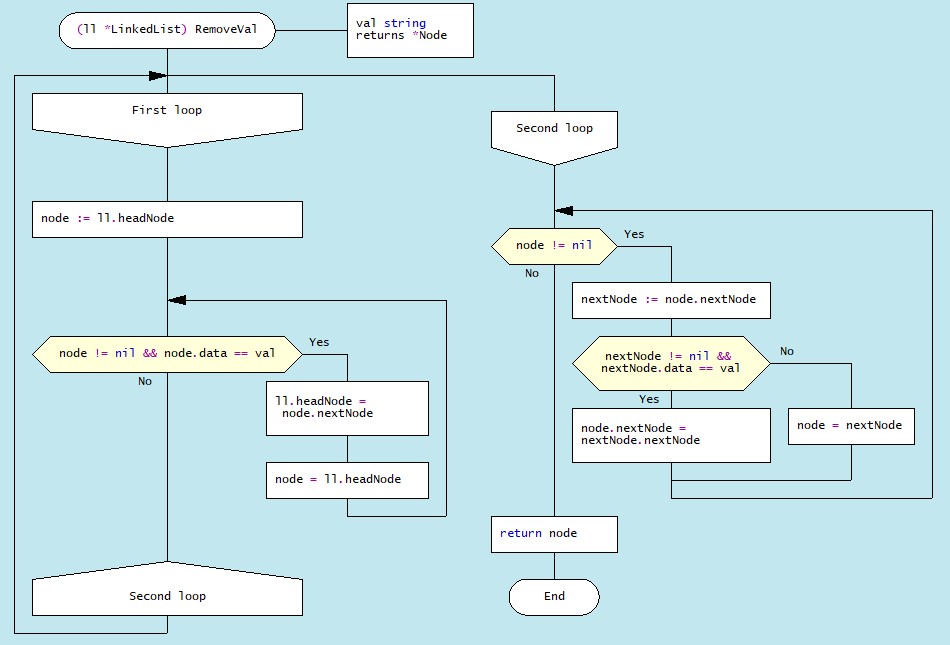
**6.3. Поиск данных в односвязном списке**

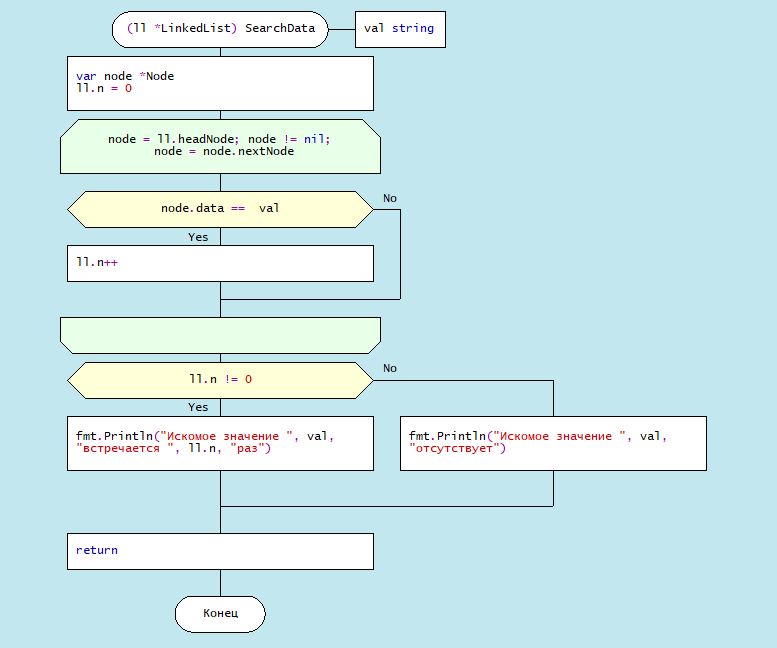
В отношении односвязного списка возможны три варианта. Во-первых, искомое значение в списке отсутствует, во-вторых, искомое значение встречается один раз и, в-третьих, искомое значение встречается неоднократно. Можно также поставить задачу удаления дубликатов, то есть лишних узлов с повторным значением.

Для решения этих задач необходимо создать односвязный список, элементы которого содержат значения «Кузнецов Ф.», «Слуцкин Л.», «Гулин А.», «Ушаков А.», «Петренко Д.». В этом списке нужно удалить запись «Ушаков А.», после чего включить новую запись «Данилов М.», разместив ее после записи «Гулин А.». После этого нужно удалить дубликаты записи «Слуцкин Л.», оставив только одну. Соответствующие DRAKON-диаграммы представлены на рис. 6.4 а.,б.,в.,г.:

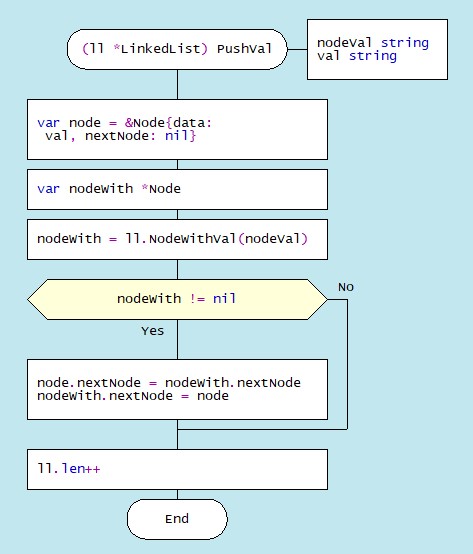
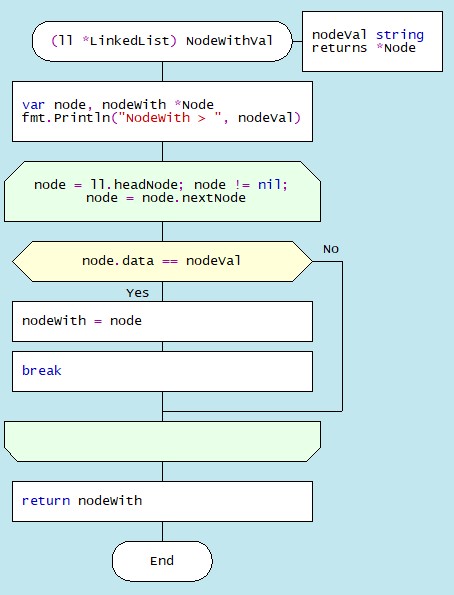


a) функция main()

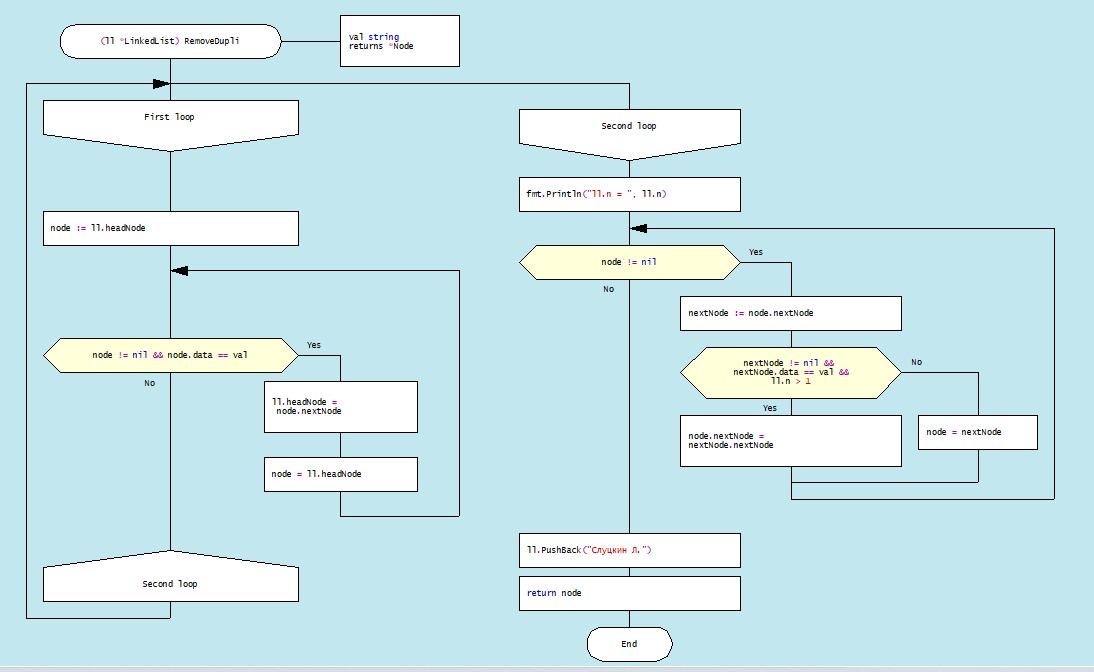
 б) функция удаления RemoveVal



в) функция поиска значения SearchData

г) функции вставки по значению PushVal + NodeWithVal



д. функция удаления дубликатов RemoveDupli

Рис. 6.4. DRAKON-диаграммы алгоритмов удаления, поиска, вставки по значению и удаление дубликатов

|  |  |
| --- | --- |
| Пути доступа к программным файлам (LinkedList Search) | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

6.4. Хеширование

Время поиска элемента в наборе данных зависит от количества сравнений значений элементов. Для сокращения времени поиска, а, следовательно, повышения эффективности вычислительного процесса, необходимо уменьшить число сопоставлений. Этого можно достичь, преобразовав набор данных большего диапазона в набор меньшего диапазона, получившего название хэширования, результатом которого является хэш-таблицы. [https://spravochnick.ru/informatika/algoritm\_poiska\_poisk\_heshirovaniem/].

С точки зрения теории абстрактных типов данных (АДТ) хэш-таблица представляет собой структуру данных, реализующую интерфейс ассоциативного массива, которая позволяет хранить пары ключ-значение и выполнять три основных операции: операцию д*обавления* новой пары, операцию *поиска* и операцию *удаления* пары ключ-значение.

**С позиций программирования хэш-таблица п**редставляет собой коллекцию элементов, содержащих пары ключ-значение, где ключ вычисляется специальной функцией, получившей название *хэш-функции*. Хэш-таблица, в свою очередь,   состоит из бакетов (bucket), наборов элементов с совпадающими или близкими значениями хэш‑функции. Существуют различные методы построения хэш-функции, простейшим из которых является *метод остатков*, где хэш-функция определяется как остаток от деления двух чисел (x,m), где x – элемент набора, m - число бакетов. В языке golang хэш-функция для этого метода имеет вид: h = x % m.

Разберем более подробно процесс создания хэш-таблицы с помощью инструментария языка Go в редакторе DRAKON WEB Editor . Вначале создается переменная типа Node, определяемая как структура, состоящая из двух полей: значение элемента - Value int и адрес следующего элемента - Next \*Node. Фактически это односвязный список (см. Раздел 1)

type Node struct {

Value int

Next \*Node

}

Затем создается хэш-таблица через структуру, имеющую два поля: первое поле (Table) представляет собой карту map, которая соотносит целое число (хэш-индекс) со связанным списком (\*Node), а второе – размер хэш-таблицы – Size int:

type HashTable struct {

Table map[int]\*Node

Size int

}

В результате, эта хэш-таблица должна будет иметь столько связанных списков (бакетов), сколько было задано константой Size. В приведенном выше случае число слотов равно 15. Заметим, что объявление типов Node и HashTable, а также константа Size вынесено в опцию File/File description (рис.6.5.).

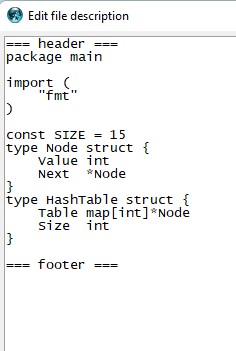


Рис. 6.5. Объявление типов Node, HashTable и константы SIZE

В качестве примера рассмотрим построение хэш-таблицы размером m = 15 для коллекции целых чисел от 0 до 120. Первоначально слоты хэш-таблицы являются пустыми (рис. 6.5.)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 11 | 12 | 13 | 14 |
| - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

Рис. 6.5. Пустая хэш-таблица из 15 слотов

**Хэш-функция, отражающая с**вязь между элементом и слотом, должна принимать любой элемент из набора данных (0 … 120) и возвращать целое число из диапазона номеров слотов (от 0 до 14). Алгоритм, реализующий метод остатков, просто берёт по очереди элемент из исходного набора и делит его на 15, возвращая остаток в качестве хэш-значения, который заносится в слот h(item)=item % 15. Например, хэш-код для элемента 119 определяется как 119 % 15 = (119 – 15\*7) = 14, значение 119 заносится в соответствующий слот (рис. 6.6.):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 11 | 12 | 13 | 14 |
| - | - | - | - | - | - | - | - | - | - | - | - | - | - | 119 |

Далее алгоритм определяет слоты для других элементов, постепенно заполняя их. Когда же алгоритм встретит в цикле элемент 104, тогда остаток от деления 104%15 = 14, следовательно этот элемент будет занесен в также в 14-ый слот. Таким образом в каждом слоте будут накапливаться соответствующие элементы, имеющие один индекс-хэш. Например для индекс-хэша, равного 8, слот будет состоять из таких элементов: 113 : 98 : 83 : 68 : 53 : 38 : 23 : 8. А вся хэш-таблица будет иметь следующий вид (табл. 6.1.):

Таблица 6.1. Хэш-таблица из 15 слотов

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 105  103 88 73 58 43 28 13 | 106  91 76 61 46 31 16 1 | 107  92 77 62 47 32 17 2 | 108  103 78 63 48 33 18  3 | 109  94 79 64 49 34 19 4 | 110  95 80 65 50 35 20 5 | 111  96 81 66 51 36 21 6 | 112  97 82 67 52 37 22  7 | 113  98 83 68 53 38 23  8 | 114  99 84 69 54 39 24 9 | 115  100  85  70  55  40  25  10 | 116  101  86  71  56  41  26  11 | 117  102 87 72 57 42 27 12 | 118  103 88 73 58 43 28 13 | 119  104 89 74 59 44 29 14 |

Таким образом, с помощью метода остатков коллекция из 120 целых чисел преобразуется в хэш-таблицу, состоящую из 15 слотов. Теперь поиск того или иного элемента значительно ускоряется, поскольку происходит в два шага: вначале по хэш-функции h =(x % m) вычисляется хэш-индекс, после чего поиск выполняется в слоте из 7 элементов. Алгоритм, основанный на методе остатков представлен следующими DRAKON-диаграммами (рис. 6.6.). Здесь hinsert – модуль заполнения элементами слотов по хэш-функции, hLookup – модуль поиска элемента в хэш-таблице, hTravers – модуль прохода по хэш-таблице.

Реализация основных функций хэш-таблицы с использованием хэш-функции выглядит следующим образом:

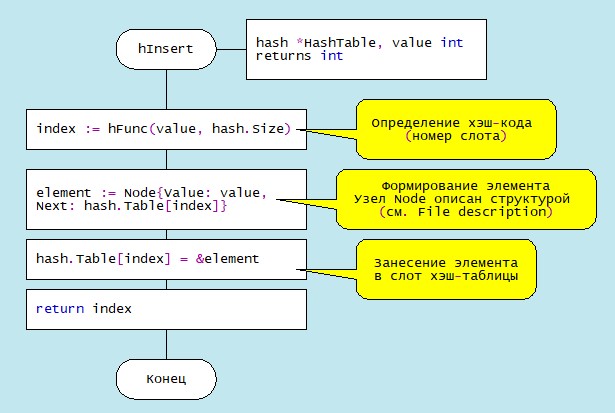
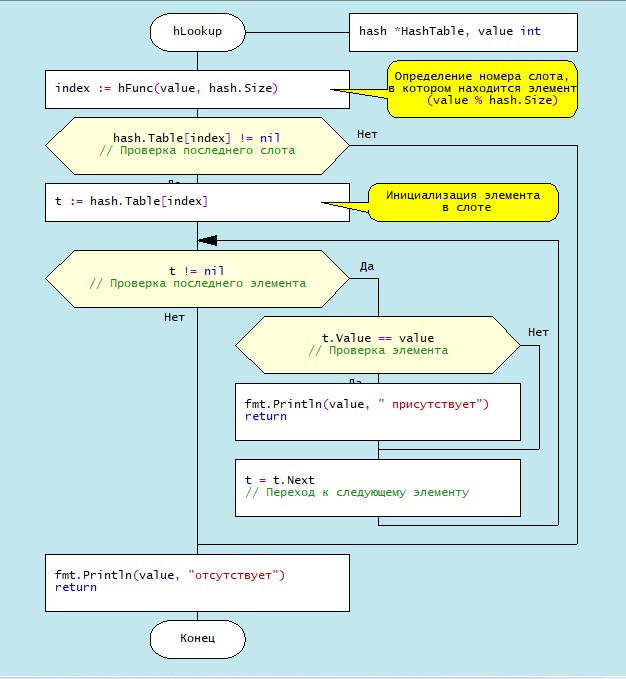
1. Создать структуру HashTable список размера m для хранения объектов.

2. Вычислить хэш-код объекта, передав его через хэш-функцию.

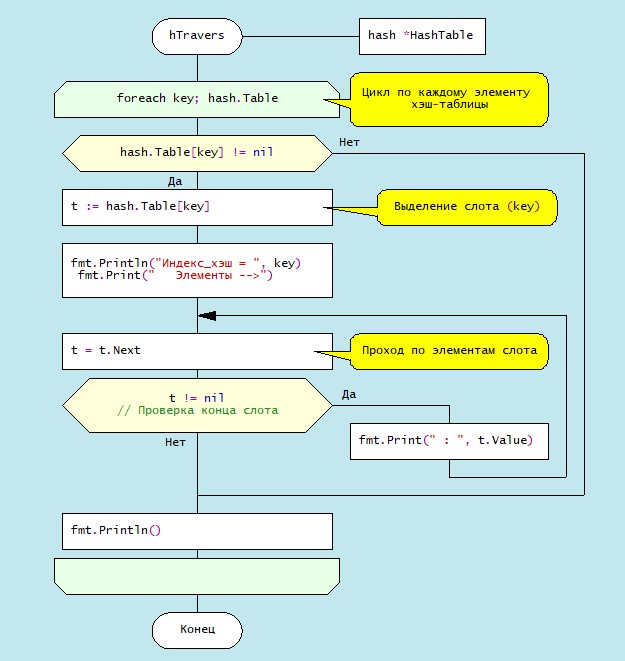
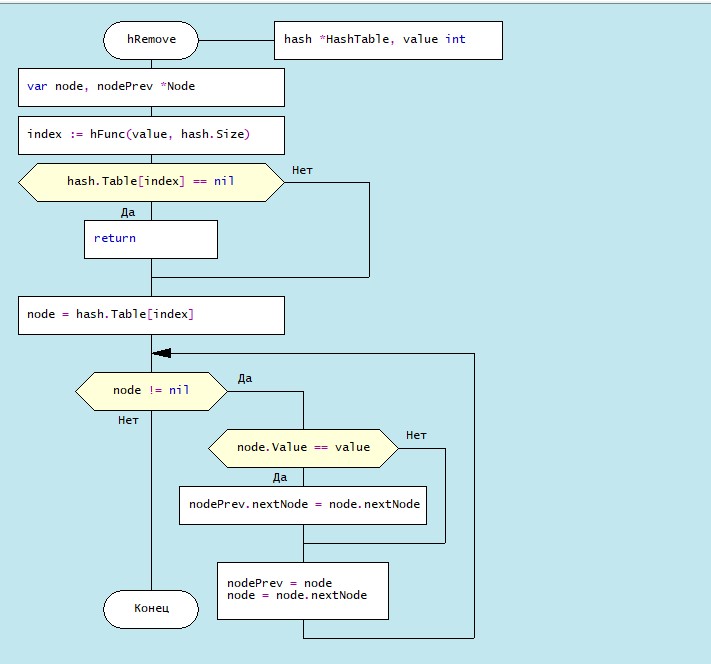
З. Получить хэш-индексы бакета, где будут сохранены объекты.

4. Сохранить эти объекты в назначенном бакете.

DRAKON-диаграммы алгоритмов реализации основных функций работы с хэш-таблицами представлены на рис. 6.6.

а) Модуль формирования хэш-таблицы б) Модуль поиска элемента

в) Модуль прохода по хэш-таблице г) Модуль удаления элемента

Рис. 6.6. DRAKON-диаграммы алгоритма работы с хэш-таблицами

Рассмотрим подробнее алгоритм удаления элемента из хэш-таблицы. Предположим необходимо удалить элемент 74. Вначале определяется бакет, в котором находится удаляемый элемент. Затем осуществляется проход по элементам этого бакета, где после каждой проверки условия node.Value == value текущий элемент сохраняется в переменной nodePrev. При выполнении вышеуказанного условия поле nodePrev.nextNode (0xc0000386d0) меняется на (0xc0000384f0), то есть пропускается удаленный элемент (рис. 6.7.):

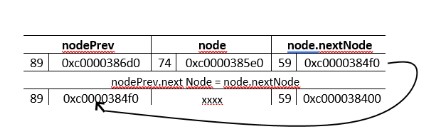


Рис. 6.7. Удаление элемента из хэш-таблицы

Другим примером «хорошей» хэш-функции является для использования с целочисленными значениями ключей является [метод среднего квадрата](https://opendsa-server.cs.vt.edu/OpenDSA/Books/CS3/html/Glossary.html#_blank). Метод среднего квадрата возводит в квадрат значение ключа, а затем извлекает средние цифры результата, давая значение в диапазоне от 0 до M. Программная реализация  алгоритма построения хэш-функции на языке Golang сводится к применению встроенных функций конвертации целых чисел в строковые (strconv.Atoi(i)) и наоборот (strconv.Itoa(i)). Например, для любого четырехзначного числа хэш-функция выглядит так:

func  hFunc(i,) int {

    var j int

    var s string

    i = i\*i

  s = strconv.Itoa(i)

    s = s[3:5]

    j,\_ = strconv.Atoi(s)

    return j

Более реальным случаем построения хэш-таблиц является так называемый эффект коллизий, который заключается в том, что в один и тот же бакет могут попасть одинаковые или близкие по значению объекты. В большинстве задач два и более ключей хешируются (то есть преобразуются в структуру меньшего диапазона) одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку в бакете. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, отображающиеся в этот индекс. Эти варианты и представляют собой две классические схемы хеширования [[Схемы хеширования (studfile.net)](https://studfile.net/preview/7745808/page:59/)]:

- хеширование методом открытой адресации с линейным опробыванием;

- хеширование методом цепочек (со списками), или так называемое, многомерное хеширование.

Однако эта тема находится вне задач настоящего пособия.



Источник: <https://intellect.icu/poisk-dannykh-algoritmy-poiska-sravnenie-algoritmov-4408https://intellect.icu/poisk-dannykh-algoritmy-poiska-sravnenie-algoritmov-4408>

[10.3. Примеры хэш-функций — структуры данных и алгоритмы CS3 (vt.edu)](https://opendsa-server.cs.vt.edu/OpenDSA/Books/CS3/html/HashFuncExamp.html)

<https://studfile.net/preview/7745808/page:59/> !!!!!!!

Источник: <https://intellect.icu/poisk-dannykh-algoritmy-poiska-sravnenie-algoritmov-4408https://intellect.icu/poisk-dannykh-algoritmy-poiska-sravnenie-algoritmov-4408>

