**SECTION 7. BASIC SEARCH ALGORITHMS**

Searching is one of the most important algorithms for processing data structures. Search algorithms are designed to check the presence of an element or extract an element from any data structure in which it is stored. More strictly, the search problem can be formulated as follows: find one or more elements in the set, and the elements sought must have a certain property. This property can be absolute or relative. A relative property characterizes an element with respect to other elements: for example, a minimum element in a set of numbers.

There are a lot of search algorithms. Their complexity varies from simple sequential search algorithms, in extremely efficient but limited binary search algorithms. Of particular note are algorithms based on the presentation of a core set of data in a different, more searchable form, which are used in real-world applications for processing data sets in huge databases.

There is no single algorithm to solve the search problem, nor is there a single algorithm for sorting problems that is better suited for all cases. Some of the algorithms run faster than others, but require additional RAM to run. Others run very quickly, but can only be used for pre-sorted arrays. At the same time, the analysis of search algorithms is somewhat different from sorting algorithms. In particular, there is no sustainability problem for them. In this case there may be situations that require introduction of new criteria of complexity and its assessment [intellect.icu].В целом, все алгоритмы сводятся к выполнению следующих шагов [intellect.icu]:

1. establishing the property of the elements of the source set; in most cases, these are the values of the elements;

2) match the value of the element with the reference property (for absolute properties) or compare the properties of the two elements (for relative properties);

3) traversing the elements of the set.

In principle, search algorithms differ between search methods and search strategies. Based on the type of search operation, these algorithms are usually classified in two categories:

*Sequential Search:* The list or array is performed sequentially and each element is checked.

*Interval search:* These algorithms are specifically designed to search in sorted data structures. These types of search algorithms are much more efficient than linear search because they repeatedly target the center of the search structure and divide the search space in half.

In this section, let’s look at the main algorithms for searching data structures. To better understand the practical use of algorithms, the section provides their DRAKON- diagrams and estimation of time and space complexity.

**7.1. Linear data searching**

a). Linear searching of raw data

In the case of a linear search of an element in an unordered dataset, for example, in an array or in a cut, the simplest algorithm is to view all elements until the desired value is found (Figure 7.1).

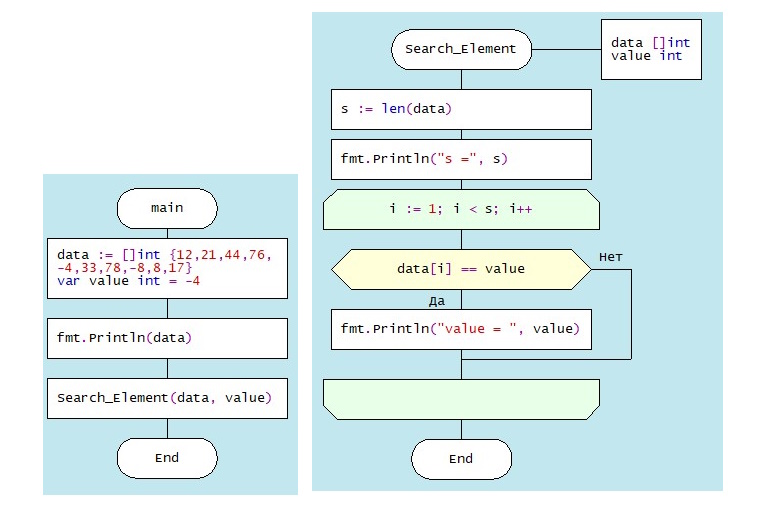


Figure 7.1. DRAKON-diagrams of linear search algorithm

This algorithm is not very effective, but it works on arbitrary collections. *Time complexity:* O(n). In the worst case (the desired element is in the last position) to find the element you need to pass all the elements cut. Here "n" is the size of the cut. additional memory. In principle, another worst case is the absence of the necessary element. *Space complexity:* O(1). No additional memory is required to accommodate the slice.

b). Linear search of sorted data in slice

If the elements of the dataset are sorted by ascending or descending, finding the desired element is much more efficient than in an unordered linear search. Because in many cases, you don’t have to go through the whole list. For example, when an item with a higher value is discovered as a result of passing through an increasing sorted list, the search is stopped. This approach saves time and increases productivity. Figure 7.2. shows the DRAKON-diagram of this algorithm.

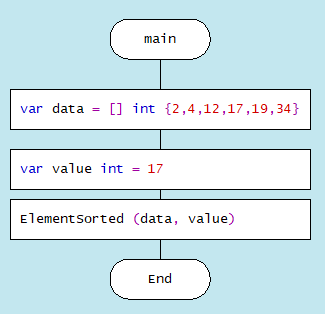
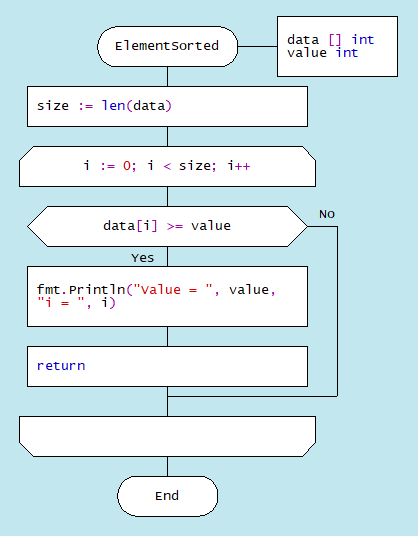
 ****

Figure 7.2. DRAKON-diagram of searching sorted slice algorithm

**7.2. Binary search for data in a sorted slice** Binary search is performed as follows: Specifies the value of the element in the middle of the data structure. The resulting value is compared to the value you are looking for.

1. If the search value is less than the value of the means, the search is carried out in the first half of the elements, otherwise - in the second.3. The search is simply that the value of the middle element in the selected half is again determined and compared to the key.
2. The process continues until an item with the search value is found or the search interval is empty.

The DRAKON-diagram of the binary search algorithm is represented in Figure 7.3. (main() module is similar to the previous algorithm):

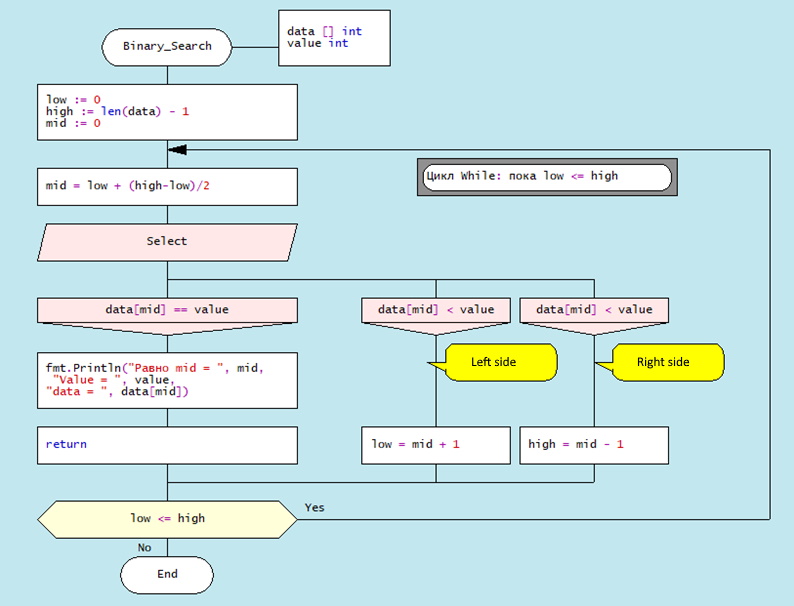
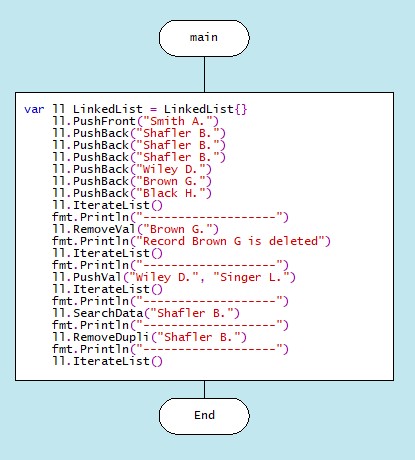


Figure 7.3. DRAKON-diagram of Binary Search algorithm

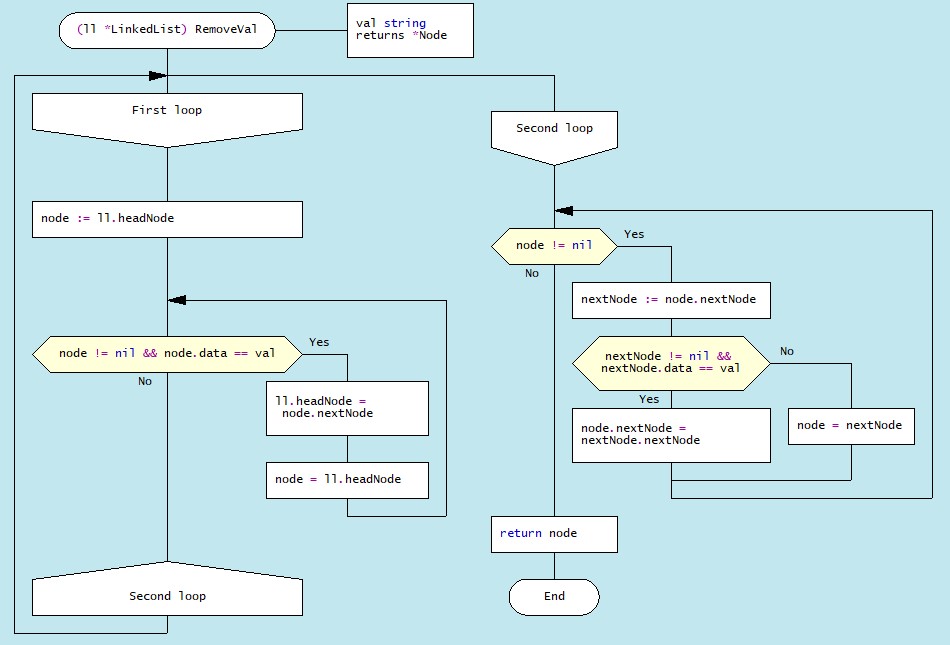
*Time complexity* of binary search algorithm belongs to class O(log n). The way to interpret this is that the asymptotic increase in the time taken by a function to perform a given input set of size n will not exceed log n. *Space complexity*: O(1). That is, no extra space required.

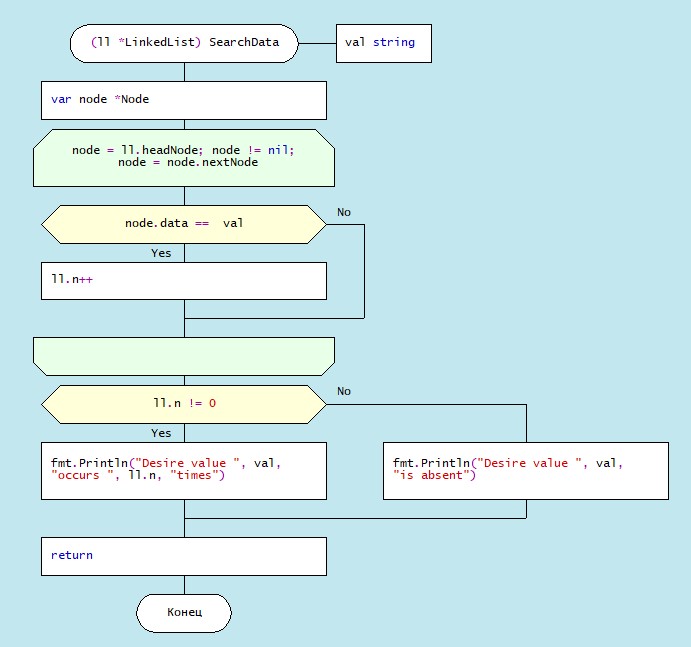
**7.3. Searching in Single-Linked List**

There are three possibilities for a single-linked list. First, the desired value is missing from the list, second, the desired value is encountered once and, third, the desired value is encountered repeatedly. You can also set the task of removing duplicates, i.e., nodes that are redundant. To solve these problems it is necessary to create a Single-Linked List, the items of which contain values "Smith A.", "Shafler B.”, "Wiley D.", "Brown G.", "Black H.". In this list you should delete the entry " Brown G." and then add a new entry "Singer B." placing it after the entry "Wiley D.". After that, you should delete the duplicates of the entry " Shafler B." leaving only one. The corresponding dragon diagrams are presented in Figure 7.4 a,b,c,d:

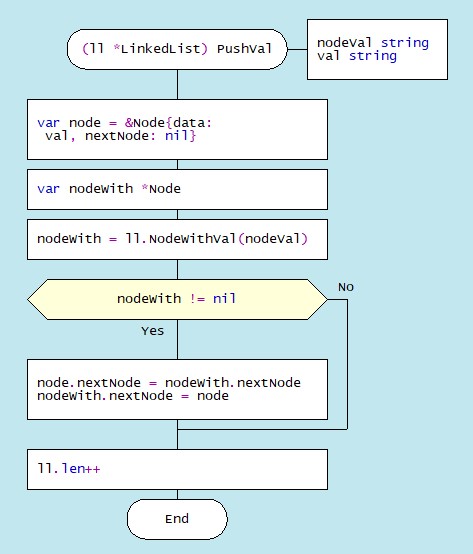
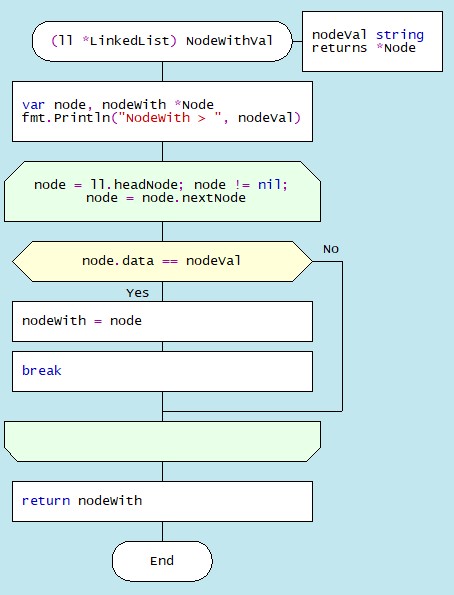


a) function main()

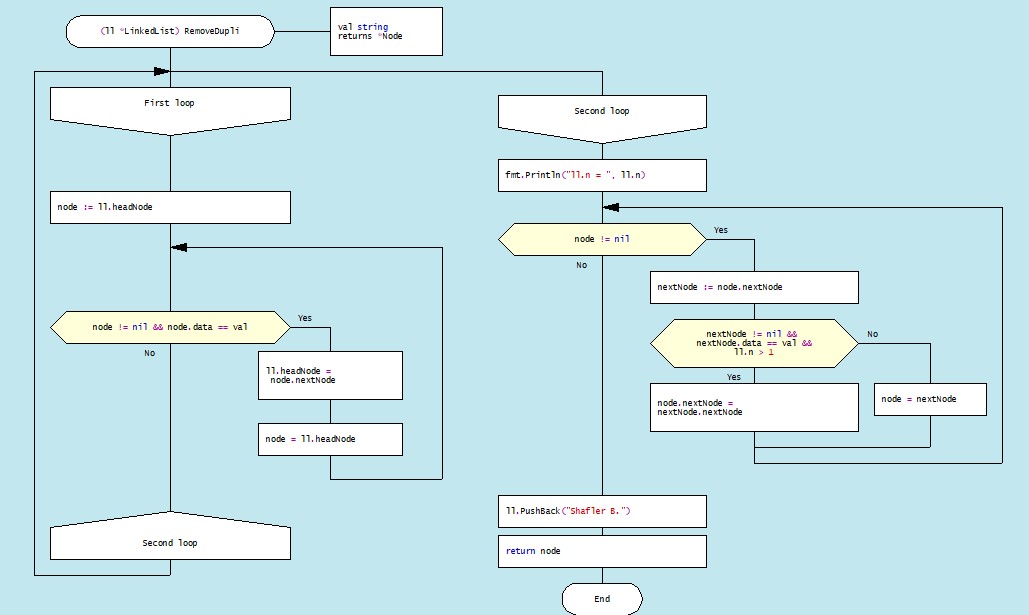
 b) function of deletion RemoveVal(val)



c). function of searching SearchData(val)

d). function of inserting by value PushVal + NodeWithVal



e). function of removing duplicates RemoveDupli

Figure 7.4. DRAKON diagram of algorithms deletion, search, insertion by value and dublicate deletion

7.4. Хеширование

The search time of an item in a data set depends on the number of element value comparisons. In order to reduce search times and thus improve computational efficiency, fewer comparisons are needed. This can be achieved by converting a larger data set into a smaller range called hashing, resulting in hash tables. [https://spravochnick.ru/informatika/algoritm\_poiska\_poisk\_heshirovaniem/].

From the perspective of the theory of abstract data types (ADT), a hash-table is a data structure that implements the interface of an associative array that allows you to store key-value pairs and perform three basic operations: the operation of adding a new pair, search operation and operation to delete the key-value pair. **From a programming position, a hash table is a collection of items containing a key-value pair, where the key is computed by a special function called a *hash function*. A hash-table, in turn, consists of buckets, a set of elements with matching or close hash values of the function. There are different methods of constructing a hash function, the simplest of which is the residual method, where the hash function is defined as the remainder of the division of two numbers (**x, m**), where** x **is the item of the set,** m **is the number of buckets. In Golang, the hash function for this method is:** h = x % m**.**

Let’s analyze the hash table creation process in more detail using the Go language toolkit in the editor DRAKON WEB Editor. First, a variable of the Node type is created, defined as a structure consisting of two fields: the element value is Value int and the next element address is Next \*Node. In fact, it is a single linked list (see Sect. 1).

type Node struct {

Value int

Next \*Node

}

Then, a hash-table is created through a structure that has two fields: the first field (Table) is a map that relates the integer (*hash index*) to the associated list (\*Node), and the second - Size of type int:

type HashTable struct {

Table map[int]\*Node

Size int

}

As a result, this hash table would have to have as many single linked lists (buckets) as was specified by the Size constant. In the above case the number of slots is 15. Note that the Node and HashTable type advertisement, as well as the Size constant, are included in the File/File Drakon description option (Figure 7.5.).

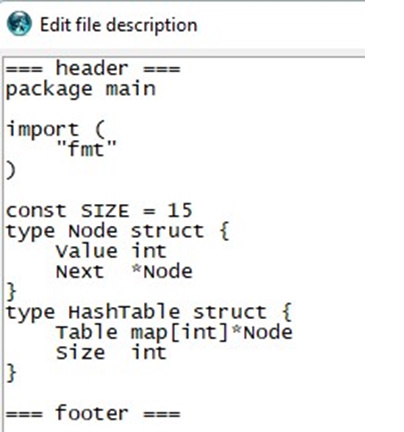


Figure 7.5. Type declaration Node, HashTable and constant Size

As an example, consider constructing a hash-table of size m = 15 for a collection of integers from 0 to 120. The hash table slots are originally empty:

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

**A hash function that reflects the relationship between the element and the slot must accept any element from the dataset (0 ... 120) and return an integer from the slot number range (0 to 14). The algorithm that implements the remainder method simply takes the element from the original set one by one and divides it by 15, returning the remainder as the hash value that is entered into the slot h(item)=item 15%. For example, the hash code for item119 is defined as 119% 15 = (119 - 15\*7) = 14, the value 119 is entered into the corresponding slot:**

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

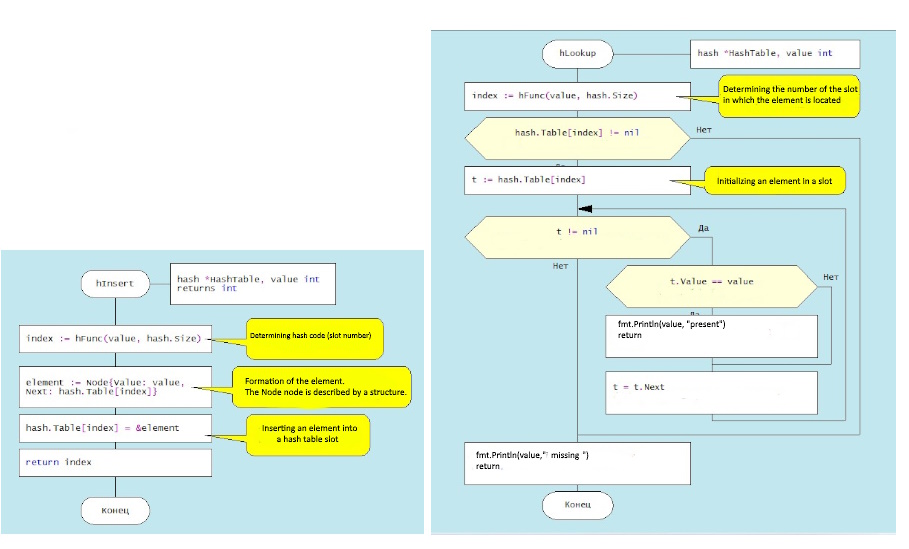
The algorithm then identifies slots for other elements, gradually filling them. When the algorithm encounters element 104 in the loop, then the remainder of the division is 104%15 = 14, so this element will also be included in the 14th slot. In this way, each slot will accumulate corresponding elements with one index hash. For example, for an index hash of 8, the slot will consist of these elements: 113 : 98 : 83 : 68 : 53 : 38 : 23 : 8. And the entire hash table will be a follows (Table 7.1.):

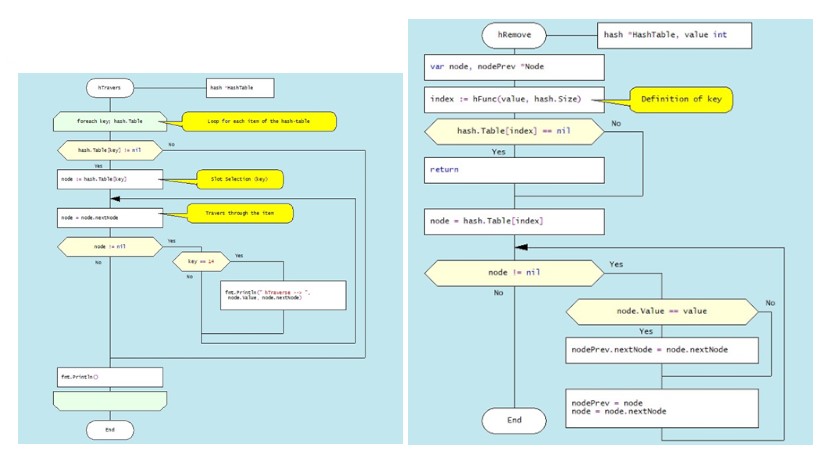
Table 7.1. Hash-table of 15 slots

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |

Thus, the remainder method converts a collection of 120 integers into a hash table of 15 slots. The search for an element is now greatly accelerated as it takes two steps: first, the hash function h =(x % m) computes the hash index, and then the search is done in a 7-element slot. The algorithm based on the residual method is represented by the following dragon diagrams (Figure 6.6.). Here hinsert - module of filling with elements of slots on hash-function, hLookup-module of search of element in hash-table, hTravers - module of passage on hash-table.

The implementation of the main hash table functions using the hash function is as follows: 1.Create a HashTable structure list of the size m to store objects. 2. Compute the object’s hash code by passing it through the hash function. 3. Get the bucket hash indices where the objects will be saved. 4. Save these objects in the designated bucket. DRAKON-diagrams of algorithms for implementing the main functions of working with hash-tables are shown in Figure 7.6.

 a). Hash-table creating function b). Item searching function



c) Hash-table travers function d) Deletion item function

Figure 7.6. DRAKON diagram of hash-table algorithms

Consider the algorithm for removing an element from the hash-table. Suppose we delete element 74. First, the bucket containing the item to be removed is determined. It then passes through the elements of this bucket, where after each node.Value == value check, the current element is stored in the nodePrev variable. If the above condition is met, the nodePrev.nextNode field (0xc0000386d0) is changed to (0xc00384f0), that is, the deleted element is skipped (Figure 6.7.):

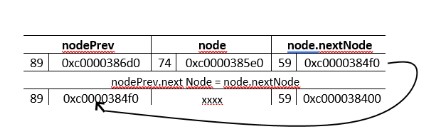


Figure 7.7. Deletion item from hash-table

Another example of a "good" hash function is for use with integer key values, the mean square method. The mean square method squares the key value and then extracts the average digits of the result, giving a value in the range from 0 to M. Software implementation of hash function algorithm creation in Golang language is reduced to the use of built-in functions of conversion of integers into string (strconv.Atoi(i)) and vice versa (strconv.Itoa(i))). For example, for any four-digit number, the hash function is:

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

A more realistic case of hash table construction is the so-called collision effect, which is that the same bucket can get the same or similar objects. In most tasks, two or more keys are hashed (that is, converted into a smaller structure) in the same way, but they cannot occupy the same cell in the hash table in the bucket. There are two possible options: either find a different entry for the new key, or create a separate list for each index hash table into which all the keys displayed in the index are placed. These options are two classic hashing schemes [Hashing schemes (studfile.net)]:

- оpen addressing hashing with linear testing;

- сhain hashing or so-called multidimensional hashing.

However, this topic is outside the scope of this manual.



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>

