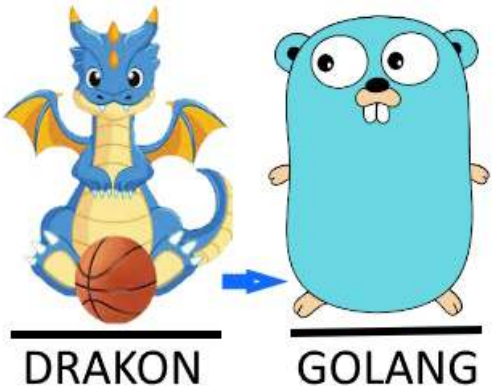
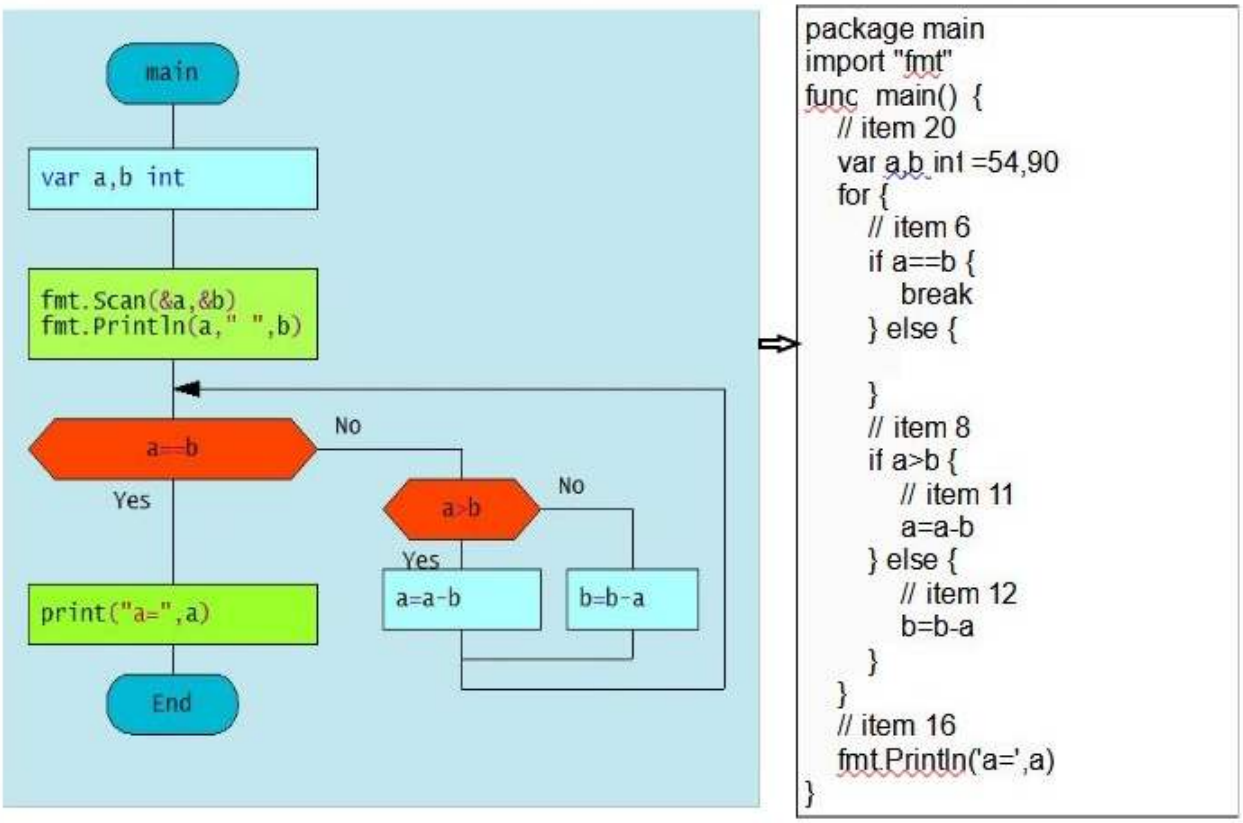
Ivanov S.A.

DATA STRUCTURES AND ALGORITHMS. HYBRID APPROACH





Drakon-diagram to Golang code in 1 click

2024

Preface

When planning a new book, any author sets a specific goal, sometimes even several goals. It is well known that dozens of books, textbooks, manuals, and even laboratory papers with the same title "Data Structures and Algorithms" have been published. They differ only in the originality of the structure of the book, the presence of examples and, most importantly, the programming language that implements the presented algorithms. Among the leaders are C++, Java, JavaScript, Python. However, in the last decade, the list of leaders has confidently included the Golang (Go) programming language, which in certain aspects has a significant advantage, primarily in relation to the high efficiency of competitive programming.At the same time, there are much fewer works on the topic of "Data Structures and Algorithms" that use the Golang programming language compared to the languages presented above.

An important point is the designation of the target audience. Most of the books on the topic are aimed at a wide audience, primarily students of higher educational institutions. In these books, algorithms for processing data structures are represented by verbal descriptions and program codes, the understanding of which is a rather laborious thought process. Algorithm flowcharts could play a role in this regard, but in this case the amount of training material would increase significantly, which is unlikely to make it easier to understand sufficiently complex algorithms. The situation can be saved by numerous services that reveal the content of algorithms with the help of illustrations, but they are fragmentary. In other words, it is rare to find a book that would combine a verbal description of algorithms in conjunction with their software implementation and detailed illustrations. However, this is a debatable issue. However, this is a debatable issue.

Overcoming this drawback, the author ventured to propose a different approach, called "hybrid", since it combines the methodological attractiveness of the schematic (graphic) representation of algorithms in the form of original graphic structures that are automatically transformed into program code. It is a technology

based on the use of a visual programming language (DRAKON), which results in a set of DRAKON diagrams that are automatically converted into program code, in this case in the Golang language. A fragment of the use of this technology is shown on the cover of the book.

During the writing process of the book, the author constantly pondered the question: to what extent is this approach more effective than verbal descriptions of algorithms or comments on program code instructions. Here, another goal of the book can be formulated: to receive feedback from readers, primarily from students, in the form of reviews and critical comments. In this regard, the author hopes for a discussion of the hybrid approach in presenting the material, which is fundamental in teaching computer science – "Data Structures and Algorithms."

The author also hopes for the understanding of readers who have found errors and inaccuracies in the presentation of the material, especially in terms of its design. The hope for indulgence is due to the following factors: the author is a deep pensioner at the age of 83 (eighty-three) years old, he got acquainted with the Golang programming language three years ago, as well as with most of the algorithms presented in the book. In addition, work on the book sometimes took place to the sound of explosions, since the author lives in Ukraine. Finally, this is the author's first experience in writing a book in English, which can probably be easily detected by native speakers.

And, of course, the process of working on the material would be extremely difficult, if not impossible, without constant recourse to Artificial Intelligence services.

In any event, the author will be very grateful to everyone who shows interest in this work and will be infinitely grateful for the reviews that he hopes to read during his lifetime.

Finally, the author would be even more grateful for a small financial encouragement for their work, which would significantly increase motivation to continue as a writer. The next stage involves the creation of an interactive service based on the fundamentals of algorithmization.

INTRODUCTION

The basis of the processes of cognition, understanding, and construction of natural, non-natural and virtual realities, one way or another, is DATA, which can be understood as unstructured information. To structure this data in order to obtain new knowledge or new meanings, it is necessary to process them according to a certain plan of action. In a stricter definition, a data structure is a form of representation of the properties and relationships of a subject area, focused on the expression of a description of data by means of formal languages. Technologically, a data structure is a container in which information is arranged in a certain way that establishes a way to place data in computer memory for quick and efficient access to them.

In turn, algorithms are sets of step-by-step instructions for solving computational problems by processing data structures. Data structures and the processes of their processing according to various algorithms are inextricably linked with each other. In some problems, the choice of algorithm is dictated by the data structure, in others, the data structure determines the choice of algorithm. One of the founders of the formation of computer science as an academic science, N. Wirth, formulated this connection in the title of the book: "Algorithms + data structures = program".

The variety of objective, non-objective and virtual worlds and the processes taking place in them predetermine the emergence of many problem situations, the effective solution of which involves the choice of data structures and algorithms for their processing. A solution is considered efficient if it solves the problem within known resource constraints. First of all, this refers to the total space available for storing data, as well as the time allotted for the execution of each algorithm. When choosing a data structure for solving a problem, one should first determine the basic operations that must be supported, secondly, to quantify the resource constraints for

each operation of the algorithm (occupied memory and execution time of operations).

A data structure requires a certain amount of memory for each stored data element, a certain amount of time to perform one basic operation, and a certain amount of programming effort. Each task has limitations on available space and time. Each solution to a problem uses the underlying operations in some relative proportion, and the data structure selection process must take this into account. Only after a thorough analysis of the characteristics of the problem being solved, it is possible to determine the best data structure for it.

Based on the considerations presented, the Handbook provides basic information about the most commonly used data structures and algorithms, since they form the basis of the professional activity of a programmer of any level. Considerable attention is paid to the issues of evaluating the efficiency of data structures and algorithms in terms of occupied memory and the execution time of basic operations. The organization of data structures and the study of basic algorithms is carried out in the Golang programming language and is accompanied by explanatory illustrations. In a number of important cases, the algorithm visualization technology based on the Drakon graphical language is used.

SECTION 1 reveals the meaning of the main concepts in the discipline Data Structure and Algorithms: data, data type, abstract data types, algorithms. A high level of competence in the development of efficient algorithms is necessary for every programmer, not only when solving practical problems related to the use of IT technologies. Knowledge in the field of data structure and algorithms is the most important component in an interview when applying for a job in a large IT company.

SECTION 2 describes the basics of the Golang programming language in sufficient amount for acquaintance and practical mastery of knowledge and skills in

the field of data structure and algorithms. Of course, basic concepts and a certain level of programming are assumed for reading this chapter.

SECTION 3 gives a brief description of the visual language DRAKON, which allows to represent any complex algorithms in the form of a DRAKON-diagram using icons, corresponding to the basic constructions of the Golang language. The technology of automated transformation of the DRAKON-diagram into the program code is given in the editor DRAKON WEB Editor, with subsequent implementation in the shell of Visual Studio Code. Actually, this is a hybrid, two-stage approach: in the first step a DRAKON-diagram is made, the icons of which are filled with constructions of the language Golang, on the second step - automatically generated software code, implemented in any software environment

SECTION 4 presents Golang language constructs that implement the basic abstract data types described in Section 1: array, cut, list etc. Explanations are given regarding the use of appropriate Golang language constructs.

SECTION 5 provides basic insights into the analysis of algorithm complexity and the evaluation and effectiveness of algorithms from the perspective of computer memory (space) and time costs. Theoretical ideas about estimating the complexity of algorithms are decisive when choosing the most effective one.

SECTION 6 deals with basic sorting algorithms. Each algorithm is represented by a corresponding DRAKON-diagram, accompanied by necessary explanations about the processing process and estimation of complexity.

SECTION 7 discusses basic algorithms for searching elements in different data structures (slices, lists). In addition, the section includes a description and DRAKON-diagram of hash algorithms that significantly reduce search time by reducing the number of element value comparisons.

SECTION 8 provides conceptual information about binary trees – nonlinear data structures, including binary search trees and self-balancing trees. This

section describes the basic DRAKON-diagrams of algorithms for inserting new ones, finding nodes with specified values and deleting nodes.

SECTION 9 provides basic information about the most complex data structures – graphs. Basic concepts of graph theory and their basic types are given. Algorithms of representation of graphs, their traversal, choice of path between vertices are considered.

SECTION 1. ABSTRACT DATA TYPES (ADT)

1.1. ADT concept

Before describing data structures in the programming language Golang should start with the concepts of 'data', 'data type' and 'data structure'. In the most general sense, data is any unstructured set (collection) of characters that are collected and processed for any purpose, primarily to extract information for subsequent acquisition of knowledge and meanings. Data structures manifold was predetermined the appearance of a data type category . The data type should be understood as an atomic, indivisible data unity, defined either by the programming language standard or by the user, and representing a set of real-world objects at the level of their most essential parameters. Generalization of experience of application of different types of data in different programming languages led to the emergence of the «abstract data type» category.

The data types were first described by D. Knuth in his classical work “The Art of Programming”, in which the author described data structures defined as ways of organizing data within a program. Together with the description of the data structures themselves, Knuth provides the “processing algorithms” of these structures in a language of special terms, reflecting actions with elements of this structure, such as structure size, adding a new element, push and pop items, etc.

In most literary sources, the abstract data type is understood as a mathematical model that defines a set of data values (carrier set) and a set of methods for handling that data (method set). Here are some examples of ATD:

Boolean - the logical ATD is a set of values {true, false}. The set of methods includes comparison operators (as well as logical operations &&, II, ==, !=).

Integer - integer ATD is a set of {..., -2, -1, 0, 1, 2, ...}, and the set of methods includes addition, subtraction, multiplication, division, remainder of division, change of sign, etc.

String - an ATD string set is a set of all finite sequences of characters in a certain alphabet, including an empty sequence (an empty string). The set of methods includes concatenation, definition of string length, substring, character index, etc.

Bit string – the carrier set of bit string is a set of all end-sequences of bits, including empty bits strings. The set of ATD method of bit string includes a complement (which flips all bits), shifts (which rotate a string of bits left or right), connection and disjunction (which merge bits at appropriate locations in strings), concatenation and truncation.

The author of the outstanding work «Perfect Code» S. McConnell significantly expanded the concept of ATD and drew attention to the fact that abstract data types «allow to work with entities of the real world, and not with low-level entities of implementation». In particular, these entities are data structures for organizing the computing process. The simplified classification of abstract data structures can be presented as follows (Figure 1.1.)

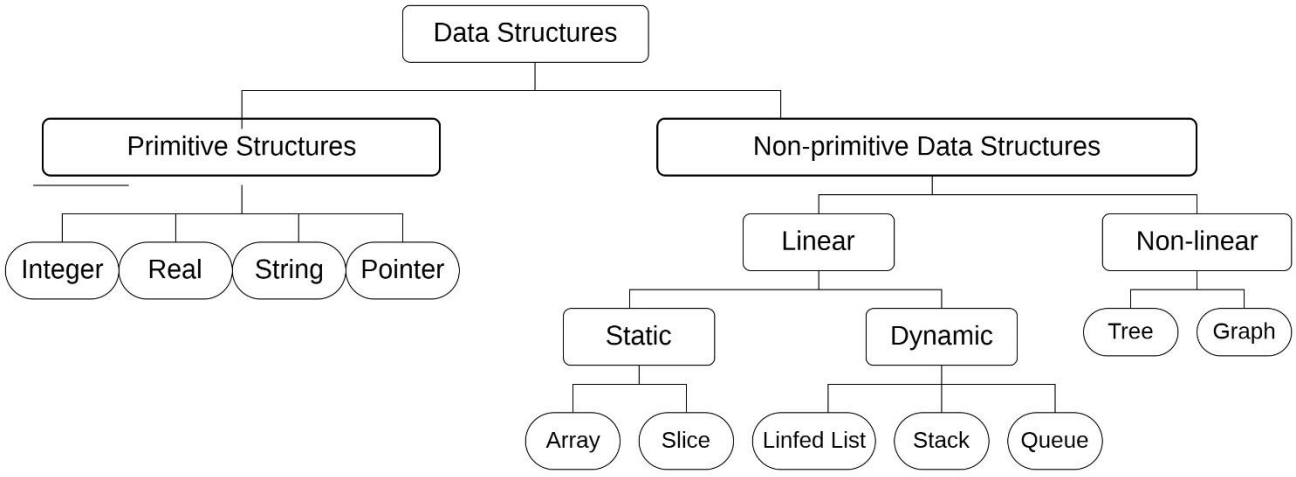


Figure 1.1. Data structures classification

Let's take a closer look at the classification. Note that each of these structures has its own advantages and disadvantages, which predetermines the need for their thorough analysis in terms of computer memory costs and access time to their elements. In this work, the main attention is focused on data structures presented in RAM. First of all, data structures are divided into linear and non-linear. In a linear data structure, its elements are contiguous, that is, the data is ordered sequentially. Such designs are quite simple to

implement. At the same time, linear data structures do not provide efficient use of memory.

In non-linear data structures, the organization of elements is not sequential. Data elements in a non-linear structure may be connected to several other data elements to reflect special relationships between them. In addition, in non-linear structures, it is impossible to go through the elements in one pass. Nonlinear data structures include maps, dictionaries, trees, and graphs. Homogeneous data structures include most of these data structures, non-homogeneous - structures that consist of data of a different nature. For example, lists are based on nodes, represented by structures that include two fields: a numerical value and the address of the next node. Typical examples of heterogeneous structure are *dictionaries, maps, and hash tables*. Let's consider the main abstract data types according to the principle of linearity.

1.2. Linear Abstract Data Types

1.2.1 Array and Slice

Let's start with the fundamental abstract data type, the array. The fundamental nature of an array, as a data structure, lies in their direct correspondence to memory systems on all computers. To retrieve the contents of a word from memory, machine language requires an address. Thus, the entire memory of the computer can be considered as an array, where memory addresses correspond to indices. Most machine language processors translate programs that use arrays into efficient machine language programs that access memory directly.

An array is a fixed set of same-type data that is stored as a continuous sequence, the indexing of which elements can start with 0 or 1. When created and initialized, the array is declared via an identifier or an address pointer at the initial address (0 or 1) element (Figure 1.2).

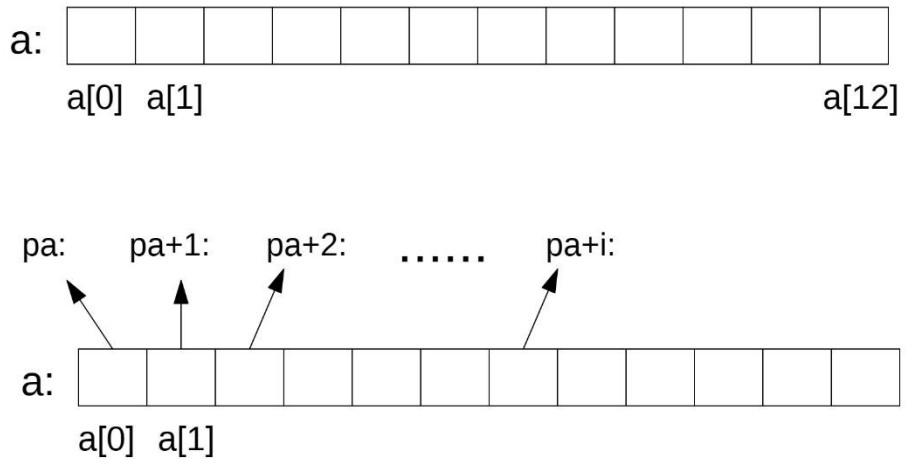


Figure 1.2. Array declaration method (*pa* - cell address *a[0]*)

The choice of declaration type determines how individual elements are accessed. The entry *a[i],* where a is an identifier, refers us to the *i-th* element of the array. At the same time, an array can be declared using a pointer, which refers the program code to the address of the initial element of the array. In most languages that use the concept of pointers, an array is specified by the expression: **pa*, and the real address in a hexadecimal expression is specified by the character &: *pa = &a[0].* If *pa* points to some element of the array, then *pa+1* by definition points to the next element, *pa+i* to the *i-th* element after pa, and *pa-i* to the *i-th* element before *pa*. So if *pa* points to *a[0],* then **(pa+1)* is the content of *a[1], a+i* is the address of *a[i], a* **(pa+i)* is the content of *a[i].*

There is one difference between an array name and a pointer acting as an array name. A pointer is a variable, so pa = a can be written, but an array name is not a variable, and entries like a = pa are not allowed. Consider the most common properties of an array as an abstract data type:

• All array elements belong to the same type (homogeneity);

• The size of the array is set once and does not change during the operation (persistence);

• All array elements have the same access (equity);

• All elements are arranged sequentially in RAM cells (location sequence);

• Array elements are uniquely identified by their indices (indexing);

• Indexes must be a simple ordinal data type.

It should be noted that any abstract data type consists of a set of values and a set of methods. For an array, the main methods are:

• Get element with number N;

• Record element with number N;

• Get the size of an array.

The advantages of ATD «array» include:

- Arrays store multiple data types with the same name.
- It provides arbitrary access to elements.
- Since the array is fixed size and stored in adjacent memory regions, there is no memory shortage or overflow.
- It is useful to store any type of data with a fixed size.
- Because elements in an array are stored in adjacent areas of memory, it is easy to iterate in this data structure, and access to the element, if known, takes a unit of time.

Limitations of array data structure include:

- The size of the array must be known in advance.
- An array is a static data structure with a fixed size, so the array size cannot be changed additionally, and therefore no changes can be made during execution.
- Insertion and deletion operations are expensive in arrays because elements are stored in continuous memory.

- If the size of the advertised array exceeds the required size, this can result in memory loss.

This disadvantage can be overcome by introducing an abstract data type such like *slice*. S*lice* is a flexible and variable data structure consisting of several elements of the same type. Like arrays, the slice is indexed and has a size (length) that can be programmatically regulated (increasing and decreasing) by having such a property like *capacity*. The slice capacity is the number of elements in the base array, counting from the first element in a slice. If a slice has sufficient capacity, then its length can be increased by it re-slicing. The slice may consist of three elements: *a pointer* indicating the first slice element, a length (slice element number) and a capacity - the maximum size to which it can be extended (Figure 1.3.):

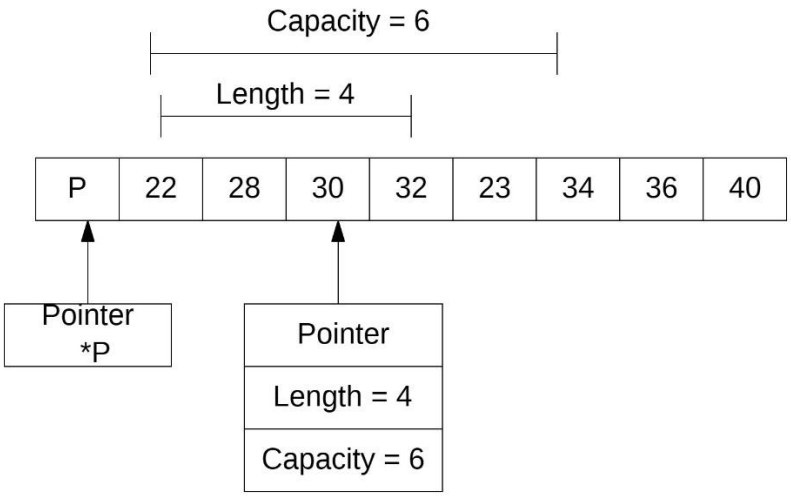


Figure 1.3. Slice and it components

The main methods for slice are the operations of adding, removing, determining a slice length, changing a capacitance. The program implementation of Golang slice (declaration, initialization, basic methods of working with this type of data) is described in more detail in the next section.

1.2.2. Linked List

A list is an ordered set consisting of a variable number of elements to which inclusion and deletion operations apply. A list reflecting the neighbourhood between

elements is called linear. The list length is equal to the number of elements in the list. If the component is not related to any other, the index field contains a value that does not indicate any element. This link is indicated by a special name - null.

There are three common types of connected types:

- Singly linked list;
- Doubly Linked list;
- Circular Linked List.

a) Singly linked list

Figure 1.4. shows the structure of the simply connected list, where the field *item* is containing *data*, the next one - *pointer* to the next element of the list. Each list must have a special element called the top of the list or head of the list, which is usually different in format from the rest of the items. In the index field of the last element of the list there is special feature null indicating the end of the list (Figure 1.4):

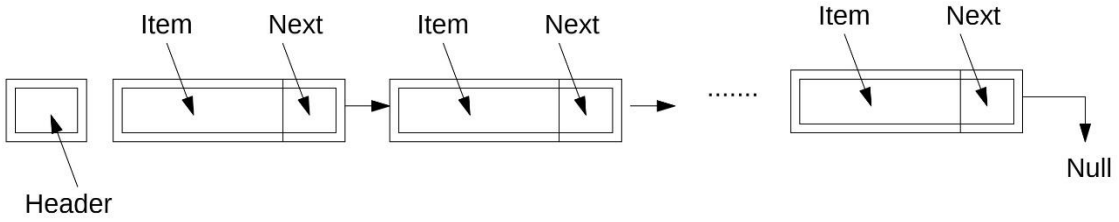


Figure 1.4. Single-linked list (item – data; next - *pointer* to the next element)

Single-linked list, thus defined, has a number of properties:

- The size of a list is the number of elements in it, excluding the last "zero" element, which is by definition an empty list.
- The element type is the type on which the list is built; all items in the list must be of this type.

- Sorting - the list can be sorted according to some sort criteria (for example, by increasing integer values if the list consists of integers).
- Access capabilities - some lists, depending on the implementation, can provide the programmer with selectors to access directly to a given number.
- Comparability - lists can be compared against each other for consistency, and depending on the implementation, the list comparison operation can use different technologies.

Below is some set of functions that can be performed over the ATD list:

• Initialize the list;

• Insert a new element;

• Remove any element;

• Find the k-th element;

• Read the next element;

• Printing elements;

b) Doubly linked list

In some cases, it is necessary to include links on both sides. In this case, both direct and reverse access is possible. It is defined as a sequence of nodes, where each node defines a region of memory. Each node is divided into three fields: *item* is the data field, *next* is the address of the next field, or forward field, and the previous, or reverse field. The data field is used to store data values. The *prev* field is used to store the address of the previous element, *next* is used to store the address of the next element. The first node of the previous field and the last node of the next field are always zero (Figure 1.5.).

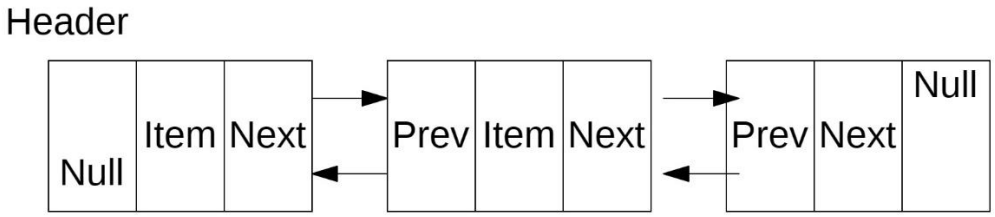


Figure 1.5. Doubly linked list

The properties and functions for the biconnected list are almost the same as for the simply connected list. The main advantage of the two-linked list is that it simplifies many operations; the main drawback is the extra memory required for address pointers.

c) Circular linked list

A variation of the types of linear lists considered is the circular list, which can be organized on the basis of both simple and two-linked lists. In this case, in the simply connected list, the pointer of the last element should point to the first element; in the two-linked list in the first and last element, the corresponding pointers are overridden, as shown in Figure 1.6.

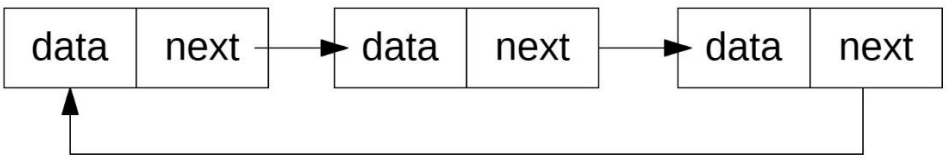


Figure 1.6. Circular list structure

When working with ring lists, some procedures are also simplified; however, when viewing such a list, you should take some precautions not to get into the infinite loop.

Some procedures are simplified when dealing with such lists. However, when viewing such a list, you should take certain precautions not to get into an endless cycle.

1.2.3. Stack and Queue

The stack is a dynamic, ever-changing set of data, the addition and deletion of which are performed solely on the principle of "last logged in, first went out" (Last-In-

First-Out or LIFO) (Figure 1.7). The stack works like manipulating a stack of books, which are shaped by actions to delete and add books.

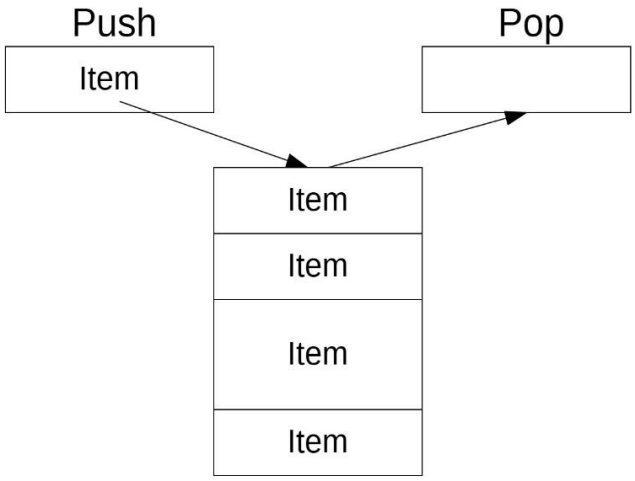


Figure 1.7. Stack structure

Stacks are widely used in various data processing scenarios, such as implementing "undo" in text editors; this operation is performed by storing all text changes to the stack. The stack is often used to organize methods calls and recursions.

The main operations that can be performed on the stack are:

• Add new data;

• Remove data from the stack;

• Return (without removing) the top value;

• Check stack full status;

In many programming languages, the stack is created either from an array or from a single-linked list. The Golang language can apply the concept of interfaces, which will be discussed further.

Unlike the ATD stack, which has much in common with the stack, elements are added (enqueue) and removed (dequeue) from different ends. This method is called "first entered, first released" (First-In-First-Out or FIFO). That is, elements are taken from the queue in the same order in which they are placed (Figure 1.8).

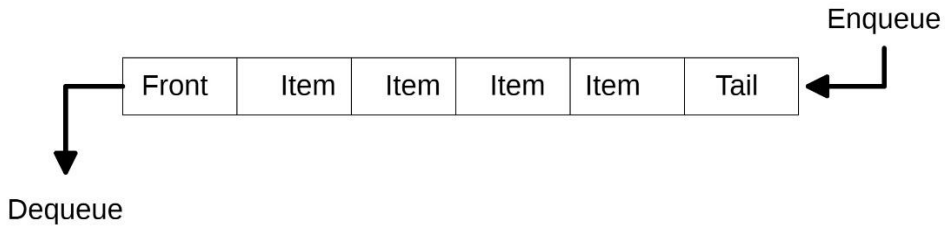


Figure 1.8. Data structure “Queue”

The following functions are used to manipulate queue components.

• Create a new empty queue;

• Add a new element to the end of the queue;

• Remove the front element from the queue;

• Check the queue for emptiness;

• Check the queue for filling;

• Determine the length (number of elements) in the queue.

The data type of the queue can have several varieties:

A *circular queue* is an extended version of a regular queue in which the last element is connected to the first element, resulting in a circular structure. Circular queue solves the main constraint of regular queue - inefficient memory usage.

A priority queue is a special type of queue in which each element is associated with a priority value. And the elements are serviced on the basis of their priority. That is, the higher priority elements are served first. However, if items with the same priority meet, they are served according to their order in the queue. In a normal queue, the "first in first out" rule is implemented, while in a priority queue, the values are removed based on priority: the highest priority element is removed first.

Queued data storage is widely used to plan CPU, printer, and other output tasks. The FIFO method is used when processing interrupts.

1.3. Nonlinear abstract data types

This subsection considers nonlinear data structures, which include the most common structures, namely *trees and graphs*. Recall that nonlinear data structures allow the expression of more complex relationships between elements than linear neighbourhood relationships, as in linear structures.

1.3.2. Trees as ATD. In the abstract representation, trees are understood as non-linear data structures consisting of nodes representing a hierarchy of relationships (parent-children). Each vertex of the tree, if there are descendants, is connected to them by direct edges (Figure 1.9).

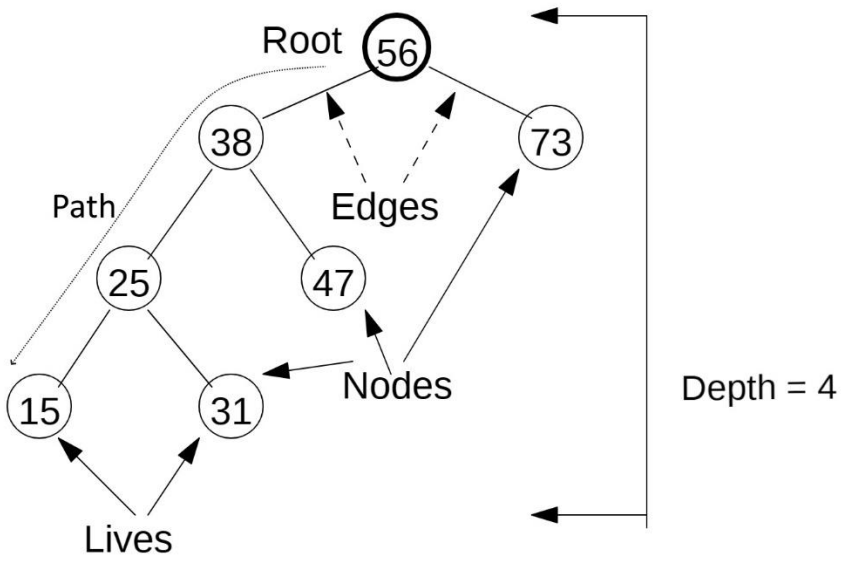


Figure 1.9. Basic tree terminology

Let’s give a definition of the basic concepts of the theory of trees.

Root: The root of the tree is the only node without any incoming edges. This is the top node of the tree;

Node: This is the fundamental element of the tree. Each node has data and two links that can point to null or its descendants;

Edge: This is also the fundamental part of the tree that is used to connect two nodes.

Path: A path is an ordered list of nodes connected by edges.

Leaf: A leaf knot is a knot that has no descendants.

Tree height: Tree height is the number of edges on the longest path between the root and the leaf.

Node level: Node level - number of edges on the path from the root node of this node.

Several computational tasks require the organization of data in different types of trees, but binary trees are particularly well studied and common. Within the concept of abstract data types, a binary tree is a set of connected nodes in which each node contains a value (the data element itself) and has no more than two children. This means that the power of the binary tree is zero or one or two. From the ATD perspective, binary trees contain values of elements of type T. The load-bearing set of this type is the set of all binary trees whose vertices are of type T. The carrying set thus includes an empty tree, trees only with root of type T, trees with root and left child, trees with root and right daughters, and so on.

To the degree of the vertices, binary trees are: strict - the vertices of the tree have a degree of zero (in the leaves) or two (in the nodes); weaker - the vertices of the tree have a degree of zero (in the leaves), one or two (in the nodes). In general, a k-level binary tree can have up to 2k-1 vertices. A binary tree containing only fully populated levels (that is, 2k-1 vertices on each k level) is called complete. Binary tree species are shown in Figure 1.10.

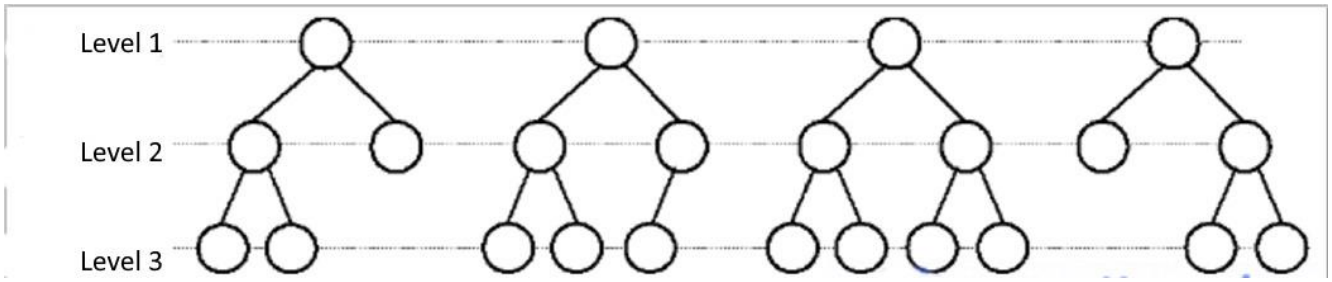


Figure1.10. Binary tree species

Common operations for binary trees as abstract data types are:

- Insert *k*-th element;
- Remove *k*-th element;
- Search for an element with the specified value *k*;
- Find the maximum value stored in the tree;
- Find the minimum value stored in the tree.
- Find the number of tree levels.

The implementation of basic algorithms for binary search trees is presented in Section 8.

1.3.2. Graphs like ATD. In the most general definition, a graph G is given by a set of vertices {V} and a set of edges {E} connecting all or part of these vertices. In other words, a graph G is completely defined by a pair of{V, E}. If edges are oriented, which is usually shown by an arrow, then they are called arcs, and the graph with such edges is called a directed (directed) graph (Figure 1.11 a). If edges have no orientation, then the graph is called non-directed (undirected) (Figure 1.11 b):

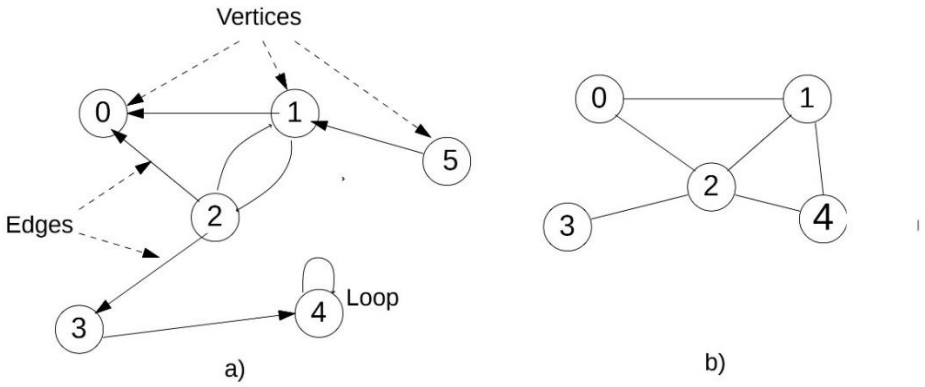


Figure 1.11. View of graph a) directed; b) non-directed

Vertices and edges are called graph elements, the number of vertices in the graph is an order, the number of edges is the size of the graph. The vertices (u,v) are called the endpoints of the e = {u,v}, and the two endpoints of the same edge are called adjacent. Two edges are called adjacent if they have a common endpoint. Two edges are called multiples if the sets of their endpoints are the same. An edge is called a loop if its ends coincide, that is, e = {u,u}. If the vertex vi is the beginning or end of the edge ek, then they (vertex and edge) are incident. The number of edges incident to a vertex is called the vertex degree (Figure 1.12).

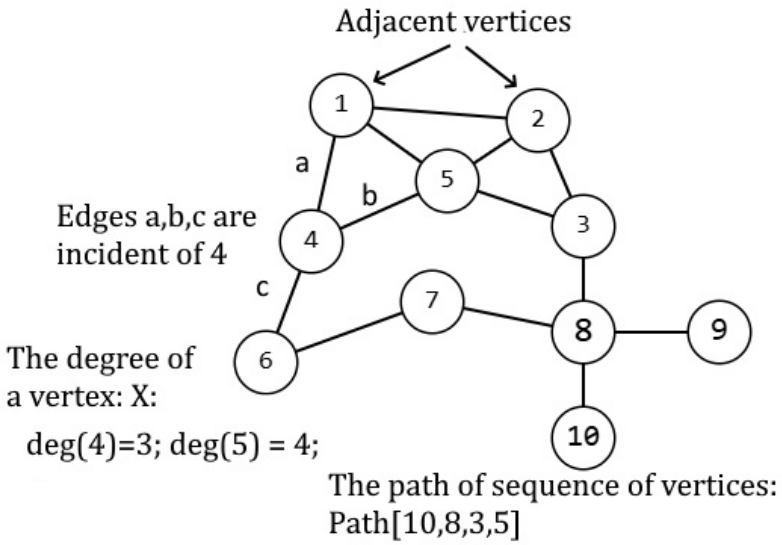


Figure 1.12. Basic graph parameters

There are many types of graphs, among which the most significant are:

- a zero graph in which there are no edges;
- a trivial graph with only one vertex;
- a non-directed graph in which the edges have no direction;
- a directional graph in which the edges have directions;
- a complete graph in which each node has an edge to another node;
- The weighted graph in which the edges are specified with the weight.

Accordingly, there are many operations with graph elements, in particular:

▪ add an edge between two vertices;

▪ operation to remove an edge while maintaining all vertices of the graph;

▪ operation of adding a vertex to a set of vertices;

▪ operation of adding this rib to the set of ribs;

▪ Dijkstra operation, which determines the minimum distance between two given nodes.

More detailed graph algorithms are discussed in SECTION 9.

...

SECTION 2. GOLANG PROGRAMMING LANGUAGE OVERVIEW

2.1. What is the Go Programming Language?

Since this work uses an integrated approach to software implementation of algorithms, it is necessary to at least briefly describe the basic concepts and constructions of its components: visual algorithmic language DRAKON and programming language Golang. It is thought that this approach is very promising, foremost, in the field of education, because it promotes the formation and development of algorithmic, computer thinking and is quite justified for application in the field of computer science as data structures and algorithms. However, it is assumed that the reader is familiar with at least one of the modern programming languages. At the same time, the Golang language description is limited to constructs sufficient to understand data structure algorithms.

This section focuses on the features of software constructions: *variables, arrays, slicess, maps, pointers, logical operators, cycles, structures, recursion,* etc. Special emphasis is placed on the description of data types both basic and user. It is also considered to be an interface. Everything that goes beyond "Data structures and algorithms" is presented in various materials, which are given both in the course of the text and in the list of literature.

Golang (Go) is a general-purpose language designed with system programming in mind. The language was originally developed at Google in 2007 by Robert Grizemer, Rob Pike and Ken Thompson. It is strongly and statically typed (requires precise indication of types of variables), provides built-in support for garbage collection and supports parallel programming. The following are the most important features of Go programming language.

- Simple and clear syntax. This makes writing code a pleasant occupation.
- Static typing. Avoids inadvertent errors, simplifies the reading and understanding of code, makes code unambiguous.

• Speed and compilation. Go’s speed is ten times faster than scripting languages, with less memory consumption. At the same time, the compilation is almost instantaneous. The whole project is compiled into one binary file, without dependencies. As they say, "just add water". And you don’t have to worry about memory, you have a garbage collector.

• Deviation from the concept of object-oriented programming (OOP). The language does not have classes, but there are data structures with methods. Inheritance is replaced by an embedding mechanism. There are interfaces that do not need to be explicitly implemented, but only enough to implement the interface methods.

• Parallelism. Parallel computations in the language are simple and graceful. Gorutins (something like threads) are lightweight, consume little memory.

• Rich standard library. The language has everything necessary for web development and not only. The number of third-party libraries is constantly growing. You can also use C and C++ libraries.

• Ability to write in a functional style. The language has closures (closures) and anonymous functions. Functions are first-order objects, they can be passed as arguments and used as data types.

2.2. Program structure in Go

The structure of the program in the language Golang is presented in Figure 2.1.

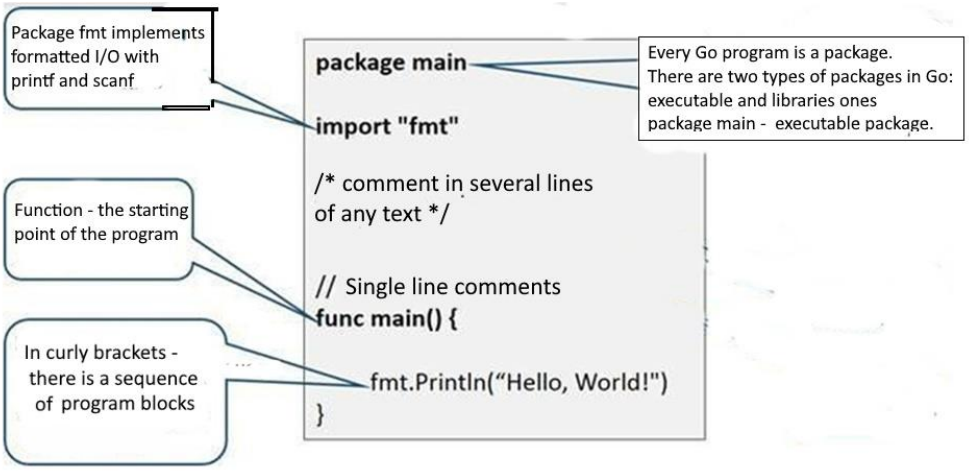


Figure 2.1. Program structure in Go language

Every program on Go is a package. Depending on its content, this package can be the main program file, or a library with problem-oriented code.

package main

…….

// loading the right packages if the commands from these packages are needed in our program import (

"fmt",
“flag”
)

/* main function means that its contents will be executed after the program is run. In addition to it, the program can have as many other functions /*

func main() {

// function content

fmt.Println("Hello World")

}

Comments in Go exist in two types: single-line and multi-line.

// - one-line commentary;

/* */ - multiline comment placed between these symbols

2.3. Variables and constants

A variable in any programming language assigns a storage location for a value associated with a symbolic name or identifier. One variable or list of variables used in the program is declared via the keyword var. Values are assigned to variables by the sign "=". When declaring a variable, you must specify its type so that the compiler knows how to process it. That is why the language Go is called strictly typed.

```
var i int
var s string
var f bool,
```

here i is an integer variable; s is a string variable, f is a logical variable.

When declaring a variable, you can assign initial values that can be changed later:

```
var a int  = 25
var a, b, c int 100, 200 300
var y = float32 = 32.5
var z string = “Enter new values"
```

In the absence of initial values, the variables are either zeroed (for numeric types) or filled in with empty lines (for string data). It is possible to briefly declare variables via the operator :=, for example a := 2.5. However, in this case, the variable is local, that is, only available in a separate fragment of the program.

Assigning variable names should be based on certain rules and style.

- variable names can only contain one word (without spaces).
- variable names can only contain letters, digits and underscore characters (_).
- variable names cannot start with digits.

The variables are case-sensitive, but the case of the first letter of the variable name in Go is particularly important. If the variable name starts with a capital letter, this means that the variable is available outside the package where it was declared (the exported

variable). If the variable name starts with a lowercase letter, it will only be available in the package where it is declared.

Constants are program objects whose values do not change in the program code. Constants are declared with the keyword: const:

```
const item string
const n int
const y float64
```

It is allowed to assign initial values without declaring type:`const item = "name"`

```
const n = 25
const  y = 45.5
```

2.4. Input and output

Data entry is done by using the Scan(&Variable Name) function to simply put the entered value into a variable or Scanf(%format, &Variable Name) to pre-specify the type of data to be entered.

To implement the data output at the beginning of the program (after package main), the corresponding fmt package is imported, which contains many output functions depending on the type of variables, location and explanatory string enclosed in double quotation marks (n - line translation). The prefix *fmt* is used to communicate with the corresponding package.

```
fmt.Print (“Hello”) // output without newline
fmt.Println () // output with cursor to new line
fmt.Printf() // outputting variable values to text
fmt.fmt.Printf("Hello %d\n", 23) // print string and integer type
fmt.fmt.Fprint("Hello ", 23, "\n") //same with newline
fmt.fmt.Println("Hello", 23) // string and number output
fmt.Printf("This year I passed %v the cities, ", &output)
```

For example:

```
package main
import “fmt”

func main() {

    var name string
    var salary int
    fmt.Println("Enter last name ")
    fmt.Scanf("%s \n", &name)
    fmt.Println("Enter salary ")
    fmt.Scanf("%d \n", &salary)
    fmt.Printf("%s \n",name)
    fmt.Printf("%d \n",salary)
    fmt.Printf("Salary %s is %d dollars ", name, salary)
}
```

Outprt:

```
Enter last name
Smith
Enter salary
1028
Smith
1028
Smith's salary is $1028/week
```

2.5. Decision operators

When solving a multitude of problems, the problem arises of choosing a further way of solving depending on some conditions. In the Go language, this possibility is implemented by the following constructions:

- selection by condition: if else; switch case; select;
- repetition of commands using iterations: for (range);
- change of behavior in the course of iterations: break and continue.

2.5.1. Conditional statement if-else

The Go language uses three decision-making constructs or, in other words, a choice of further computational process: if-else, switch and select. The syntax of the first operator is as follows:

```
    if condition {
    // executable code provided condition == true
   } else {
   executable code provided condition == false
}
```

For a single comparison, curly braces can be omitted. The opening bracket stays on the same line as the condition:

```
package main
import "fmt"
func main() {
    // if the conditionis correct
    if ID == "Apple" {
         //what is done is what is in brackets:
        fmt.Println("enter your login and password ")
    /* if the condition is not met, then you can immediately
check another condition. You can do this any number of times
*/
    } else if ID =="Google" {
         // get the second answer
        fmt.Println("Your operating system is not supported ")
    // do what belongs to the last if
    } else {
         fmt.Println("Input error")
    }
}
```

2.5.2. Switch statement

The switch expression switch provides a simple way to access different parts of the program based on the value of the expression. Syntax of switch statement:

```
switch optstatement; optexpression{
case expression1: Statement..
case expression2: Statement..
...
default: Statement..
}
```

After the first variant found, the operator performs the necessary actions and stops work.

```
switch ID {
    // first value is checked
    case "Apple":
        fmt.Println("Enter your username and password ")
    // second value is checked
    case "Google":
        fmt.Println("second value is checked ")
    // if nothing was found
    default:
        fmt.Println("Input error ")
}
```

2.5.3. Select statement

The select statement allows you to select one of several expressions to execute. The main difference between select and switch is that select works on a standby basis. This means that the select command will not run until the message associated with sending and receiving on any channel is executed. It should be noted that the visual

algorithmic language DRAKON in the implementation of the editor Drakon Web Editor uses a similar operator Select, which will be discussed in the next section.

2.5.4. Loops

The Golang language adopts only one loop format - *for*, which has the following variants.

1. The classical, C-like cycle with variable, condition and cycle step:

for i := 0; i < 8; i++ { loop body }

2. Preconditioned loop

The simplest loop is to declare only the condition, and place the rest inside the loop:

```
package main
import "fmt"
func main() {
    // loop variable
    var count = 10
    // as long as the variable is greater than 0, the
loop runs
    for count > 0 {
        // output the current value of the variable
        fmt.Println(count)
        // output the current value of the variable
        count = count - 1
    }
```

2.5.5. Loop within range

There is another for loop variation that iterates the range of values for the data type. The range keyword is used in a cycle to iterate the elements of *an array, slice or*

map. When using an *array* and a *slice*, the loop returns an index of the element as an integer. When using cards, it returns the key-value of the following pair:

```
package main
    import "fmt"
    func main() {
        nums := []int{2, 3, 4}
        sum := 0
        for _, num := range nums {
            sum += num
        }
        fmt.Println("sum:", sum)

        for i, num := range nums {
            if num == 3 {
                fmt.Println("index:", i)
            }
        }
        kvs := map[string]string{"ca": "Paris", "co": "France"}
        for k, v := range kvs {
            fmt.Printf("%s -> %s\n", k, v)
        }
        for k := range kvs {
            fmt.Println("key:", k)
        }
    }
```

The following table provides a complete list of Golang keywords:

break	case	chan	const	continue
default	defer	else	fallthrough	for

func	go	goto	if	import
interface	map	package	range	return
select	struct	switch	type	var

2.6. Data types

Data types define the types of values that are saved by variables when writing the program. Data types also help identify operations that can be performed using data. The following types of data types are dislocated.

- Fundamental types. This category includes numbers, strings, and logical (boolean) values.
- Aggregate types. This category includes array and structures
- Reference types. This category includes pointers, cuts, associative array (map) and functions.
- Embedding types. Go supports user-defined types as aliases or structures.

2.7.1. Basic types

a). Integer and real types of variables and constants

Integer Go types are divided into whole with a sign and whole without a sign. Integer types with the sign include int, int8, int16, int32, and int64; whole types without the sign include uint, uint8, uint16, uint32, uint64, uintptr. The range of values and amount of occupied memory of integer types are represented in the table. 2.1:

Table 2.1

Type	Range	Occupied computer memory
int8	–128 — 127	8 bit (1 bytes)
uint8	0 — 255	
int16	–32 768 — 32 767	16 bit (2 bytes)
uint16	0 — 65535	
int32	–2 147 483 648 — 2 147 483 647	32 bit (4 bytes)
uint32	0 — 4 294 967 295	
int64	–9 223 372 036 854 775 808 — 9 223 372 036 854 775 807	64 bit (8 bytes)
uint64	0 — 18 446 744 073 709 551 615	

The Go language adopts two floating point number formats:.

- float32 - The largest float32 is the constant math.MaxFloat32, which is about 3.4e38. The lowest positive value is 1.4e-45/.

- float64 - The largest float64 is the constant math.MaxFloat64, which is about 1.8e308. The smallest positive value is 4.9e-324.

Float32 provides about six decimal places of accuracy, while float64 provides about 15 digits. The float64 type is preferable for most problems, because when float32 is used, many iterative algorithms quickly accumulate a roundoff error.

To determine to which data type the Go compiler assigns a variable, the Printf() function is used, the parameters of which have a special symbol %T:

```
    a := 234,45
fmt.Printf("Type %T for %v\n", a, a)
Output: Type float64 for 234.45
```

b). String variables and constants

A string is an immutable sequence of bytes. Strings may contain arbitrary data, including bytes with the value 0, but usually contain readable text. Text strings are usually interpreted as UTF-8-encoded sequences of code points. The built-in len function returns the number of bytes per string, and the print operator of the %x specification returns one byte of s for the Latin alphabet:

```
func main() {
    s := "Hello, World"
    fmt.Println("String >> ", s)
    fmt.Println("String lengh= ", len(s))
    fmt.Println("Hex bytes: ")
    for i := 0; i < len(s); i++ {
        fmt.Printf(" %x",s[i])
    }
    fmt.Println()
    q := " Hello, World "
    fmt.Println()
    fmt.Println("String >> ", q)
    fmt.Println("String length = ", len(q))
    for i := 0; i < len(q); i++ {
        fmt.Printf(" %x",q[i])
        }
```

Output:

```
String a >>  Hello, World
String length = 12
```

```
Hex bytes:
 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64
String >>  Hello, World
String length =  20
```

d0 9f d1 80 d0 b8 d0 b2 d0 b5 d1 82 2c 20 d0 9c d0 b8 d1 80

c) Logical variables and constants

The logical data type (bool) can be either true (truth) or false (falsehood). Boolean operators are used in programming to compare and control data flow:

```
func main() {
x := 5
     y := 8
     fmt.Println("x == y:", x == y)
    fmt.Println("x != y:", x != y)
    fmt.Println("x < y:", x < y)
     fmt.Println("x > y:", x > y)
     fmt.Println("x <= y:", x <= y)
     fmt.Println("x >= y:", x >= y)
    }
```

Output:

```
x != y: true
x < y: true
x > y: false
x <= y: true
x >= y: false
```

2.6.2. Aggregate types

In Go language, the term “Aggregate types” refers to data types that are formed by combining basic data types

a) Array

An array is a series of fixed-length data that is used to store homogeneous elements in memory. Arrays in Go are almost identical to arrays in other programming languages. Array elements are indexed using the [] index with their zero position, which means that the first element index is array[0] and the last element index is array[len(array)-1], where len(array) is the array length. However, due to the fixed length of arrays are not very popular unlike the Slice (Cut) design, which is incomparably used in Go. The syntax of the array looks like this:

[N]T{value1, value2, value3, ...value n}, where N – numbers of elements

b) Structure

Structure design is the type of data defined by the developer and serves to represent any real objects. Structures contain a set of fields that represent different attributes of an object. Type and struct keywords are used to define the structure:

type *s name_* struct{

structures fields

}

The following is an example of an enterprise employee data structure:package main

```
import "fmt"


type Employee struct {
   firstName, lastName, address string // string type fields
   age, phone, salary int   // integer type fields
}
func main() {
// structure initialization
 emp := Employee{firstName: "Max", lastName: "Smith", age: 42, phone: 123456789, salary: 34000, address: "Amarillo
 "}
fmt.Println("first and last names: ", emp.firstName, emp.lastName)
fmt.Println("Employee age: ", emp.age)
fmt.Println("Employee salary: ", emp.salary)
fmt.Println("Employee telephone: ", emp.phone)
fmt.Println("Employee address: ", emp.address)
}
```

The memory for a new struct variable is allocated using the new function, which returns a pointer to the allocated memory:

var w *T = new(T),

or in different lines if the structure is declared in the package area:

var w *T

w = new(T)

When using the short form of assigning a variable value (:=), that is, w := new(T), the variable z is a pointer to a memory area where the structure fields contain null values according to their types:

```
 type structEmpl struct {
name string
age int
salary float64
}
func main() {
 ms := new(structEmpl)
 fmt.Println(ms)
 ms.name = "Fangur"
 ms.age = 45
 ms.salary = 1200.5
 //fmt.Printf("Last name: %s\n", ms.name)
 fmt.Printf("Age: %d\n", ms.age)
 fmt.Printf("Salary: %8.1f\n", ms.salary)
 fmt.Println(ms)
}
```

2.6.3. Refetence types

a) Pointer

Pointers in the Golang programming language are variables that are used to store the memory address of another variable. Variables are used to store some data at a specific memory address in the system. The memory address is always represented in hexadecimal format (starting from 0x, for example, 0xFFAAF, etc.) (Figure 2.2):

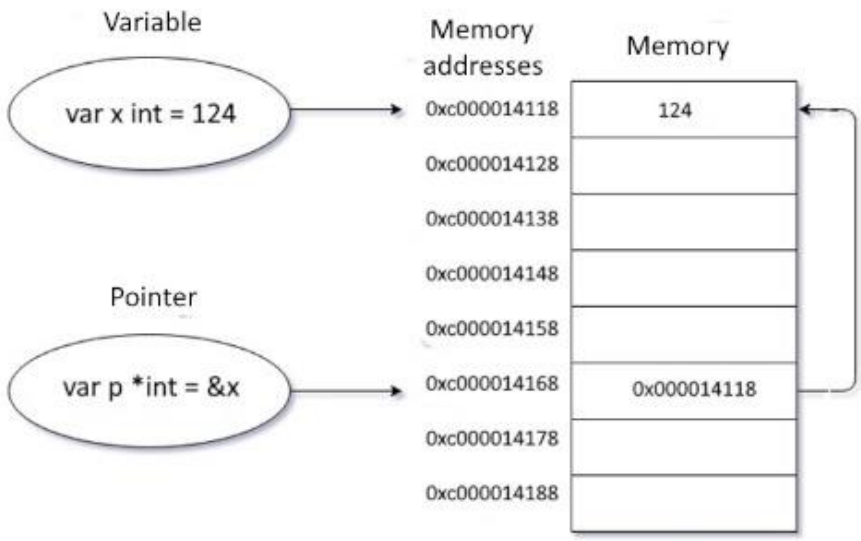


Figure 2.2. Variable, pointer, memory

The pointer is usually called a special type of variable. The main and only operators of the pointers are: the dereferential operator (*) and the address operator (&). The dereferencing operator (*) is used to declare a pointer variable and access the value stored in the address. The address operator (&) is used to return the address of the variable or to access the address of the variable index. Pointer declaration syntax:

```
var pointer_name *Data_Type,
```

where D*ata_Type* - any valid data type, for example, `var pos *string`.

To work with the pointer, it must be initialized with the memory address of another variable using the &address statement, as shown in the following example:

```
var a = 124
```

```
var s *int = &a
```

The uninitialized index will always have a zero value of <nil>. Here is an example of index initialization and processing:func main() {

```
  var z1, z2 int = 64, 128
  var p1, p2 *int
  p1 = &z1 // p1 pointer initialized
  fmt.Println("Variable value z1 = ", z1)
  fmt.Println("Address z1 = ", &z1)
  fmt.Println("Variable value z2 = ", z2)
  fmt.Println("Address z2 = ", &z2)
  // p2 pointer is uninitialized
  fmt.Println("The value saved in the variable p1 = ", p1)
  fmt.Println("The value saved in the variable p2 = ", p2)
}
```

Output:

```
Variable value z1 =  64
Address z1 =  0xc000086080
Variable value z2 =  128
Address z2 =  0xc000086088
The value saved in the variable p1 =  0xc000086080
The value saved in the variable p2 =  <nil>
```

The Dereference Statement * is used to declare a pointer variable and to access the value stored in the variable to which the pointer points:

func main() {

```
    var y = 157
    var p = &y
    fmt.Println("Variable value in y = ", y)
    fmt.Println("Variable address y = ", &y)
    fmt.Println("Value saved in p = ", p)
    fmt.Println("Value saved in  y(*p) = ", *p)
}
```

Output:

```
Variable value in y = 157
Variable address y = 0xc000014078
Value stored in p = 0xc000014078
Value stored in y(*p) = 1576)
```

b) Slice

Slices are sequences of variable length, all of whose elements are of the same type. The slice type [] T is written, where the elements are of type T.

There are three components:

Pointer: The pointer points to an array element that is accessible through a cut, which is not necessarily the first element of the array.

Length: the number of slice t elements. It cannot exceed the capacity.

Capacitance: The capacitance is usually the number of elements between the beginning of the slice and the end of the base array representing the change.

The built-in len and cap functions contain slice length and capacitance values respectively. Multiple slices can share the same base array and can refer to overlapping parts of that array. The first position of the index in the slice is always 0, and the last is the slice length minus one (len- 1). The slice declared in the same way as the array, but it does not contain a slice size. Thus, it can grow or decrease according to some algorithm.

The syntax of the cut design is as follows:

[]T or []T{} or []T{value1, value2, value3, ...value n},

here T is the element type. For example:

var my_slice [] int

The figure (Figure 2.3) shows the visual representation of the cut through its components - pointer (ptr), length (len) and capacitance (cap):

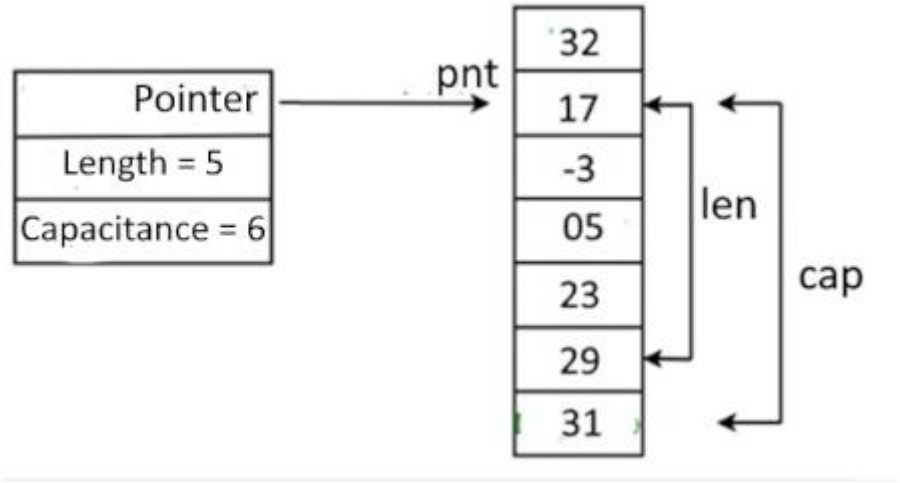


Figure.2.3. Representation of slicing in memory through components

Slices can be created from existing slices with corresponding indices (Figure 2.4):

```
package main
        import (
            "fmt"
        )

        func main() {
            var sr = []int{17, -21, 5, 62, 24, 48, 78, -43}
```

```
    sr0 := sr[2:6]
    fmt.Println("sr0 = ", sr0)
    sr1 := sr[2:]
    fmt.Println("sr1 = ", sr1) // prints [5 7 9 11 13
15]
    sr2 := sr1[:3]
    fmt.Println("sr2 = ", sr2)
}
```

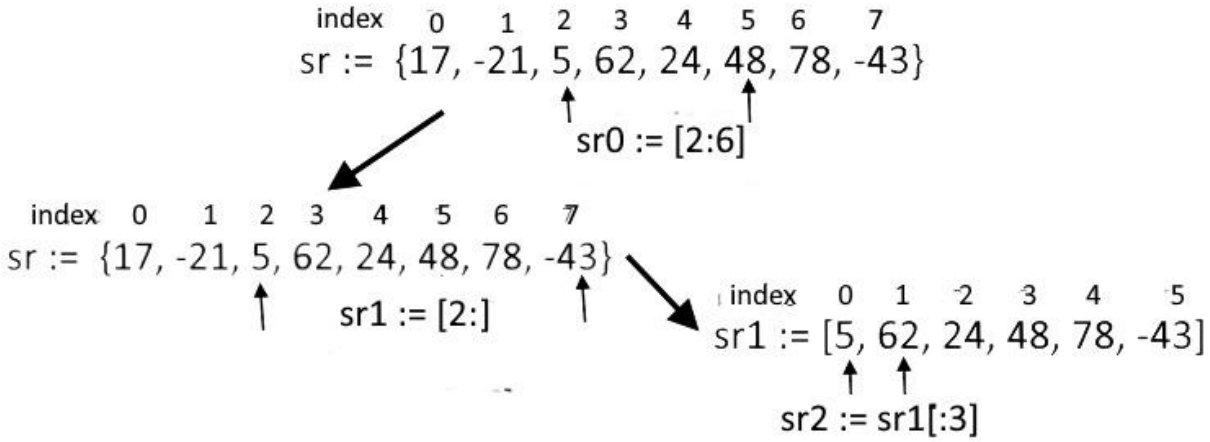


Figure 2.4. Create new slice from existing slice

It is important to note that each created cut sr0, sr1, sr2 has its own memory, that is, all created cuts have different addresses, even if they are fragments of one slice.:

sr --> 0xc000004078; sr0 --> 0xc000004090; sr1 --> 0xc0000040c0;

sr2 > 0xc0000040f0

2.7. Associative arrays (Map)

Associative arrays (later cards) are storage containers of the pair "key-value". Maps are one of the most common and useful data structures. Unlike data structures

such as arrays or cut-offs, maps provide quick and efficient searches for elements. The map does not allow duplicate keys, but may have duplicate values.

The announcement syntax of the card with the key type int and the value - string:

```
var names map[int]string
```

The card can be initialized in two ways:
Using the function *make()*: `var names = make(map[int]string);`

Using literal syntax: `var films = map[int]string;`

Here is an example illustrating both methods of initialization of the map:

```
func main() {
    names := map[int]string{0: "Cameron", 1: "Spielberg", 2:
"Lee", 3: " Donner"}
    fmt.Println(names)
    films := make(map[string]string)
    films["Cameron"] = " The Abyss"
    films["Spielberg "] = " The Goonies"
    films["Lee"] = "Jungle Fever"
    films["Donner"] = "Superman"
    fmt.Sprintln(films)
    for key, value := range films {
        fmt.Println(" \n", key, “>”,value)
    }
}
}
```

Output:

```
Cameron > The Abyss
Spielberg > The Goonies
Lee > Jungle Fever
Donner > Superman
```

2.7. Function

A function is a group of operators that share a task. With the help of functions you can repeatedly call its operator block as a unit in other parts of the program. Functions in Go can be assigned to variables, passed as an argument, and can be returned from another function. When declaring a function, you need to specify what type of variables are transferred to the function(s) and what type of data the function returns (return values). In Go, you must specify the data type for each parameter. To declare a function, the keyword func is used. The general structure of the function declaration is shown below:

```
   func name  > [Parameters] ) [Returned types]
{
   Function body
}
```

The multiplication function of two real numbers multiply(x,y) looks like this:

```
package main
 import "fmt"
func multiply(x,y float64) float64 {
    var res
    res = x * y
    return res      // return result
}
```

```
func main() {    // function performed
    var x float64 = 15
    var y float64 = 120.6

    var mult float64     // variable for result
    mult = multiply(x,y)      // function calculation
    fmt.Print(mult)     // outputting
}
```

The function can return as many values as required:

```
func main() {
    var a, b int = 60, 20
    vAdd, vSub := addSub(a, b)
    fmt.Printf("a + b = %d\n", vAdd) // prints "35 + 25 = 60"
    fmt.Printf("a - b = %d\n", vSub) // prints "35 - 25 = 10"
}

func addSub(x, y int) (int, int) { // multiple return values
(int, int)
    return x + y, x - y
}
```

An important feature of functions in Go is the ability to accept parameters by value or by reference.

2.8. Methods

Methods together with functions provide different ways of organizing program code. A method in Go is a function associated with a particular type that acts on a variable of a certain type called a receiver (recipient). The Golang language does not support the Class concept common in other languages. It is with the method that the object provides its properties, including the behavior of the object. The method must

meet the following conditions: it must be of a certain type and must be defined in the same package.

The method declaration looks like a function declaration, but it has an additional part of the parameter declaration. An optional parameter can contain one and only one parameter of the method recipient type. The recipient parameter should be enclosed in parentheses between the func keyword and the method name.

Func (*Receiver name* Type) *Method name* (Parameters)(Return type){

// Operator block

}

In Go, it is allowed to define a method whose receiver has a structure type. This receiver is available inside the method as shown in the following example. The rec receiver structure describing the rect object is declared:

```
type rect struct {
     var width, height int
 }
```

Two methods are associated with this structure that determine the behavior of the object described by the structure *rect*:

```
func (r rect) area() int {
    return r.width *r.height
}

func (r rect) perim() int {
 return 2*r.width + *r.height
}
```

2.9. Interface type

Golang is not a classical object-oriented language, that is, it does not support the implementation of the concept of "classes" and "inheritance" in a direct way. However, Go contains a very flexible concept of interfaces that provides many aspects of object-oriented programming.

Interfaces in Go provide a way to indicate the behavior of an object by a set of methods defined by the interface type. An interface-type variable can store any type of value with a set of methods, which is any superset of the interface. The concept of interface allows to organize different groups of methods applied to objects of different nature. In other words, interfaces are collections of method signatures declaring name, type parameters, and return types of methods in the interface.

The syntax of the interface is as follows:

```
type Namer interface { // Namer – interface type
Method1(param_list) return_type
Method2(param_list) return_type
```

For example, you need to create a method to determine the area of geometric shapes - circle and rectangle.

```
type shape interface {
   area() float64
   perimeter() float64
}
```

This code defines the interface for shapes and declares two functions, area() and perimeter() with a return float64 type.

```
package main
import "fmt"
// Interface declaration
type shape interface {
```

```
    area() float64
    perimeter() float64
}
// Outline declaration "rectangle"
type rectangle struct{
  length, height float64
}
// Declaration of the "circle structure
type circle struct{
  radius float64
}
// Declaring Methods for a Rectangle
func (r rectangle) area() float64 {
    return r.length * r.height
}
func (r rectangle) perimeter() float64 {
    return 2 * r.length + 2 * r.height
}
// Declaration of methods for the circle
func (c circle) area() float64 {
    return 3.142 * c.radius * c.radius
}
func (c circle) perimeter() float64 {
    return 2 * 3.142 * c.radius
}

func main() {
    r := rectangle{length: 10.0, height: 5.0}
    c := circle{radius: 5.0}

    fmt. Printf("Rectangle area - %8.1f n", r.area())
    fmt.Printf("Rectangle parimeter - %8.1f n", r.perimeter())
    fmt.Printf("Circle area - %8.1f n", c.area())
    fmt.Printf("Circle Parimeter - %8.1f n", c.perimeter())

}
```

Output:

```
Rectangle Area -  50.0

Rectangle Parimeter -  30.0

Circle area -  78.5

Lap Circummetre -  31
```

Conclusion

It should be noted once again that this section provides basic information on the programming language Golang, sufficient to understand the material set out later. Outside the section such information as Go-subroutines and channels, parallelism and shared variables, packages and tools go, reflection, etc. Next, in the relevant sections will explain in more detail the tools and constructions, as set out in this section. In addition, the recursion tool will be explained in detail because of its special value for iterative processes in data structure processing algorithms.

In addition, it should be noted that in this book, which implements the concept of hybrid programming, Linear constructions of the Go language are used as filling icons of the visual algorithmic language DRAKON. Other designs, in particular, cycles in the range or operator Select is different from Go-designs. Moreover, some DRAKON language designs are converted into other designs during the automatic generation of code. In particular, the software implementation of the Select statement is converted into a composite if-else choice operator. This is due to the desire of the authors of the DRAKON language to make the generated code more efficient, increasing its speed and saving computer memory.

SECTION 3. DRAKON - VISUAL ALGORITHMIC LANGUAGE

As the experience of teaching informatics shows, mastering the art of programming and algorithmic thinking encounters known difficulties. They are largely due to a lack of skills in formulating the problem and developing an algorithm to solve it. Even at the beginning of the mass training, programming in schools and universities was noticed that programmers can be divided into two groups: "field officer" and "staff officer". A field officer, as a rule, possessing a certain amount of programming knowledge, immediately rushes to write program code. No comment or explanation. Naturally, in the process of programming, there are unforeseen situations that lead to constant changes in the text of the program. In the end, and in many cases, the "field officer" surrenders or begins to consider the usefulness of being a "staff officer". In turn, the "staff officer" first ponders the algorithm of the program. And here to help him various methods of visualization of algorithms.

3.1. Algorithm representation visualization

In theory and practice the following forms of representation of algorithms have been formed:

- Words (written in natural language);
- Visual (images from graphic characters);
- Pseudocodes (semiformalized descriptions of algorithms in conditional algorithmic language);
- Software (programming language texts).

Visual representations of algorithms were the most effective and common in terms of visualization and understanding. Conceptually, visualization is a form of presentation of information, data, knowledge in the form of images, in order to achieve maximum convenience of perception, understanding and analysis. The visual representation allows a better view and understanding of the structural elements used in

the design of the algorithm, including the logic of their interaction. The most common means of visualizing algorithmic constructions are flowcharts and DRAKON-diagrams. The advantage of flowcharting is the availability of a corresponding State standard. However, flowcharts have recently been criticized: they claim that they are unsuitable for structural programming, difficult to formalize, they cannot be used to generate the programming code.

An alternative to flowchart imaging is DRAKON-based algorithm engineering, which results in DRAKON- diagrams []. DRAKON-diagrams are suitable for formalized recording, automatic code retrieval and execution on the computer. However, a more important aspect of DRAKON technology is its cognitive difference from flowchart technology. While flowcharts do improve programm clarity, this is not always the case, and improvements are limited. In addition, there are many cases when unsuccessful flowcharts confuse the case and make understanding difficult. In contrast, DRAKON-diagrams are much better understood (Figure 3.1.).

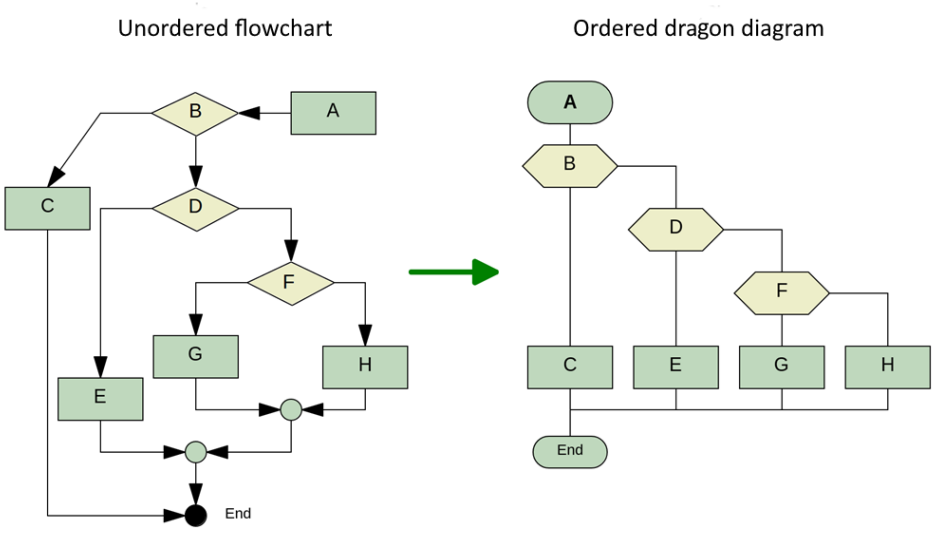


Figure 3.1. Comparison of a visual representation of an algorithm in the form of a flowchart and a DRAKON-diagram

Through the use of special formal and informal cognitive techniques, the DRAKON-diagram visually presents the solution to any algorithm or technology problem that is as complex as possible in an extremely clear, visual and understandable way. Special importance of visualization of algorithms with the help of DRAKON-diagrams acquires in the educational process, both in the study of computer science and in classes of natural as well as humanitarian disciplines.

3.2. Algorithmic language DRAKON Editor

Currently, there are several DRAKON-diagram platforms (DRAKON WEB Editor, DRAKONHUB, DRAKON.tech, "IS DRAKON". Software code generation in one of the known programming languages, including Golang, is realized with the help of the freely distributed editor DRAKON Editor [Mitkin]. The current version of the editor is DRAKON-Editor 1.31, which can be downloaded from http://DRAKON-editor.sourceforge.sounet/ (Figure 3.2.).

Figure 3.2. Archived editor file

To run the DRAKON Editor 1.31. on the Linux and Mac operating system, you must:

- extract files from DRAKON editor 1.31.zip archive;

- start the editor: /DRAKON_editor.tcl from the terminal (inside the unassembled folder.

The Windows operating system requires:

- extract files from DRAKON Editor 1.31.zip archive;

- open the executable file of the editor by double clicking on the fragment DRAKON_tcl (Figure 3.3.): .

Figure 3.3. Launch editor DRAKON Editor 1.31.

After activating the editor (DRAKON_Editor), the option "Create a new diagram file" is selected, and then a window opens to save the diagram file. As soon as you create a new file, you must save it, because in the DRAKON Editor, you do not need to save all revisions - files are saved automatically.

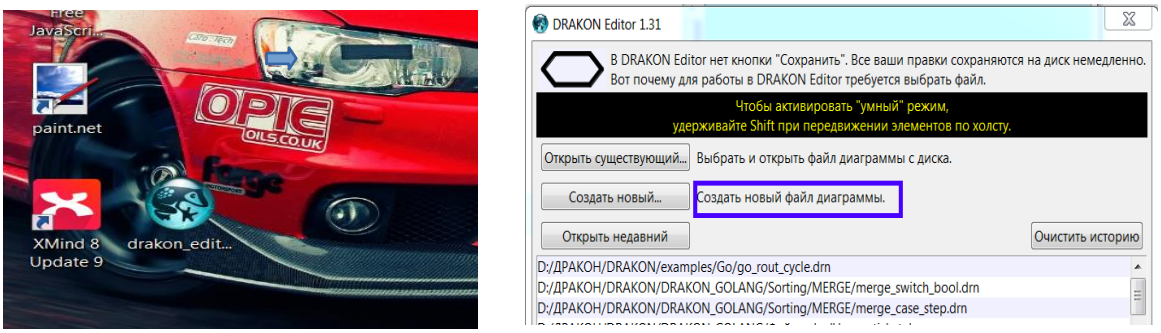


Figure 3.4. Create a new diagram file

After creating a new file, the editor interface window opens (Figure3.5.):

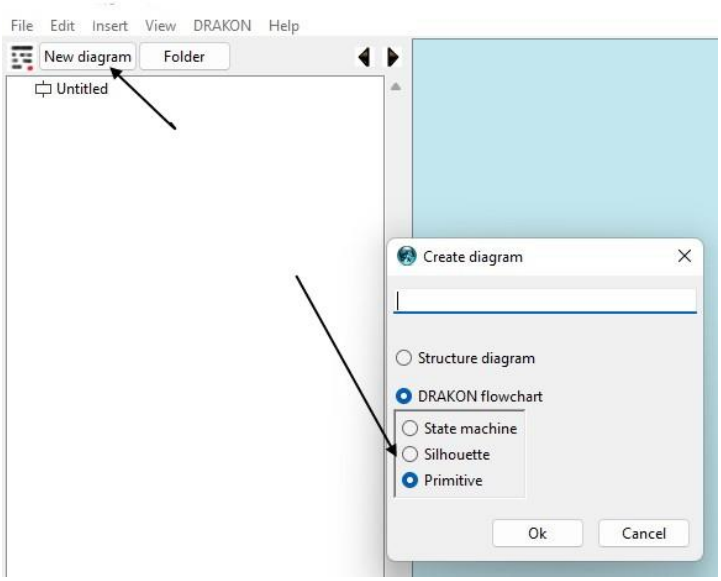


Figure 3.5. Editor interface window DRAKON WEB Editor

Next you need to click the icon «New diagram» and in the newly opened menu choose the view of the diagram (Primitive /Primitive/ or Silhouette /) (Figure3.6.).

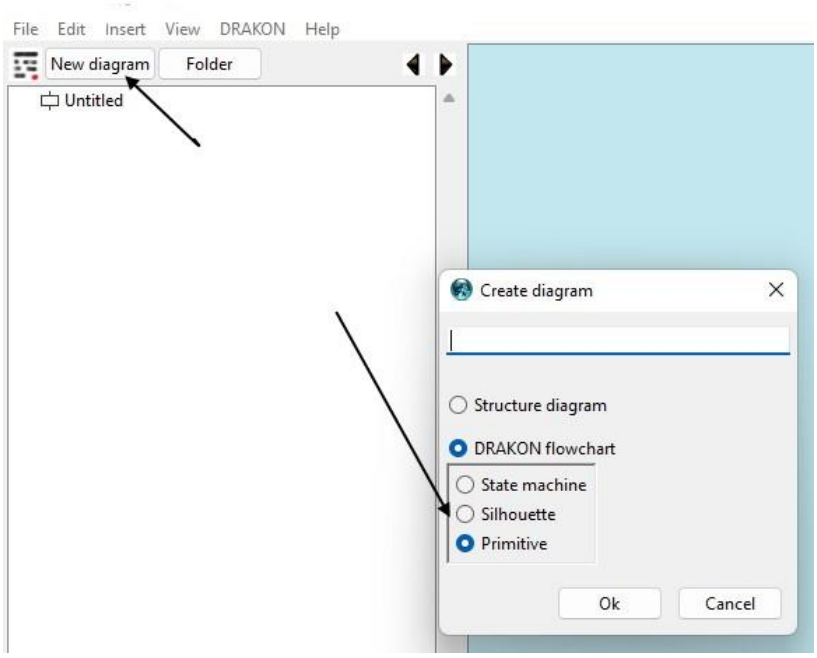
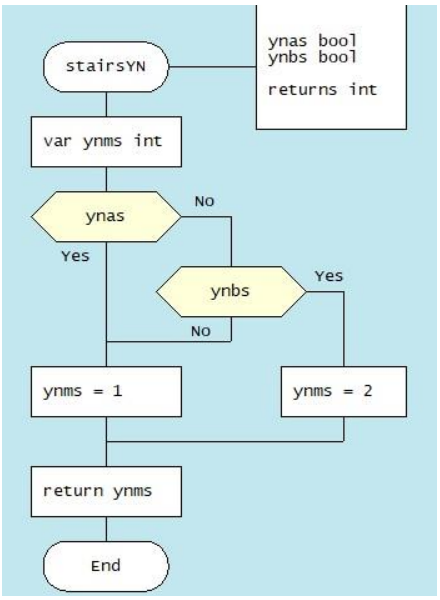
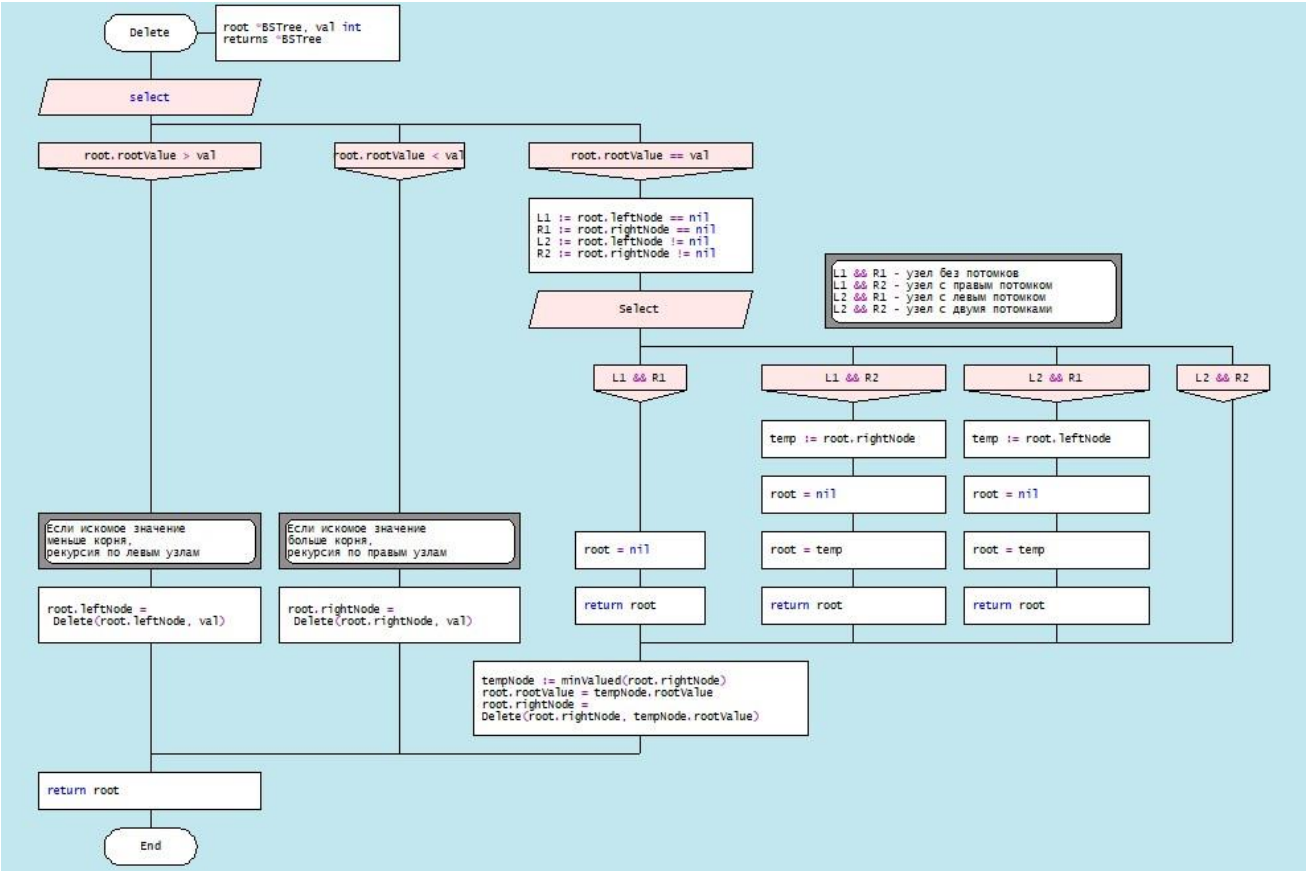


Figure 3.6. Creation a new diagram

The choice of diagram type is determined by the complexity of the algorithm (Figure3.7.):



a) Primitive



b) Silhouette

Figure 3.7. Types of diagrams: a) "Primitive"; b) "Silhouette"

To create a DRAKON-diagram, the user calls the context menu (right mouse button), and according to the algorithm chooses the necessary icons, filling them with the corresponding operators of the selected programming language (Figure 3.8).

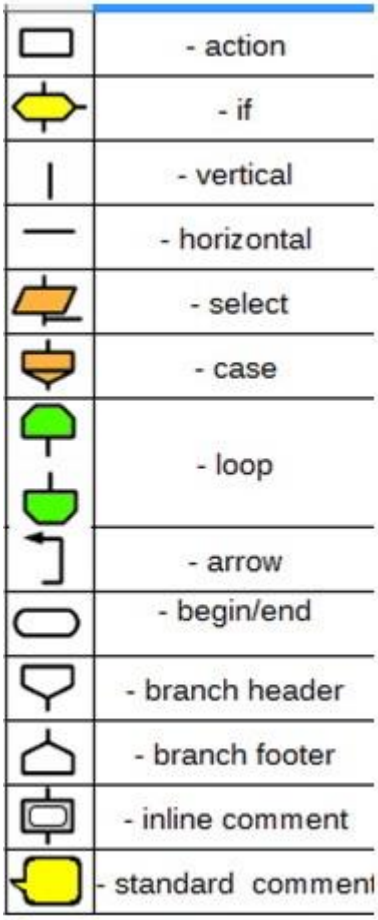


Figure 3.8. Set of DRAKON Web Editor icons

The selected icon is moved by the mouse to the vertical line (skewer - in terms of DRAKON-technology) in the right place. The icon space is then filled in with the corresponding text that displays a Snippet in a specific programming language.

Creating a DRAKON-diagram must follow certain rules:

1. Creating a DRAKON-diagram begins with a name that should reflect the purpose of the (function) algorithm and be located at the top.

2. The diagram should have only one beginning and one end. The "end" graph is placed at the bottom of the diagram (Figure 3.9):

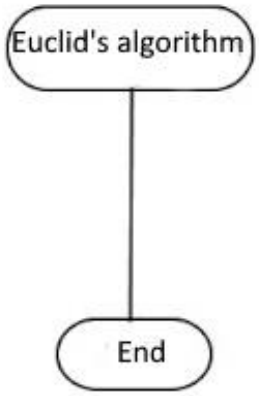


Figure 3.9. Diagram start

3. The action flow represented in the diagram should only go from top to bottom. This approach is more convenient, because in our cultural area texts are read in this way.

4. Avoid turning. The only case where lines have to change direction is where decisions are made. Turns are needed only when the algorithm requires making a choice between different actions. If there are no solutions, you need to go down. In any case, it is necessary to minimize the number of turns.

5. Crossings of lines are absolutely not allowed. All attempts to apply crossings must be prevented. However, in case of an intersection, the editor will give an error.

6. A top-down action prevents the use of arrows. The only exception is a loop of type while (Figure3.10.):

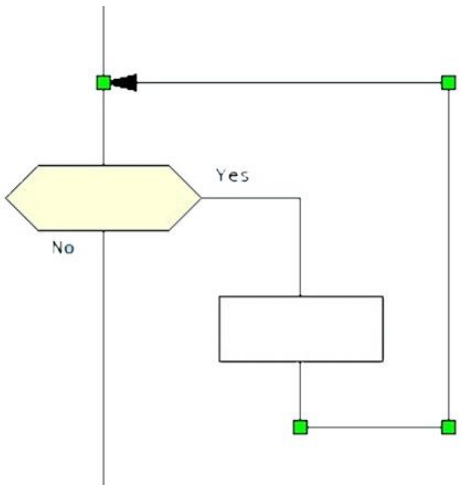


Figure 3.10. Data flow from bottom to top (While loop type)

7. When creating a DRAKON-diagram, only straight vertical and horizontal lines should be used, as straight lines are easier to understand than curves.

8. It is necessary to keep the same distance between adjacent icons. The DRAKON WEB Editor allows you to "cleverly" optimize the diagram using the "Shift + mouse" keys.

9. Branching is done only to the right. Branching to the left should be excluded. Following this rule significantly increases the predictability and uniformity of charts.

The reader may get the impression that creating DRAKON-diagrams is a very complex process. It should be recalled once again that in the editor DRAKON WEB Editor everything is quite simple and clear. The editor will not allow the violation of the stated rules, which is checked by the verify option (the Figure shows an error - there is no space between the icons (Figure3.11)):

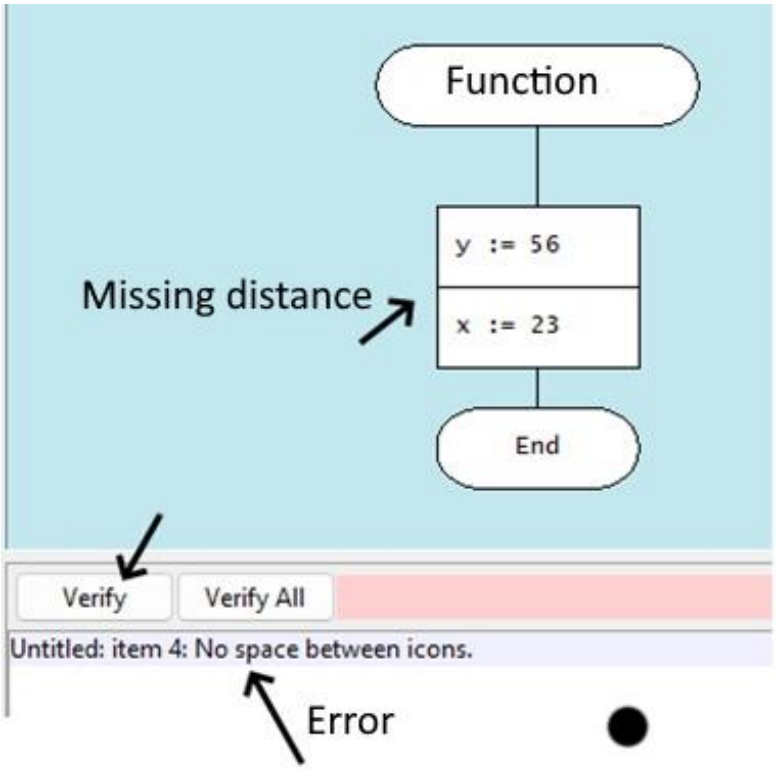


Figure 3.11. Work of the editor, indicating the error

The DRAKON WEB Editor implements a complete set of rules of visual syntax, which frees the user from the need to remember the syntax rules in detail. DRAKON WEB Editor creates only correctly created diagrams and does not allow diagrams in which the syntax is broken to generate program code.

3.3. Basic structures of DRAKON WEB Editor algorithms

The logical structure of an algorithm can be represented by a combination of three basic structures: linear, branched and cyclic. The typical feature of the basic structures is the presence of one entrance and one wig.

3.3.1. Linear structure

The linear structure of an algorithmic process implements operations that are performed sequentially in order of writing. A typical example of such a process is a standard three-step computing scheme:

a) Input data; b) formula calculation; c) output result. The graphical representation of the basic element of the linear structure in the software products of the DRAKON line has the form of simple rectangles, For example, in the algorithm for finding the minimum and maximum values of the array, minim and maxim variables are given the value of the zero element of the array (Figure 3.12).

Figure 3.12. Linear Structure of DRAKON-diagram

3.3.2. Basic typees of branched structures

The branched structure contains at least one condition check, which ensures the transition to one of the possible solutions. Each option leads to a common output, that is, the algorithm will continue regardless of which path is chosen. The branching structure exists in two main options:

a). The DRAKON Snippet of the construction "if (condition), then (action) otherwise (action)", that is, in programming languages is the statement if...else. An example of an if-else construction in a DRAKON-diagram in an algorithm for finding minimum and maximum array values (Figure 3.13):

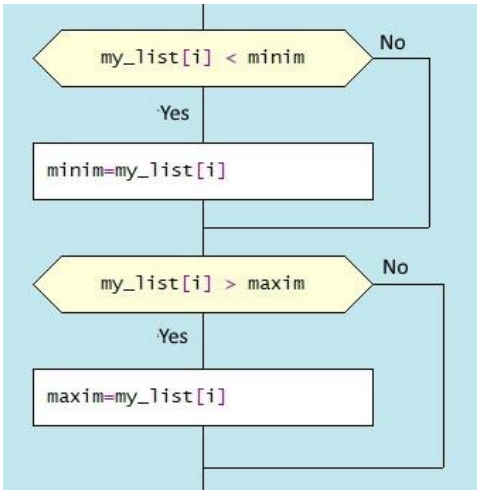


Figure 3.13. Diagram snippet with “if-else”

b). DRAKON Snippet Construction «Select»

The values to which the expression in the "Choose" graph will be compared are covered in the "Option" graphs. If there is no text in the extreme right, it means "all other values". Below is an example of how to branch a sorting algorithm (merge sort) (3.14).

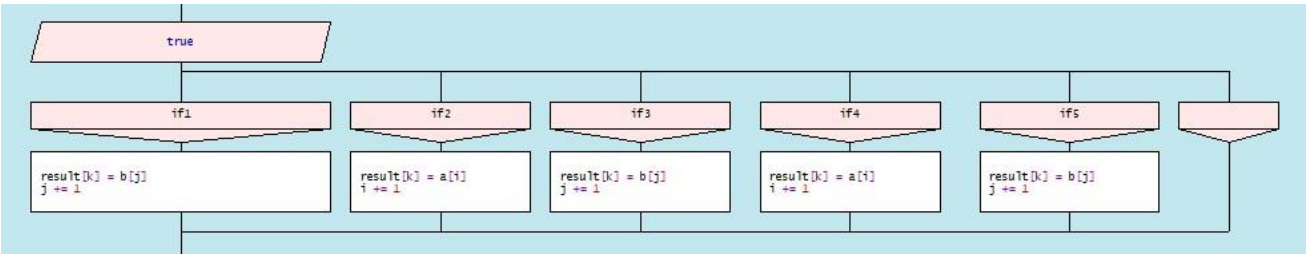


Figure 3.14 Branch selection snippet (Select statement)

3.3.3. Base loop constructions

The loop structure involves repeating the same sequence of actions repeatedly. The number of repetitions is determined by the input data or task conditions. Loop structures include, first of all, the construction "Loop-For" (C-style loop, "Loop for each", composite constructions "Loop-With_Arrow" ("Do-Chek loop") and "Do-Chek-Do").

a). "Loop-For" consists of three parts. In the first part, the loop initialization is fixed. In the second one, the loop completion condition is checked. If true, the body operators of the loop are executed until the expression becomes false. If it is false, the loop ends and the control is passed to the next operator. In the third part, the loop parameter increases. The snippet of the DRAKON-diagram with the "Loop-For" design has the appearance (Figure 3.15):

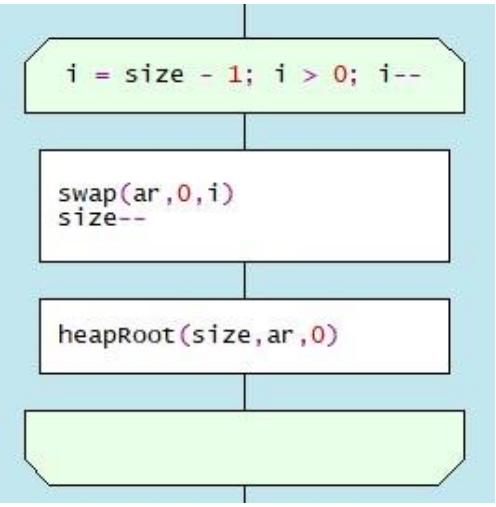


Figure 3.15. DRAKON construction "Loop-For”

b). Loop "foreach" executes the operator or block of operators for each array element or data list (Figure 3.16).

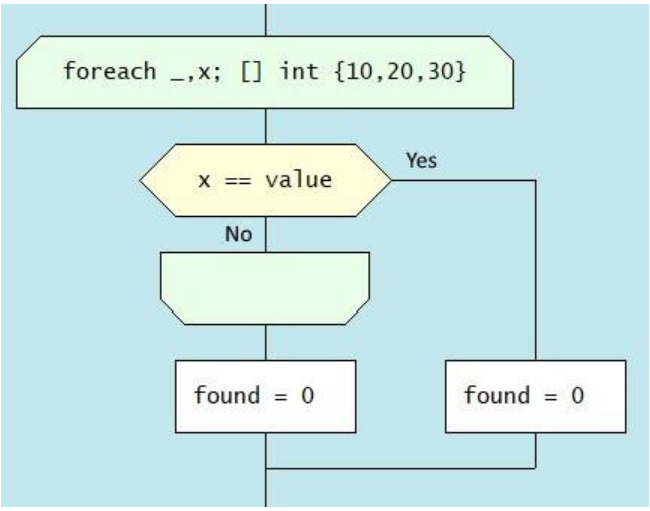


Figure 3.16. DRAKON construction "Loop for each”

c). Example of composite construction "Loop With Arrow" (Figure 3.17)

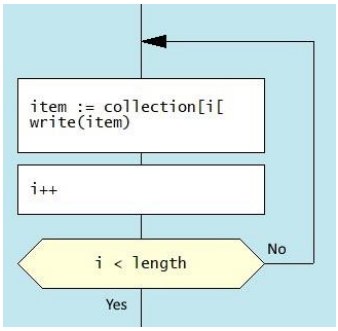


Figure. 3.17. DRAKON construction "Do-Chek loop ”

d). Example of composite construction “Do-Chek-Do” (Figure 3.18)

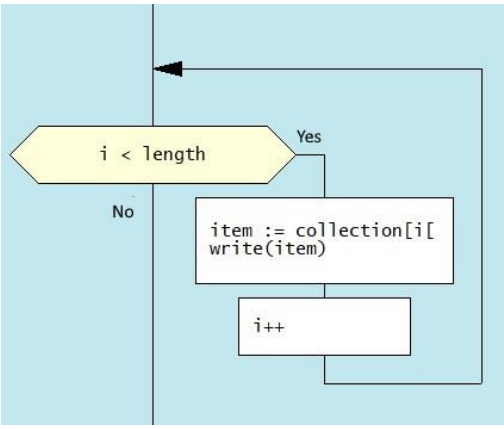


Figure 3.18 Diagram snippet “Do-Chek-Do”

3.3.4. DRAKON-diagram example

Let us show with a concrete example the construction of the diagram of the algorithm for calculating the most common divisor by the Euclid method. The largest common divisor (GCD) is a number that divides without remainder two numbers and divides itself without remainder by any other divisor of the given two numbers. Simply put, this is the largest number by which you can divide without residue the two numbers for which you are looking for GCD.

The algorithm for finding GCD by Euclid division is as follows:

1. The larger number is divided into the smaller number.
2. If divided without remainder, then the smaller number and is NLD (should leave the cycle).
3. If there is a remainder, the larger number is replaced by the remainder of the division.
4. Moving on to paragraph.

The DRAKON-diagram of the most common divisor algorithm is shown in Figure 3.19.

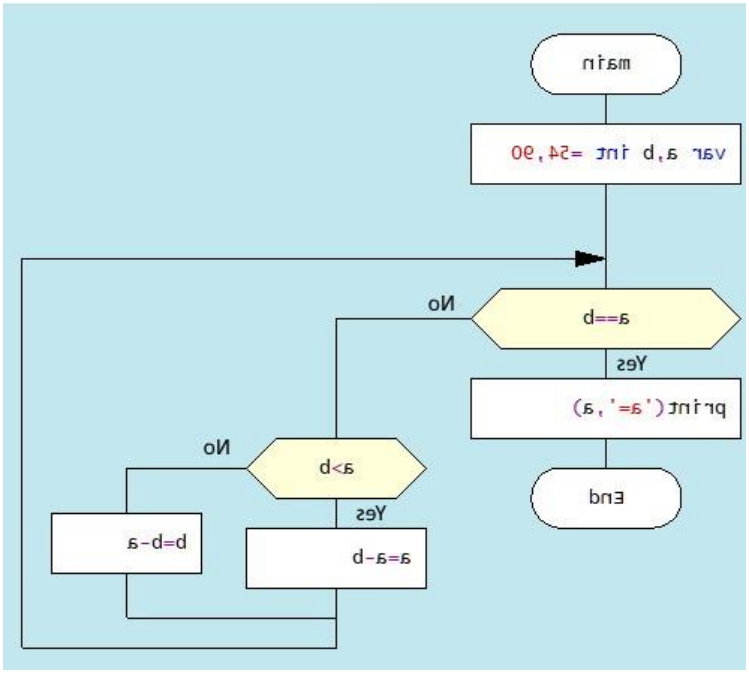


Figure 3.19 Euclid algorithm DRAKON-diagram

3.4. From DRAKON-diagram to program code

Each DRAKON-diagram corresponds to the program module. Figure 3.20 shows the structure of the array sorting algorithm consisting of a series of separate modules (DRAKON-diagrams) and the main DRAKON-diagram containing the main program's main Go design:

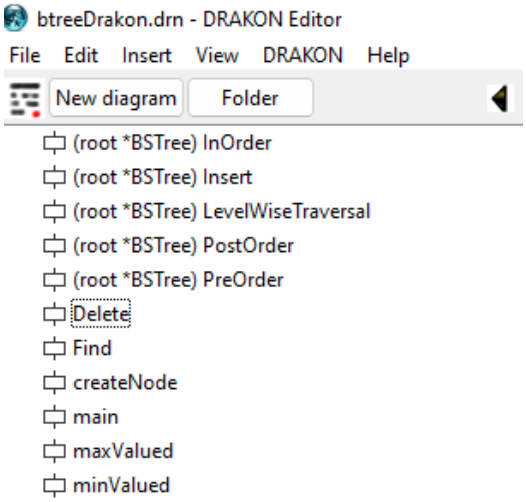


Figure 3.20. Tree processing program structure

As described above, the Golang programming language program structure consists of packages. The language designs included in the package are included in the File/Description section of the DRAKON WEB Editor interface (Figure 3.21.):

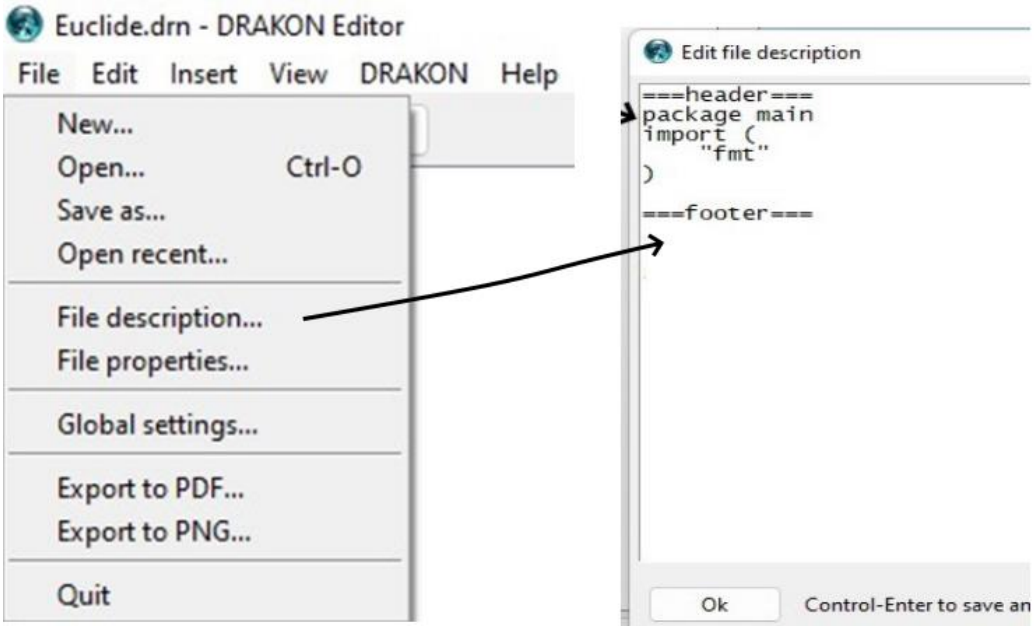


Figure 3.21. Setting of description of program via File/Description

In the field between labels *===header===* and *==footer===* you should have language designs available in each program module, for example, global variables.

Next, the user should specify the programming language to use. To do this, open the File

properties option (Figure 3.22):

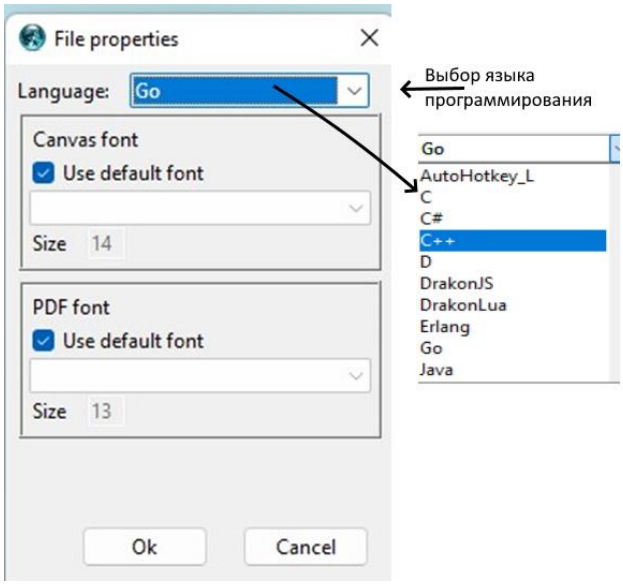


Figure 3.22. Choosing a programming language

The final stage of development is the generation of the program code (Figure 3.23):

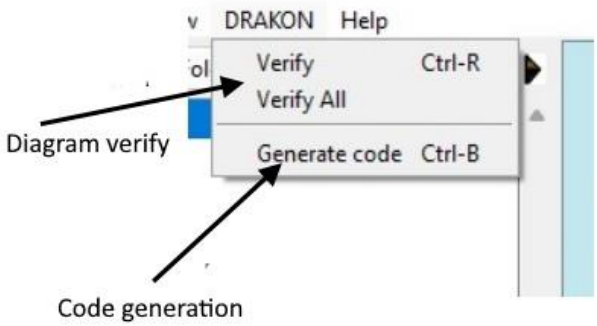


Figure 3.23. Option «Code generation »

The source code in the format `.drn` and generated code in the format `.go` are saved in the same directory.

3.5. Execution of the generated code

The generated code can be opened in one of the integrated development environments (IDE): Visual Studio Code, VIM, Eclipse, Atom, Sublime Text and a

number of others. This book uses the IDE Visual Studio Code (VSC), the description of which is not included here. The VSC programming environment is presented in Figure 3.24:

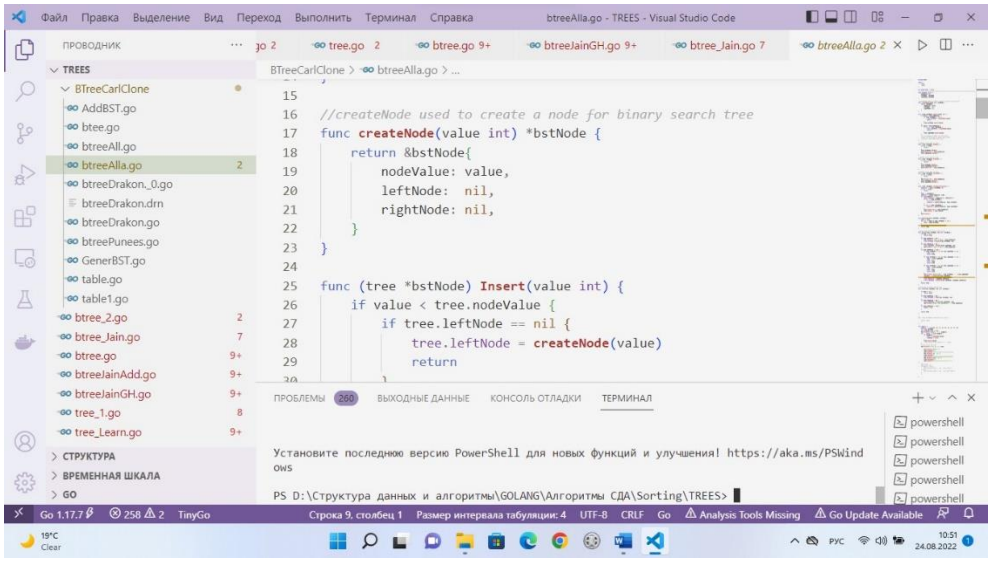


Figure 3.24. Visual Studio Code interface with the Golang-program

Almost as part of the hybrid approach debugging program often have to be carried out in parallel in two environments: DRAKON WEB Editor and Visual Studio Code. Naturally, in case of graphical syntax errors a DRAKON-diagram code generation will not be made, at the same time the presence of semantic errors in the algorithm code will be generated, but the results will be incorrect.

The debugging process of the generated code is recommended to be carried out with the help of the appropriate programming environment toolkit. After debugging is complete, all corrections must be made to the corresponding graphs in the DRAKON-diagram. In any case, it is always necessary to ensure that the contents of the graphs correspond to the diagram and program code in the in the relevant IDE.

SECTION 4. IMPLEMENTATION OF BASIC ABSTRACT DATA TYPES IN GOLANG

4.1. Data Structuring Features

Recall - the abstract data type is a mathematical data model and various operators defined within this model. Data processing algorithms can be developed in terms of ATD, but to implement them in a particular programming language, it is necessary to find a way to represent ATD in terms of the data types and operators (methods) supported by this programming language.

In practical terms, the data structure is a collection of data structured in such a way as to ensure their efficient use by users. This requires a certain ordering of data, primarily at the level of storage in computer memory. Reducing space and increasing its organization in conjunction with reducing the time complexity of different tasks is the main objective of working with data structures.

Considerable experience in the development of computer technology and computing technology has made it possible to classify data structures into different categories. First, simple and integrated structures are distinguished by complexity. The criterion of simplicity is the indivisibility of this, that is, in a computer implementation - a simple chain of bits. Simple, basic structures include variables of different types: integers, real numbers, logical, string.

Integrated (composite, complex) are data structures whose component parts are other data structures, including simple and integrated. Many basic integrated data structures are predefined by a specific programming language: arrays, slices, structures, etc. Such structures can be created by users for a specific problem, using basic integrated structures.

In terms of data representation, there are two types of structures: **logical and physical**. The logical structure is an abstract scheme of data representation envisioned

by the user or programmer. The physical structure, on the other hand, refers to the specific arrangement of data in the memory of the computing machine. In most cases, the logical and physical structures of the same data do not coincide. In a logical (abstract) structure, data is typically arranged adjacent to each other, whereas in a computer implementation, this data may be located in different memory areas.

An important feature of a data structure is the presence of connections between the elements of the structure. On this basis, they distinguish between incoherent and coherent. Disconnected structures are characterized by the absence of connections between the elements of the structure, while connected structures are characterized by the presence of connections. Disconnected structures include arrays, strings, stacks, queues; to connected - linked lists.

In many cases, when working with data, such characteristics as variability can play a role, that is, a change in the number of elements and (or) connections between the elements of the structure. On this basis, static, semi-static and dynamic structures are distinguished. Static include arrays, sets, records, tables, to semi-static - stacks, queues, trees, to dynamic - linear and branched linked lists, graphs, trees.

According to the nature of the ordering of the elements in the structure, a distinction is made *between linear and nonlinear* data structures. Linear structures, depending on the nature of the relative arrangement of elements in memory, are divided into structures with a sequential distribution of elements in memory (vectors, strings, arrays, stacks, queues) and structures with an arbitrary connected distribution of elements in memory (singly linked and doubly connected linear lists). Nonlinear structures include multi-connected lists, trees, graphs.

One of the defining characteristics of data structures is the way they access data. What is important in the access method is the search mechanism - an algorithm that determines the access path that is possible within a given memory structure and the number of steps along this path to find the required data. Among data access methods, there are two main groups: sequential and direct *Sequential access* means that a group

of items is accessed in a predefined ordered sequence. An example of sequential access is a singly linked list. *Direct access* to the various elements of the data structure is provided by specifying the unique address of these elements.

Finally, it is worth noting such a feature of data structures as *homogeneity*. Homogeneous structures are structures that contain many simple data of the same type (numeric, logical, string, etc.). Heterogeneous structures combine different types of data. Examples of homogeneous structures are arrays, slices, and stacks. Heterogeneous structures include records and sets.

The presence of a large number of features of data structures predetermined various attempts to classify them. From a programming point of view, the main features are: linearity/non-linearity, data access, homogeneity/heterogeneity of data (Table 4.1.).

	Array	Slice	Stack	Queue	Linked List	Map	Tree	Graph
Linear	X	X	X	X	X			
Non-linear						X	X	X
Direct	X	X						
Sequential			X	X	X	X	X	X
Homogeneous	X	X	X	X	X	X	X	X
Heterogeneous					X	X	X	X

This section covers the Go-implementation of linear abstract data such as array/slice, linked lists, map, stacks, and queues. Nonlinear data structures (map, tree, graph) will be discussed in the corresponding sections.

4.2. Linear data structures

4.2.1. Linear data with direct-access structures

Linear data structures are structures in which data elements are arranged in sequence. Linear structures can be distinguished by the way the individual elements of a data collection are accessed.

a). Array

An array is a collection of data, belonging to the same type. For example, the collection of integers 24, 12, 36, 6, 47, 11 forms an array (Figure. 4.1.).

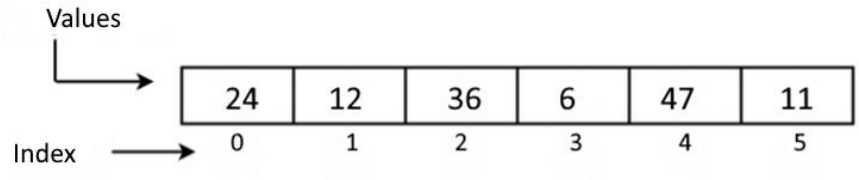


Figure 4.1. One-dimensional array

It is not allowed to mix values of different types, such as an array containing both characters and integers. There are different ways to declare arrays:

var array_name[]Type

or

var array_name[length]Typle{item1, item2, item3, ...itemN}

In addition, in the language Go arrays can be declared in the shortened form:

array_name:= [length]Type{item1, item2, item3,...itemN}

In Go, you can create a multidimensional array using the following syntax:

array_name[Length1][Length2]..[LengthN]Type

However, array data is rarely used in Go. Much more convenient is a collection of data like a *slice*.

b) Slice

A slice is a variable-length data collection that stores elements of a homogeneous type. A slice can be thought of as a slice of an array. Slice syntax (T - type) of data:

[]T

or

[]T{}

or

[]T{value1, value2, value3, ...valuen}

A slice has three components: *pointer, length, and capacity*. To create a slice in this form, use the *make* function. For example, in Figure 4.2. creating a parentSlice looks like this

:*parentSlice = make([]int, 20, 20)*

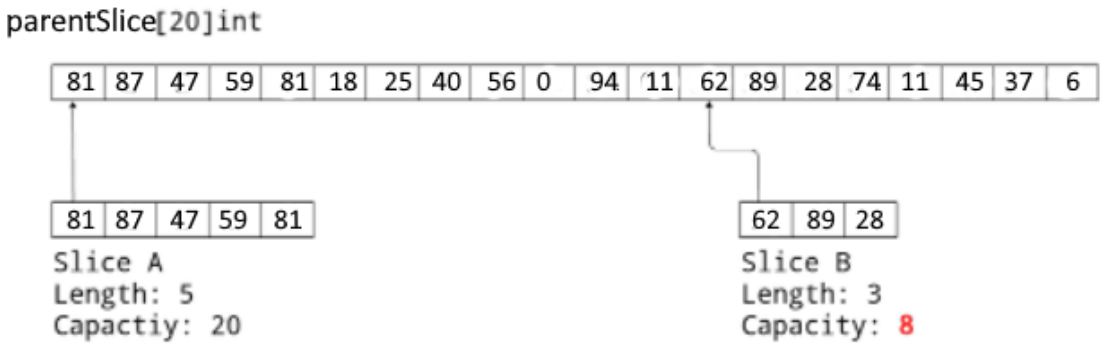


Figure 4.2. Create different slices of *SliceA* and *SliceB*

The pointer indicates to the first item of the array accessible through the slice (which is not necessarily the same as the first item of the array). Length is the number of slice elements; it cannot exceed the capacity, which is typically the number of items between the start of the slice and the end of the underlying array. These values are returned by the built-in functions *len* and *cap*. Multiple slices can be created from a single array with different pointer, length, and capacity values. In Figure 4.2. the primary slice/array *parentSlice* is shown, on the basis of which two slices of different lengths are created, starting from different places. To do this, use the notation *sliceA := parentSlice[:5]* and s*liceB := parentSlice[12:15]*. Here, the capacity of *SliceA* is 20 and *sliceB* is 8 because Go defines this value as the difference between the length of the primary array (20) and the index of the first item of *Slice B* (12).

The following functions are used for work with slices:

1. The built-in *append()* function is used to add elements to the slice. If the basic slice size is insufficient, a new slice is automatically created and the old slice is copied.
2. The *len()* function returns the number of elements present in the slice.
3. The *cap()* function returns the capacity of the base slice.
4. *Copy()*, the content of the original slice, is copied to a slice of the destination.

4.2.2. Linear data with nodirect-access structures

Golang supports data collections as structures consisting of a set of multiple data types (fields) represented as a single entity. In Golang the structure is implemented with the help of data type *Struct*:

```
package main
import (
    "fmt"
```

```
)
type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}
func main() {
    // creating a structure with specifying field
    emp1 := Employee{
        firstName: "Peter",   age:      35,      salary:   20000,  lastName:
"Wolf",
     }
    // creating a structure without specifying field names
    emp2 := Employee{"Nick", "Smith", 49, 35000, }
    fmt.Println("Employee 1", emp1)
    fmt.Println("Employee 2", emp2)
}
```

Resuit:

```
Employee 1 {Peter Wolf 35 20000}
Employee 2 {Nick Smith 49 35000}
```

Note. When you type the order of the fields on the monitor *(emp1)*, it is not necessary to match the order when you create the structure. In the second case of creating a *structure (emp2)* the order must coincide.

Golang allows to create pointers to a structure.

```
package main
```

```
import (
    "fmt"
)

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
    emp3 := &Employee{
        firstName: "Sam",
        lastName:  "Shaffer",
        age:       55,
        salary:    22000,
    }
    fmt.Println("First Name:", (*emp3).firstName)
    fmt.Println("Age:", (*emp3).age)
}
```

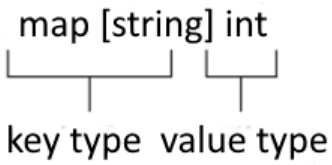
Output:

```
First Name: Sam
Age: 55
```

b). Map

Maps are collections that use unordered key-value pairs, where keys are unique identifiers associated with each value in the map. Maps are especially effective in data

retrieval algorithms. Map keys can be of almost any type, unlike arrays and slices, which use sequences of numbers for keys. The type for keys and calculations in Go needs to be refined. To declare a map with keys of type string and values of type int, the following syntax is used:



`A map can be created using the `make:`

```
employeeAge := make(map[string]int)
employeeAge["P.Wolf"] = 45
```

Syntax	Meaning
var mapName map [KeyType] ValueType	to declare a map
var mapName = map [KeyType] ValueType {}	declare and assign an empty map
var mapName = map [KeyType] ValueType {key1: val1, key2: val2}	declare an empty map
mapName: = make (map [KeyType] ValueType)	declare and initialize a default size map
mapName: = make (map [KeyType] ValueType, length)	declare and initialize a default size map

We will demonstrate various operations related to working with maps using a specific example. Our objective is to create a map containing data about employees and their ages (`employeeAge`).

employeeAge := map[string]int{}

Initialization of this map:

```
employeeAge = map[string]int{
  "P. Wolf": 45,
   "V. Smith": 47
}
```

Addition of a new entry:

```
employeeAge["R. Tompson"] = 34
```

Output to the monitor:

```
map[S. Bondes:34 D. Levis:47 G. Ivens:45]
```

Other operations with the map.

Nonlinear data structures (map, tree, graph) will be discussed in the corresponding sections.

```
age := employeeAge["T. Marks]
```

Delete a key-value pair:

```
delete(employeeSalary, "Tom")
```

To verify that the key exists:

```
val, ok := mapName[key]
```

If the key exists, the *variable val* will be the value of the key in *the map*, and the variable *ok* will be *true.* If the key does not exist, the *val* variable will receive a zero default value of type *value*, and the *ok* variable will be *false.*

4.3. Linear sequential access structures

Sequential access to data means that only one element of the structure can be accessed at any given time, and the elements are accessed in a certain order. Classical examples of a sequential access structure are a singly linked list, a stack, and a queue..

4.3.1. Linked List

A linked list is a dynamic data structure, the elements of which are called nodes, consisting of two parts: content and reference. The content part for storing a data value can be one of the basic data types, such as an integer, a floating point

number, a string, or some other data type. The reference part is a link that is used to store the address of the next element in the list.

Linked lists find their way into many computing tasks, from organizing operating systems to creating playlists. In particular, they are useful when processing the file system: sometimes it is difficult to find disk space to hold the entire file, so it can break into scattered fragments. To organize work with these fragments, a linked list of sections is formed in which file fragments are stored on disk. In this they differ from arrays or slices, where all elements are located adjacent to each other.

There are different types of linked lists. First of all, they differ in the number of fields (simply connected and biconnected lists) and in the way the elements are connected (linear and cyclic). In the simplest case of a simply connected list, each node (except for the last node) contains a link (pointer) to the next node of the same list. The reference part contains the address of the next node. The reference part of the last node contains the value nil (Figure 4.3).

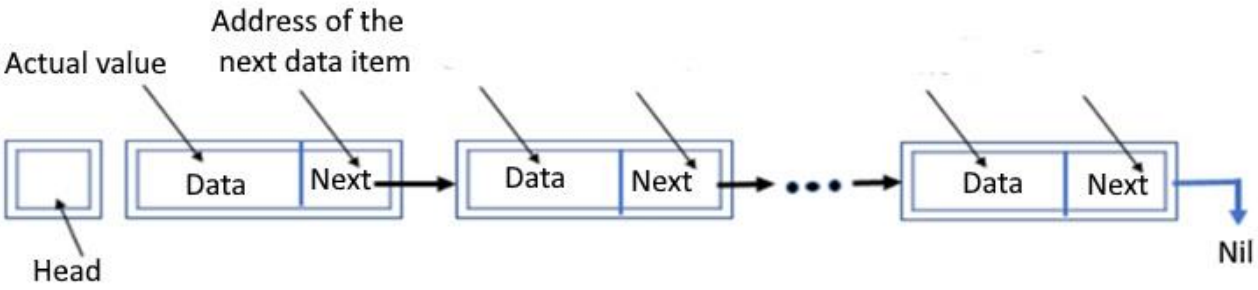


Figure 4.3. Simply linked list structure

The structure of an individual linked list node can be described by the struct data type as follows:

type *Node* struct {

data string
*nextNode *Node*

```
}
```

The linked list structure contains the *list length, head node* and *tail node*:

```
type LinkedList struct {
   len      int
   headNode *Node
}
```

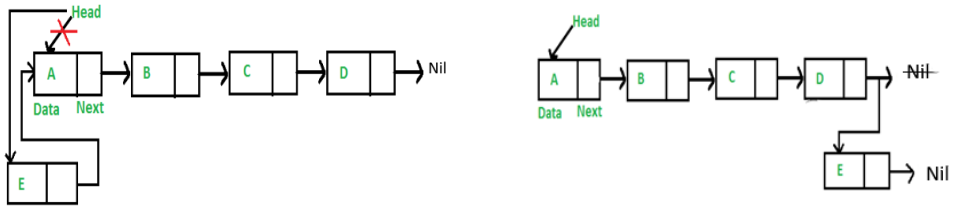
The "len" field in the linked list structure contains its length. The "headNode" field stores the memory address of the header (the first node in the linked list). Initialization (instance creation) of a structure of type LinkedList is performed as follows:

```
var ll LinkedList = LinkedList{}
```

The Golang language provides the ability to implement various operations with linked lists, among which the main ones are:

- • inserting an element into the list;
- • removal of an element from the list;
- • search for an element in the list;

Consider the operation of inserting an item into a linked list (Figure 4.4):



a) b)

Figure 4.4. Inserting a new node at the beginning

a) and at the end; (b) of the linked list

Three methods are used to insert new nodes: at the beginning of the list - *pushFront(val)*, at the end of the list - *pushBack(val)* and at the specified position *pushVal(nodeVal,val)*, where the val parameter is the value of the list element of one type or another (in the given string), *nodeVal* - the value of the list element after which the new node is inserted. Let us consider sequentially the algorithms of all three methods. Removing nodes implements the *removeVal(val)* method

a) Method *pushFront* – inserting a new node at the beginning of the list

This method implements the process of inserting a new node at the top of the linked list. The algorithm of the method is presented in the form of DRAKON-diagram with the subsequent automatic generation of program code in the DRAKON WEB Editor (Figure 4.5).

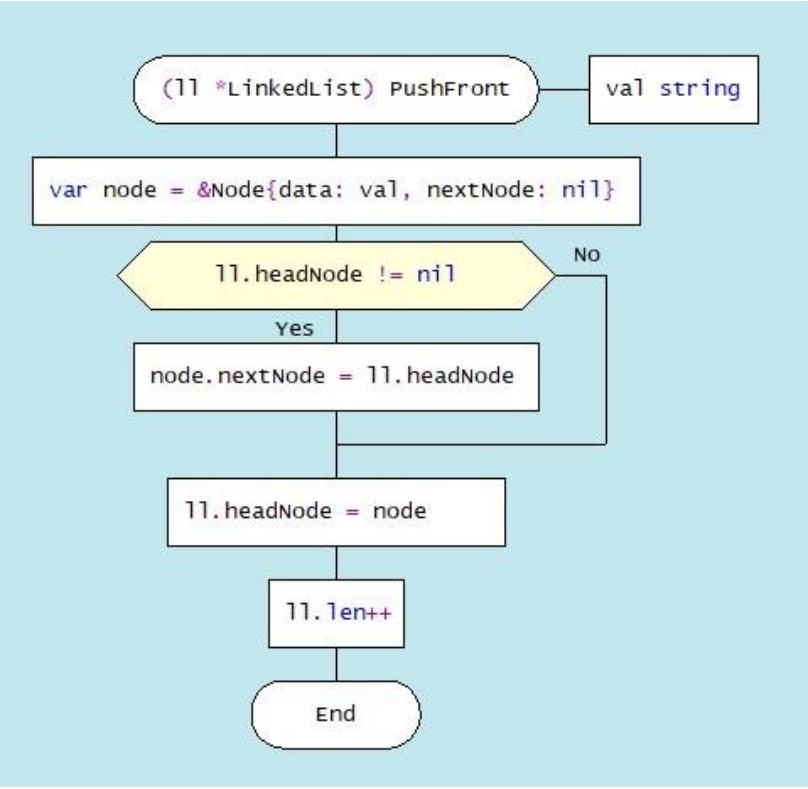


Figure 4.5. DRAKON-diagram of rhe method *pushFront(val)*

b) The method *pushBack* – inserting a new node at the end of the list

Inserting a new node at the end of a singly linked list is implemented using the *pushBack(val)* method. The algorithm for inserting a new node at the end of the list consists of the following steps. A new node *(newNode)* is created, to which a parameter *(val)* is passed. If the list is empty, then the new node will be both the first and the last node in the list. If the list is not empty, then all nodes are traversed to the end of the list and a new node is added to the end of the list. DRAKON-diagram of the method *pushBack(val)*.

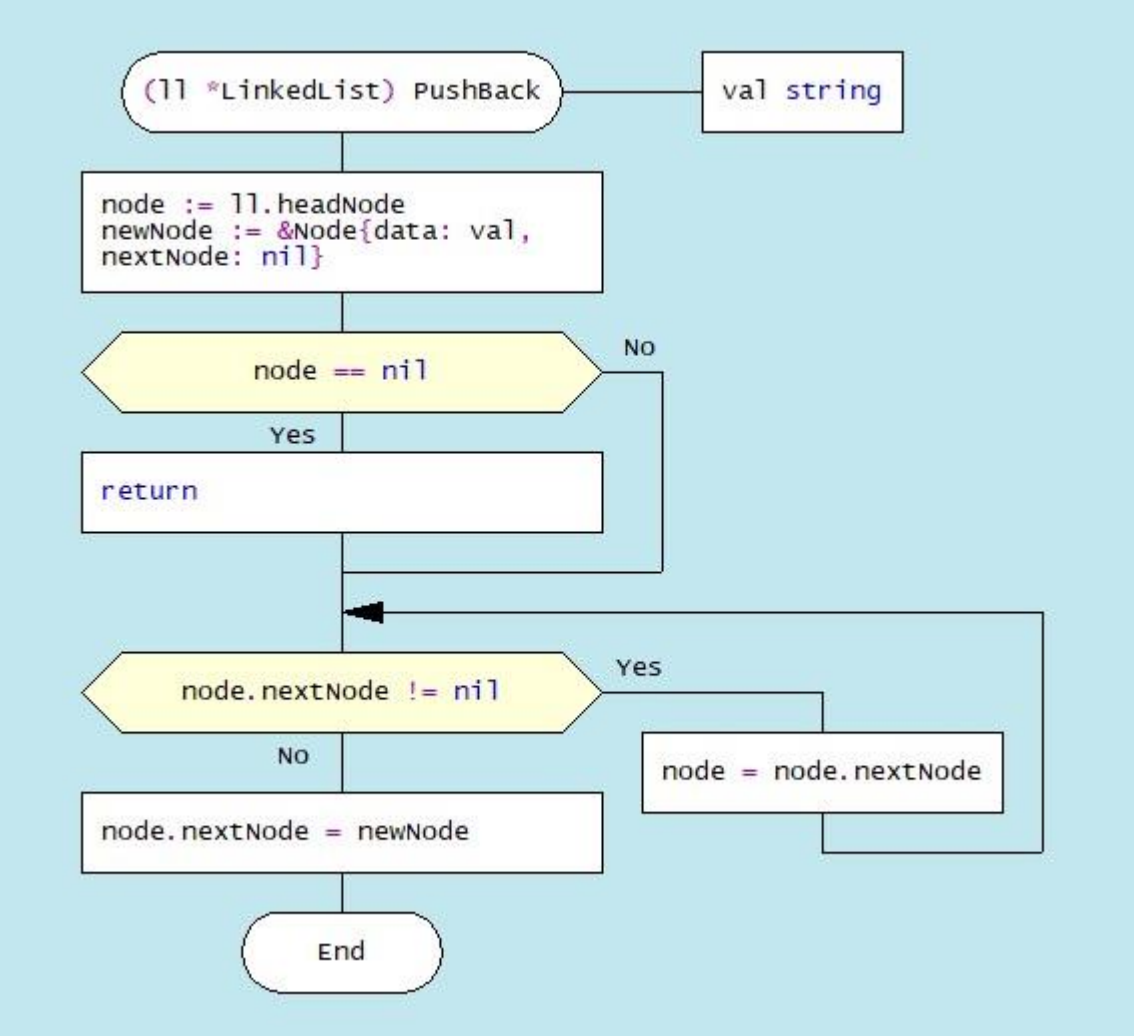


Figure 4.6. DRAKON-diagram of node insertion algorithm to the linked list end

c) The method *pushVal(nodedata,val)* - insertion after specified node

The third method *pushVal(nodeVal,val)* implements the process of inserting a new node with the *val* parameter after the node with the *nodeVal* value. The method algorithm first refers to the *NodeWithNode(nodeVal)* module, which defines the node after which the new node should be inserted. In our case, the word "probably" is inserted into the list created above after the node with the content "This". The dragon diagram of such an insertion algorithm and the corresponding program code are shown in Figure 4.7.

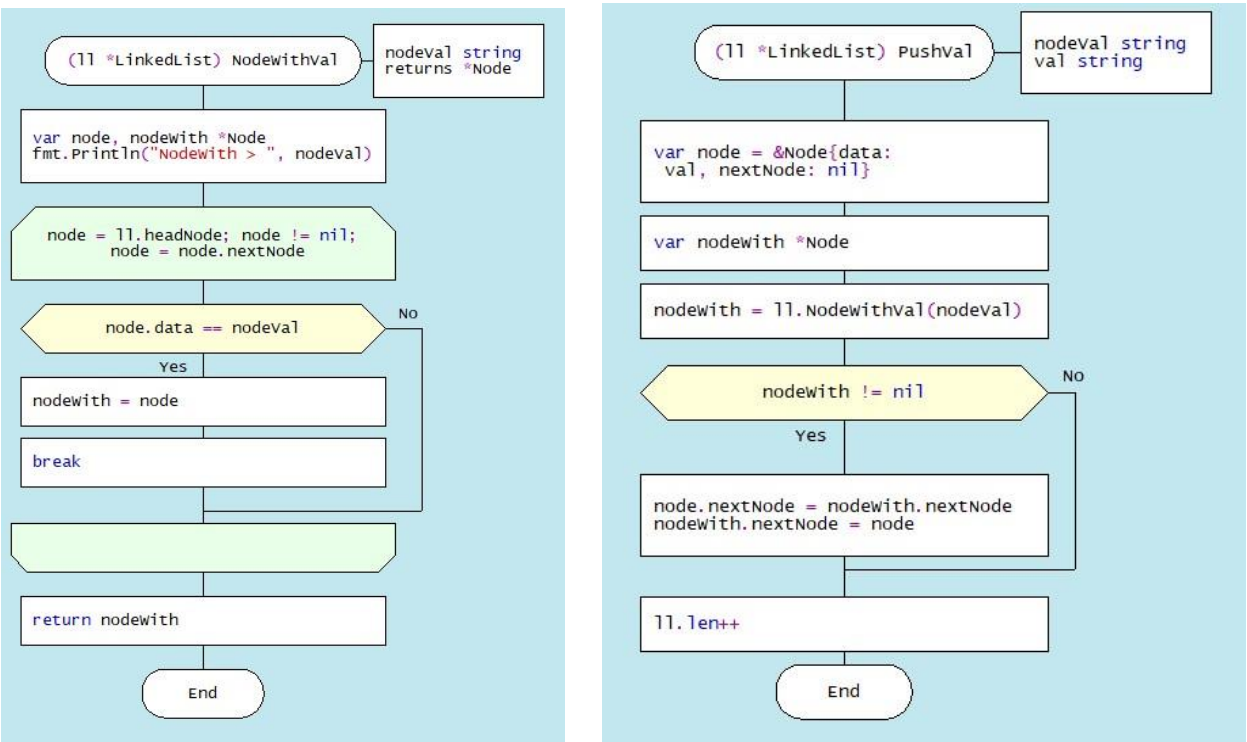


Figure 4.7. DRAKON-diagrams of node insertion algorithms NodeWithVal and pushVal

d) The method removeKey method (val) – removing the specified node

The basic methods of working with linked list nodes include methods for deleting one or more nodes or deleting nodes by condition. Consider the algorithm for deleting a node by a specifided value. Removing a node from a linked list after the

specified value *(k)* is performed using the *removeVal(val)* method, the parameter of which is the key *val(* the data field of the Node list node structure). To delete a list node with the key *val*, you must first locate the node. In this module, the first node (*ll.head.data == key* is checked first) and if the key matches the value of the *ll.head.data* field, then the address of the first node is replaced with the address of the next node, which becomes the main node (ll.head.next). Further in the loop, the node with the desired key is searched for and the nodes are shifted (Fig. 4.8).

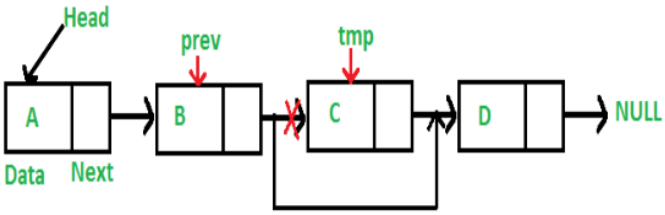


Figure 4.8. Illustration of removing a node from a list

A Drakon-diagram of the algorithm for removing a node by value is shown in Figure 4.9.

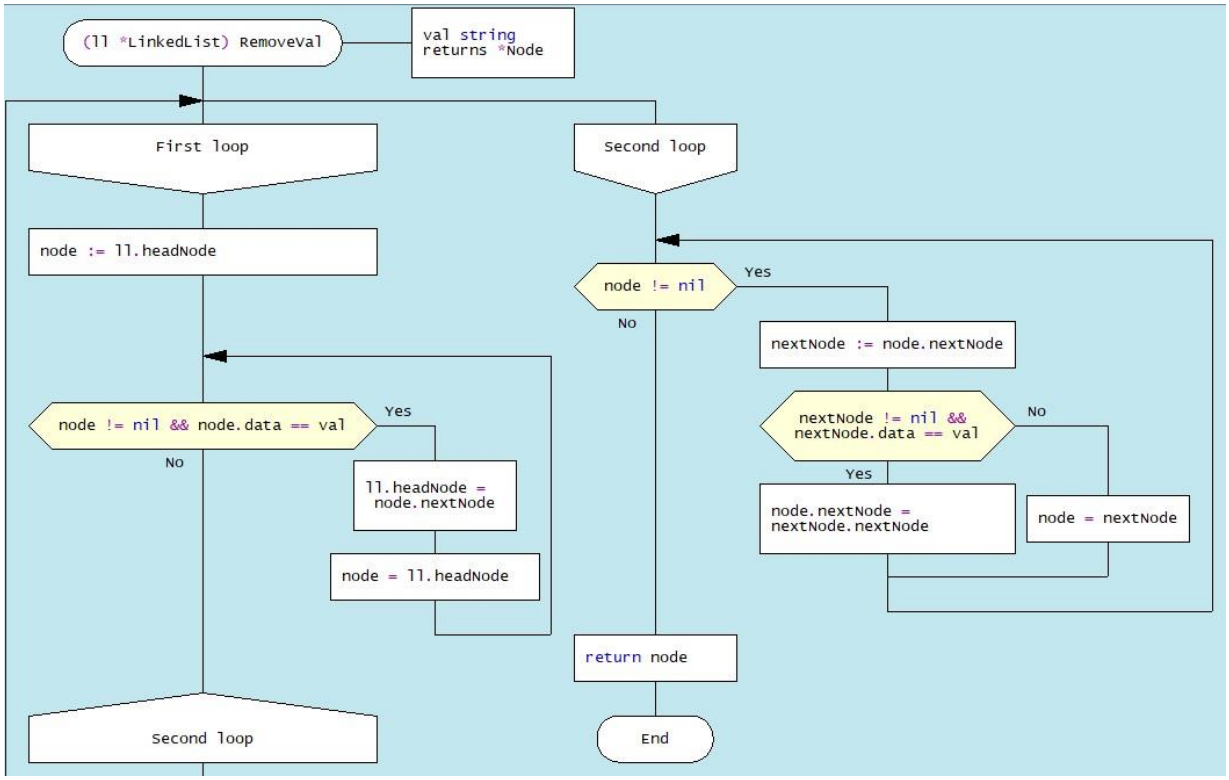


Figure 4.9. Drakon-diagram of removing a node from the removeVal list

As an example, let's create a list consisting of seven nodes containing seven values: *"Smith A.", "Shafler B.", "Shafler B.", "Wiley D.", "Brown G.", "Black H."*. To form a list, you need to create the *Node{data, nextNode)* type, where *data* is the value, and *nextNode* is the address of the next node and the *LinkedList(len, headNode)* type, where *len* is the length of the list, and *headNode* - list header with type **Node*.

Next, in the *main()* function, you need to initialize and create an instance of the *linked list* and insert the first node (header) by calling the *pushFront("Smith A.")* method. Next, new nodes are inserted at the end of the list using the *pushBack()* method:

```
func main() {
   var ll LinkedList = LinkedList{}
   ll.n = 0
   ll.pushFront("Smith A.")
   ll.pushBack("Shafler B.")
   ll.pushBack("Shafler B.")
   ll.pushBack("Wiley D.")
   ll.pushBack("Brown G.")
   ll.pushBack("Black H.")
```

Next, after the node with the value of *Brown G.* insert a node with the value *Singer L.* and delete the node with the *Black H.* value .

```
   ll.pushVal("Brown G.", "Singer L.")
   ll.iterateList()
   ll.removeVal("Black H.")
   ll.iterateList()
}
```

Further, in this list, the data about the person Shafler B. is repeated, as a result of which one record must be deleted. To do this, you need to find this record and delete it, the drakon diagrams of the record search algorithm is shown in Fig. 4.10, .

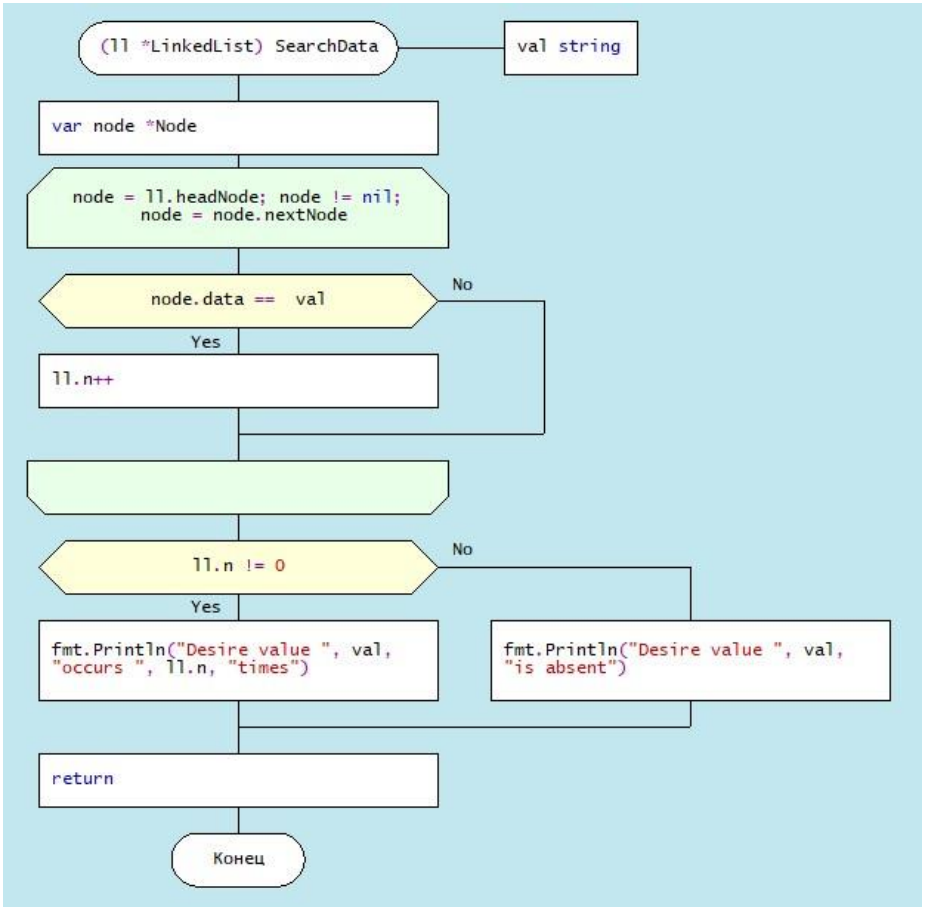


Figure 4.10. Drakon-diagram of the *SearchVal() method*

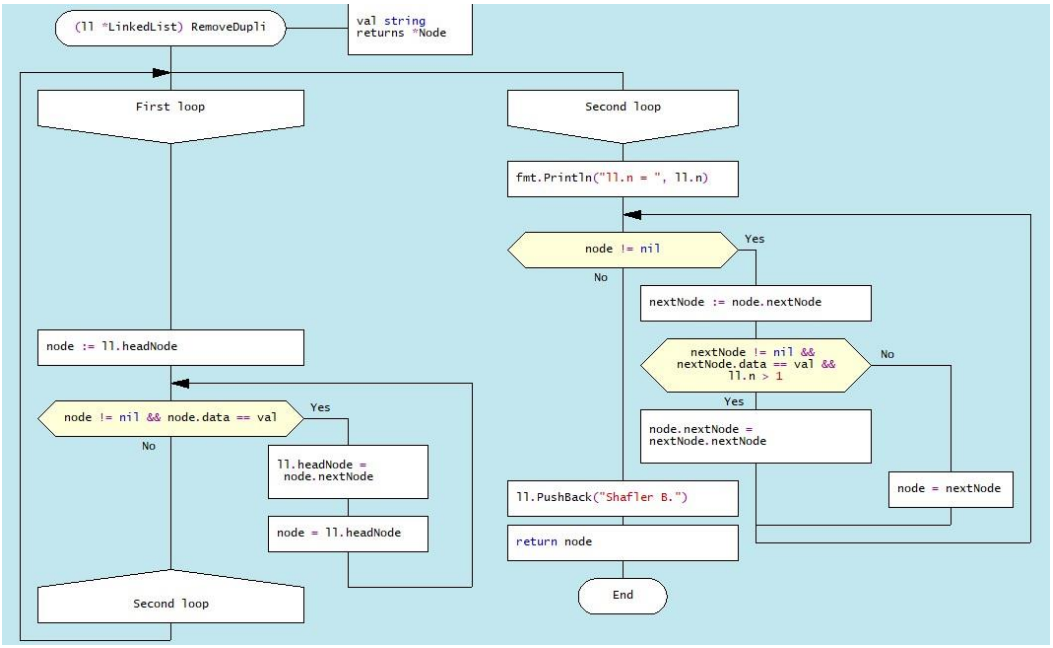


Figure 4.11. Drakon-diagram of the *removeDupli() method*

The results of processing list nodes are displayed on the monitor:

```
Smith A.
Shafler B.
Shafler B.
Wiley D.
Brown G.
Black H.
-------------------
Record Brown G is deleted
Smith A.
Shafler B.
Shafler B.
Wiley D.
Black H.
-------------------
NodeWith >  Wiley D.
Smith A.
Shafler B.
Shafler B.
Wiley D.
Singer L.
Black H.
-------------------
Desire value  Shafler B. occurs  2 times
-------------------
ll.n =  2
-------------------
Smith A.
Wiley D.
Singer L.
Black H.
Shafler B.
```

4.3.2. Stack

A stack is an abstract data type that contains elements with two basic operations: *push*, which adds an item to the collection, and *pop*, which deletes the last item added. A media set of this type includes a set of all stacks that contain elements of type T, including an empty stack, a stack with one element of type T, a stack with two elements of type T, and so on. From a technological point of view, a stack is a

memory, in which the values of the data are loaded and retrieved according to the "last in - first out" (LIFO - Last-In-First-Out) strategy. Data enters the stack from only one side, called the top of the stack (Figure 4.11.):

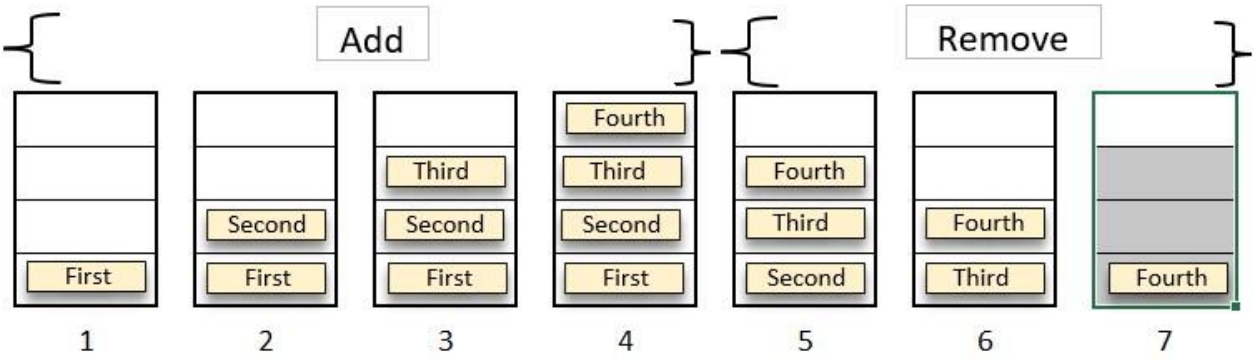


Figure.4.12. Last-in, first-out stack work

A common illustration of a stack is a stack of plates: only one plate is available at any given time - the one that is placed on the pile last, i.e., the top. In order to access an arbitrary plate inside the stack, you must first sequentially remove all the plates above it. The most common use of this structure is the implementation of the "Undo Last Action" operation in various editors (Ctrl+Z). There are other examples of data organization in a stack, such as using recursive function calls. Each function call address is stored on the stack and retrieved in reverse order.

Before performing any operations on the stack, let's look at how to programmatically create stacks in the Golang programming language. To create a stack, you can use a new *type of Stack* in the form of a slice:

```
type Stack struct {
    data []interface{}
}
```

In this description `interface{}` it's a special type in Golang that can represent values of any type. Thus, a slice `data` can contain elements of different types (for example, numbers or strings).

According to the purpose of the stack, the main operations with its elements are the function (method) of adding *push(item*) and the function (method) of removing *pop():*

```
func (s *Stack) push(item interface{}) {
s.data = append(s.data, item)
}

func  (s *Stack) pop()  {
    if len(s.data) == 0 {
        fmt.Println("Stack is empty")
    } else {
        s.data = s.data[:len(s.data)-1]
    }
}
```

Let's take a look at the *s.data = s.data[:len(s.data)-1]*. This is the key line that removes the element from the stack. It works by modifying *the s.data* slice to exclude the last element. In Golang, slicers are dynamic, and you can resize them using the *s.data[:n]* syntax, which returns the first n elements of the slice. Here, n is equal to *len(s.data)-1* , which means "all items except the last one". In this way, the last item is removed from the stack.

Suppose the members of a single stack instance are integer data: 10, 20, 30. 40, and the other instance string data "First", "Second", "Third", "Fourth". Then, in the *main()* function, there are two instances of the *stack of type* Stack *are created*:

```
stack := Stack{}
```

Three elements are added to the initially empty stack using the *push(item)* function:

```
 stack.push("First")
stack.push("Second)
sack.push("Third")
```

Then remove one item at the top of the stack:

stack.pop()

As a result of the specified operation, the output is displayed:

"Stack --> [First Second Third]

Stack after removing --> [First Second]

According to the second approach, the new type is declared as a linked list in the form of a structure with two elements: the address of the head node *(head)*:

```
type Stack struct {
    head *Node
}
```

In turn, the head struct member is of type **Node*,

```
type Node struct {
    data interface{}
    next *Node
}
```

This means that, according to the concept of the Golang programming language, the *Stack* structure inherits the properties of the *Node structure.* Then the *push(val) function* looks like this:

```
func (s *Stack) push(val interface{}) {
    newNode := &Node{data: val, next: s.head}
    s.head = newNode
},
```

and the function to remove an element from *the pop()* stack is:

```
func (s *Stack) pop() interface{} {
    if s.head == nil {
        return nil
    }
    data := s.head.data
    s.head = s.head.next
    return data

    }
```

A useful stack function is the peek(), returns the element from the head of the stack without removing it:

```
func (s *Stack) peek() interface{} {
    if s.head == nil {
        return nil
    }
    return s.head.data
}
```

In this case, the main() function looks like this:

```
func main() {
    stack := &Stack{}
    stack.push("First")
    stack.push("Second")
    stack.push("Third")

    fmt.Println(stack.peek()) // 3
    fmt.Println(stack.pop())  // 3
    fmt.Println(stack.pop())  // 2
    fmt.Println(stack.pop())  // 1
    fmt.Println(stack.pop())  // nil
}
```

For this option, the `main()` function looks like:

```
func main() {
    stack1 := &Stack{}
    stack2 := &Stack{}
    stack1.push(10)
    stack1.push(20)
    stack1.push(30)
    stack2.push("First")
    stack2.push("Second")
    stack2.push("Third")
    fmt.Println("The top element integer value of the stack",
stack1.peek())
    fmt.Println("Contents of an integer stack")
    fmt.Println(stack1.pop())
    fmt.Println(stack1.pop())
    fmt.Println(stack1.pop())
    fmt.Println(stack1.pop())
```

```
    fmt.Println("The top element string value of the stack",
stack2.peek())
    fmt.Println("Contents of an string stack")
    fmt.Println(stack2.pop())
    fmt.Println(stack2.pop())
    fmt.Println(stack2.pop())
    fmt.Println(stack2.pop())
```

4.3.3. Queue

Queue — it is a linear data structure that differs from the stack in the order in which the elements are deleted: the last added element is deleted in the stack; in the queue, on the contrary, the element added first is deleted (Figure 4.12).

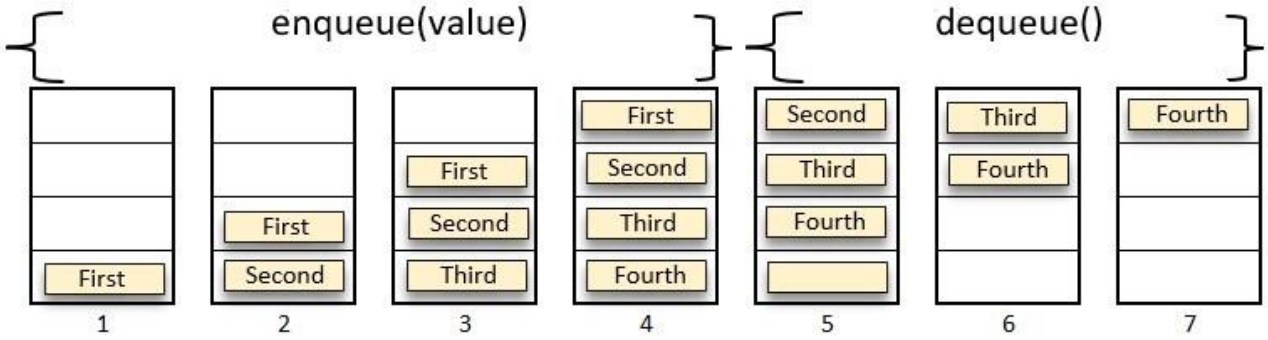


Figure 4.12. The work of the "queue" structure

The data structure in the form of a queue finds its application in multitasking systems, in communication systems (networks with intermediate storage), in queuing networks. Queues play an important role in computing when resources are provided on a first-come, first-served basis, such as jobs sent to a printer or processes waiting for a processor in the operating system.

The Queue abstract *data type* is defined as a class whose objects implement the FIFO or First-In-First-Out principle for the items that are added and removed. From the point of view of the concept of ATD, a queue is a container that contains values of a certain type. The media set of this data structure is the set of all queues that

contain items of type T, including an empty queue. In order for the algorithm that implements the basic functions of the queue to work the same for all data types, the Golang language uses an empty interface (interface{}) that can contain values of any type. In the case of a queue, the new type is declared:

```
type Queue interface {
    enqueue(item interface{})
    dequeue() interface{}
    isEmpty() bool
}
```

The set of basic operations supported by the queue includes:

enqueue (item) – add a single item to the end of the queue.

dequeue () - remove a single item from the top of the queue.

Queue creation is implemented in Golang either using a *slice* or a *linked list*. In the first case, to create a queue, enter the type in the form of a structure:

```
type SliceQueue struct {
   queue []interface{}
}
```

The following are the main methods related to the processing of queue elements - adding a new *enqueue (item)* element to the end of the queue and removing the *dequeue ()* element:

```
func (q *SliceQueue) enqueue(item interface{}) {
    q.queue = append(q.queue, item)
}
```

```
func (q *SliceQueue) dequeue () interface{} {
   temp := q.queue[0]
   q.queue = q.queue[1:]
   return temp
}
```

Additional method in relation to the *Queue structure* is the *isEmpty()* method, which returns true in the case of an empty queue, Size(), which returns the length of the queue, and some others:

```
func (q *SliceQueue) isEmpty() bool {
    return len(q.queue) == 0
}
```

Working with the queue in the *main() function* begins with creating two instances of a queue of *the SliceQueue* type with zero length:

```
queue1 := &SliceQueue{ }
queue2 := &SliceQueue{ },
```

where *queue1* – queue instance with integer items;

queue2 – queue instance with string items.

Next, we implement an illustration of the operation of two queue instances, the first for numeric data, the second - for string data.

For integer data:

```
queue1.enqueue(10)
queue1.enqueue(20)
queue1.enqueue(30)
queue1.enqueue(40)
```

For string data:

```
queue2.enqueue("First")
queue2.enqueue("Second")
queue2.enqueue("Third")
queue2.enqueue("Fourth")
```

The output of the program code:

A set of integers has been formed in the queue: 10, 20, 30, 40

Result with queue1

Output 1 --> 10

Output 2 --> 20

Output 3 --> 30

Output 4 --> 40

Output 5 --> 50

The queue of numbers is empty

A set of integers has been formed in the queue: First, Second, Third, Fourth

Output 1 --> First

Output 2 --> Second

Output 3 --> Third

Output 4 --> Fourth

The queue of strings is empty

You can create a queue based on a l*inked list*. To do this, a *Node* structure is created that contains *a value* of any type (interface{}) and a pointer (*Node) to the next node of the same type:

```
type Node struct {
    value interface{}
    next  *Node
}
```

A *Queue* structure is also created that contains pointers to the *front* and *rear* nodes of the queue, as well as a *size* variable to control the size of the queue:

```
type Queue struct {
    front *Node
    rear  *Node
    size  int
}
```

The *enqueue (val)* method adds a new node with a value value to the end of the queue. If the queue is empty (*size == 0*), then the new node is both a front

node and a back node. Otherwise, a new *node is added to the back node, and the posterior node pointer is updated.*

```
func (q *Queue) enqueue(value interface{}) {
    newNode := &Node{value: value, next: nil}
    if q.size == 0 {
        q.front = newNode
        q.rear = newNode
    } else {
        q.rear.next = newNode
        q.rear = newNode
    }
    q.size++
}
```

The dequeue method deletes and returns the front queue node. If the queue is empty, nil is returned, if the queue is not empty, then the front pointer is updated to the next node and the queue size is reduced.

```
func (q *Queue) dequeue() interface{} {
    if q.size == 0 {
        return nil
    }
    value := q.front.value
    q.front = q.front.next
    q.size--
    return value
}
```

SECTION 5. COMPLEXITY AND STABILITY ANALYSIS OF ALGORITHM

5.1. Main characteristics of the algorithms

The previous sections focused on the description of the most common data structures, on the one hand as abstract entities, and on the other - their concrete implementation in the programming language Golang. This section will deal with the theoretical analysis of algorithms for processing data structures from the perspective of estimating the time spent on program code execution and computer memory usage.

In theory and in practice, data structure processing algorithms are defined as a set of sequentially performed, stable procedures that provide a final computational result based on a fixed and limited set of source data. Most data processing algorithms include basic data processing procedures such as searching, sorting, adding, updating, deleting, etc.

Foremost, it should be noted that a "good algorithm" must have two characteristics: a) correctness and b) finality. *Correctness* means that if an algorithm is designed to solve a particular problem, it must always produce the correct result for all the source data and no input data will get the wrong result. The *finiteness* of the algorithm means that a finite number of computational operations must be performed to obtain the result, i.e. the execution of the program code must stop at some point in time.

Once the algorithm is correct and finite, the most important characteristic of the algorithm is efficiency, which is achieved by meeting the following requirements:

1. The algorithm should effectively use the resources available to the system;
2. The computational time (time taken to generate an output data corresponding to a particular input ones) should be minimal;
3. The memory used by the algorithm should also be as small as possible.

In most computational tasks, it is necessary to strive for a compromise between computation time and occupied memory. In other words, when choosing a method for solving a computational problem, you need to decide on priorities: what is more important - the computation time or the amount of memory occupied. It may seem that this is not so important for modern computers. However, when solving complex problems associated with the processing of huge amounts of data, for example, for artificial intelligence problems or for bioinformatics problems (for example, genome decoding), the search for the optimal combination of computing time and computer memory becomes decisive. In this regard, we will define the term "algorithm complexity" as a measure of the amount of time and/or space required to solve a problem, depending on the size of the data.

Such analysis involves determining a function that links the length of the input data of the algorithm to the time it takes (its *time complexity*) or the number of storage locations it uses (*the complexity of the space*). The algorithm is considered effective when the values of this function are either negligible or slowly increasing relative to the size of the input data.

5.2. Theoretical analysis of algorithm complexity

Ultimately, the created algorithms must be implemented on some computing device using the generated program code. Therefore, one should distinguish between the complexity of the algorithm itself and the complexity of its computer implementation. The fastest algorithm implemented on a slow computing device may be less efficient than a less successful algorithm implemented on a computer with more processing power or a programming language capable of parallel computing.

Thus, when assessing the complexity of the algorithms, they are carried out on an abstract machine with random access to memory, which makes it possible not to take into account the low-level parameters of the computing device (processor memory size, multitasking, etc.). The model of such a machine consists of memory and a processor, which work as follows:

- memory consists of cells, each of which has an address and can store one data element;
- each memory call takes one unit of time, regardless of the number of the addressable cell.
- the amount of memory is sufficient to execute any algorithm;
- the processor performs any elementary operation in one time step;
- loops and functions are not considered elementary operations.

A Labor intensity of the *Tn* algorithm associated with estimates of its complexity is determined by counting the number of operations performed. For example, consider the algorithm for finding the maximum element of an array.

```
package main
import "fmt"

func main() {

 array:= []int{11, 9, 17, 45, 411}
 N := len(array)
 maxNumber := array[0]
 for item:= 1; item < N; item++ {
    if array[item] > maxNumber {
    maxNumber = array[item]

  }

 }
 fmt.Println("The smallest value of an array element = ",
maxNumber)
}
```

When implementing this algorithm, the following will be performed:

1. (N - 1) the operation of assigning a new value to the loop counter i;

2. (N - 1) the operation of comparing the counter with the value N;

3. (N - 1) the operation of comparing an array element with the value of maxNumber;

4. from 1 to N operations of assigning a value to the variable maxNumber.

It is apparent that such a calculation for determining the complexity of the algorithm and estimating its complexity is individual and depends on the size of the input data. Therefore, it was quite natural to propose a qualitative assessment of complexity: depending on the “quality” of the initial data - the best option, the worst and the average.

5.3. Complexity qualitative assessment

Asymptotic notations such as omega-notation, theta-notation, and Big-O notation that reflect the three time bounds of the algorithm (lower, middle, and upper) are used to quantify the complexity of algorithms. Omega notation is the lower bound of the algorithm’s running time, which characterizes the best algorithm complexity in terms of achieving efficiency. For any value of n of the source data, the minimum time required by the algorithm is set to Ω(f(n)).

The Big-O notation represents the upper bound on the execution time of the algorithm, which characterizes the worst complexity of the algorithm in terms of efficiency. In most practical cases, this notation is used to evaluate the effectiveness of algorithms, since it is the worst-case scenario of the computational process that the software developer must evaluate. Finally, theta notation (Θ-notation) is an upper and lower bound on the execution time of algorithms and is used to analyze their average complexity. Consider in more detail the Big O notation that determines the upper bound of any algorithm, that is, the algorithm cannot take longer than the upper bound. To begin with the simplest time-constant algorithm is an algorithm that requires the same amount of time, regardless of its input data, whose complexity is denoted as O(1). For example, given two numbers, you need to calculate the sum.

This is followed by the logarithmic time complexity O(log n). When the time taken by the algorithm is proportional to the logarithm of input size n, it is said to have logarithmic time complexity. An example is the simple calculation of the number of

print operations of the index cycle:

```
func main() {
    n := 16
    for i:= 1; i < n; i=i*2 {
        fmt.Println("i = ", i)
        }
    }
```

Recall that $log_2 16 = log_2 2^4 = 4\ log_2 2 = 4$ since $log_2 2 = 1$. Algorithms with logarithmic complexity are considered highly efficient because the ratio of execution time of one operation to the size of the data array decreases with its increasing size. Algorithms that work in logarithmic time are usually found in binary tree operations or binary search operations. Such algorithms will be discussed in the next section.

Linear time complexity (O) characterizes algorithms whose execution time linearly depends on the size of the input data (n). The simplest example is the problem of computing the sum of n numbers:

```
func main() {
    n := 16
    sum := 0
    for i:= 0; i < n; i++ {
        sum = sum + i
    }
    fmt.Println("sum =", sum)
}
```

Typical examples of linear time complexity are algorithms associated with arrays sorting, which will be shown in the next section. It is important to note that linear complexity algorithms do not require additional memory.

This is followed by the quadratic complexity of O(n2) algorithms, the number of operations in which is quadratic depending on the size of the input data. As an example we can give the algorithm of multiplication of two numbers in the interval *[1:n;1:m]:*

```
                func main() {
    n := 3
    m := 5
    pow := 1
    for i:= 1; i < n; i++ {
        for j:= 1; j < m; j++ {
            pow = pow*i*j
        }
    }
    fmt.Println("pow =", pow)
}
```

The comparison of the discussed copmlexities is presented in the form of a table. Table. 5.1. Comparison of general copmlexities

n	Constant O(1)	Logarithmic O(log n)	Linary O(n)	Linear logarithmic O(n log n)	Quadratic $O(n^2)$
1	1	1	1	1	1
2	1	1	2	2	4
4	1	2	4	8	16
8	1	3	8	24	64
16	1	4	16	64	256

Or in graphic form (Figure.5.1.).

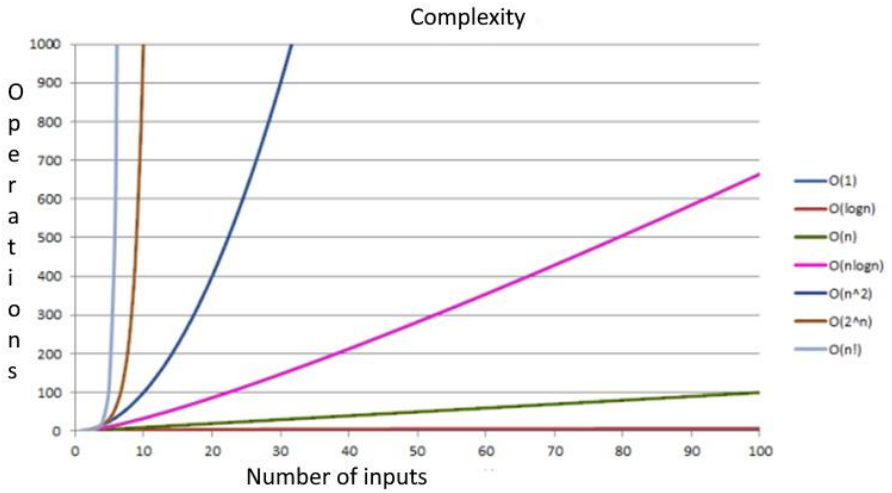


Figure 5.1. Comparison of general copmlexities

As shown in this table and graph, as the complexity of the function increases, the number of computations or the time required to perform the function can increase significantly. Next, the algorithms will be described with estimates of their complexity.

Another characteristic of the computational complexity of algorithms is *the spatial complexity*, estimated by the amount of memory required by the algorithm in its life cycle. This space consists of fixed and variable parts [Vasiliev]. A fixed part is the space required to store simple variables and constants, as well as the program size, which is independent of the complexity of the task. A part variable is the space required by variables, the size of which depends entirely on the size of the task. For example, recursion stack space, dynamic memory allocation, etc.

Spatial complexity is evaluated as O(1) if only the input data set is used. If the problem requires an auxiliary memory of the same size as the input memory (the memory to store the original array), then the spatial complexity is evaluated as O(n2).

It is especially important to know whether the variety used is stable. Especially in situations where your data already has some order that you would like to maintain when you sort it with a different sort key. For example, a spreadsheet has rows that contain student data, which is sorted by name by default. You'd also want to sort it by class, keeping the name order in order.

In other hand, sorting stability doesn't matter when the sort keys of the objects in the collection are the objects themselves – an array of integers or strings, for example, because we can't tell the difference between duplicate keys.

SECTION 6. BASIC SORTING ALGORITHMS

6.1. General characteristics of sorting algorithms

Sorting operations are used in almost all areas of human activity. Sorting algorithms are among the most common in IT data processing technologies. In general, the sorting task can be formulated as follows: there is a sequence of similar records, one of which is selected as a key (sorting key). You want to convert the original sequence to a sequence containing the same entries, but in the order in which the key values are increasing (or decreasing). The purpose of sorting is to facilitate a subsequent search of items in a sorted set.

At present, a sufficient number of sorting algorithms have been developed, with different possibilities of application depending on the number of items in the collection. Some algorithms are simple to implement and are suitable for small input datasets, but longer time is required for large datasets. Other algorithms are effective for sorting large datasets, but their use for small datasets is simply not effective. In addition, the algorithms differ in complexity of implementation. Understanding some of them requires visual accompaniment, in particular by using DRAKON- diagrams to represent logic of algorithm and explanatory illustrations. Data sets in the form of a slice almost cover arrays, so this section considers the slice sorting.

6.2. Bubble sorting

Sorting by "bubble" is the simplest algorithm, easy to implement for sets with few items. The algorithm got its name because the larger values gradually "*pop up*" at the end of the set. The DRAKON-diagram of the algorithm is represented in Figure 6.1. The implementation of the "*bubble*" sorting algorithm consists of two modules: `main()` and `bubblesort (ar [] int)`.

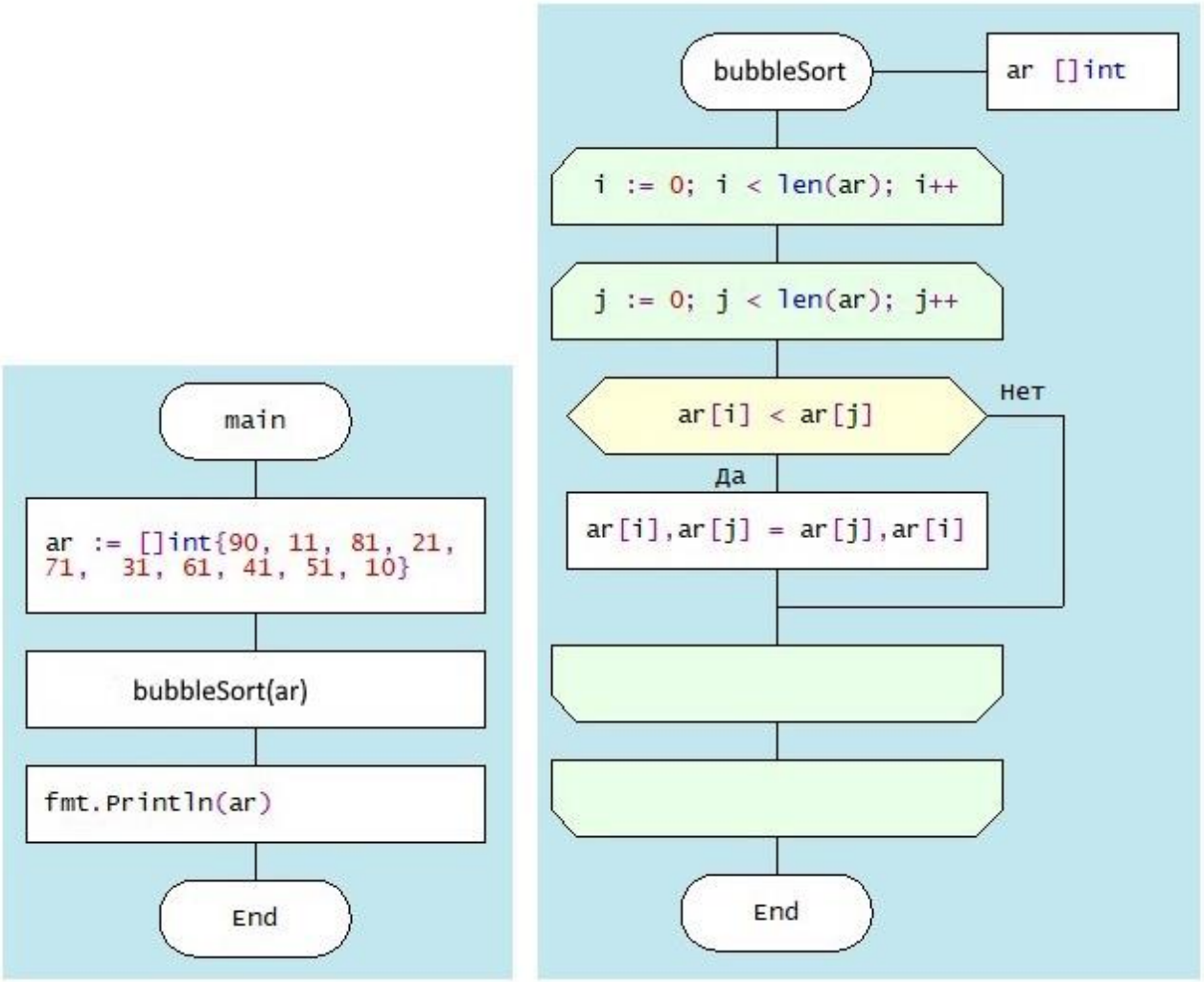


Figure. 6.1. Drakon-diagram of *bubbleSort* algorithm

In this algorithm, the set is traversed by index (j) for each index (i). In this case, each pair of values *ar[i]* and *ar[j]* are compared. If you want to sort values in ascending order, then the two items are swapped if the value of *ar[j]* is less than the value of *ar[i].* Otherwise, there is a transition to the next pass on the index (i). Thus, the largest values appear at the end of the set. Figure 6.2. presents a fragment of the algorithm at i = 8, and Figure 6.3. displays only strings in which the exchange of items took place.

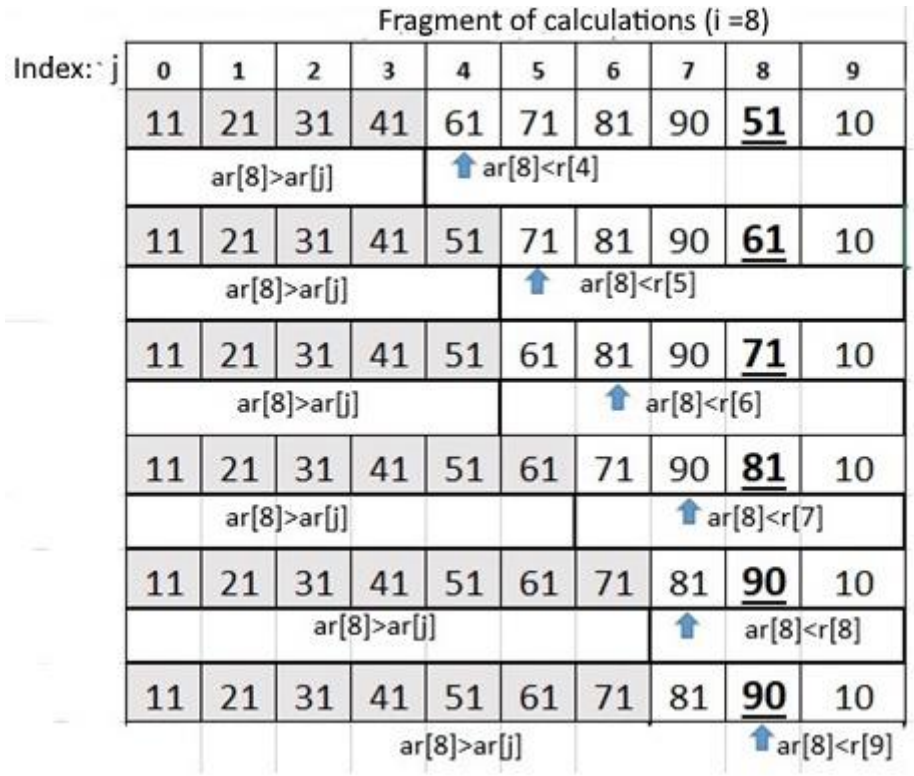


Figure 6.2. Fragment of the sorting process "Bubble”

```
[90 11 81 21 71 31 61 41 51 10]
[11 90 81 21 71 31 61 41 51 10]
[11 81 90 21 71 31 61 41 51 10]
[11 21 81 90 71 31 61 41 51 10]
[11 21 71 81 90 31 61 41 51 10]
[11 21 31 71 81 90 61 41 51 10]
[11 21 31 61 71 81 90 41 51 10]
[11 21 31 41 61 71 81 90 51 10]
[11 21 31 41 51 61 71 81 90 10]
[10 11 21 31 41 51 61 71 81 90]
[10 11 21 31 41 51 61 71 81 90]
```

Figure 6.3. The rows in which the items were exchanged

Obviously, the time complexity of this algorithm is quite high O(n2), since it is determined by the number of condition checks ar[i]<ar[j] and the number of exchanges ar[i] ar[j]. At the same time, the space complexity of the "bubble" algorithm is O(1), since

it does not require additional memory to organize the computational process. The algorithm has a high level of stability.

Because of its simplicity, Bubble sorting is often used, for example, in computer graphics, where it is popular for its ability to detect minor errors in almost sorted arrays and correct them with linear complexity (2n). However, the "bubble" algorithm is extremely inefficient for sorting large datasets.

6.3. Selection Sort

The choice sorting algorithm is based on the comparison operations, in which the data set is divided into two parts: the sorted left part and the unordered one in the right part. The selection DRAKON-diagram of the sorting algorithm is shown in Figure 6.4.

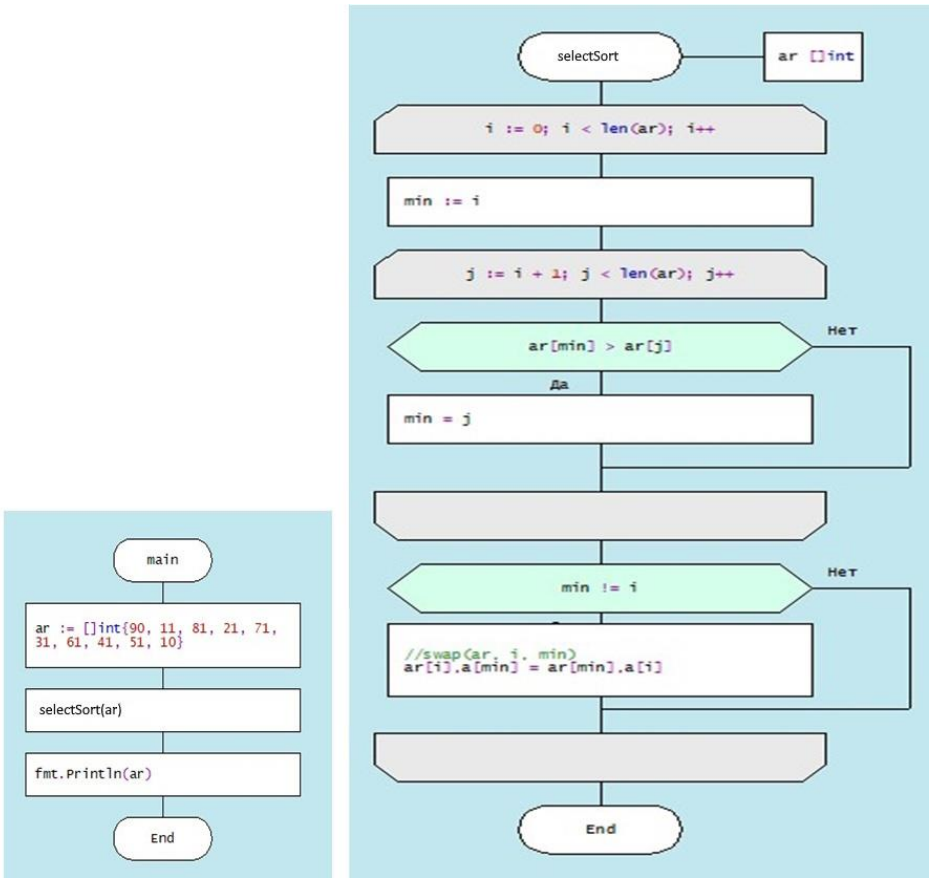


Figure 6.4. DRAKON-diagram of *selectionSort* algorithm

Function `selectionSort` is performed by obtaining the smallest value in each iteration and then replacing it with the current index. And the sorted part is empty, and the unsaved part is the whole set. The smallest item is selected from an unsorted array and replaced with the leftmost item, and this item becomes part of the sorted array. This process continues to move the undistributed edge of the array to one item to the right. For example, given a set of integers [90, 12, 83, 24, 75, 38, 62, 41, 59, 10]. In the first position, where 90 is currently stored, the algorithm passes the whole set and finds the smallest value - 10, after which the two values are reversed. This process shall then be applied to the remaining items in the slice (Figure 6.5.):

90	75	83	24	12	38	62	41	59	10
10	75	83	24	12	38	62	41	59	90
10	12	83	24	75	38	62	41	59	90
10	12	24	83	75	38	62	41	59	90
10	12	24	38	75	83	62	41	59	90
10	12	24	38	41	83	62	75	59	90
10	12	24	38	41	59	62	75	83	90
10	12	24	38	41	59	62	75	83	90
10	12	24	38	41	59	62	75	83	90
10	12	24	38	41	59	62	75	83	90

Figure 6.5. Selection and replacement of slice items

The evaluation of the complexity of the selection sort algorithm is presented in the table.

Time complexity:	
Worst case	$O(n^2)$
Average case	$O(n^2)$
Best case	$O(n^2)$
Space complexity:	$O(1)$
Selection sort algorithm is unstable	

The reason why time complexity is the same for all three cases is that the sorting algorithm by choice uses two nested cycles. The outer cycle is executed n times, where n is the number of elements in the array. In each iteration of the outer cycle, the inner cycle is executed (n-1) times. Thus, the total number of comparison and permutation operations is n*(n-1), giving the time complexity of O(n 2) for all three cases 14.

6.4. Insertion Sort

The insertion sort is performed by repeatedly extracting the item from the unordered part of the set and then inserting it into the sorted part of the set until all items are inserted. This algorithm is usually used by people when sorting stacks of papers. The DRAKON-diagram of the insertion sorting algorithm is presented in Figure 6.6.

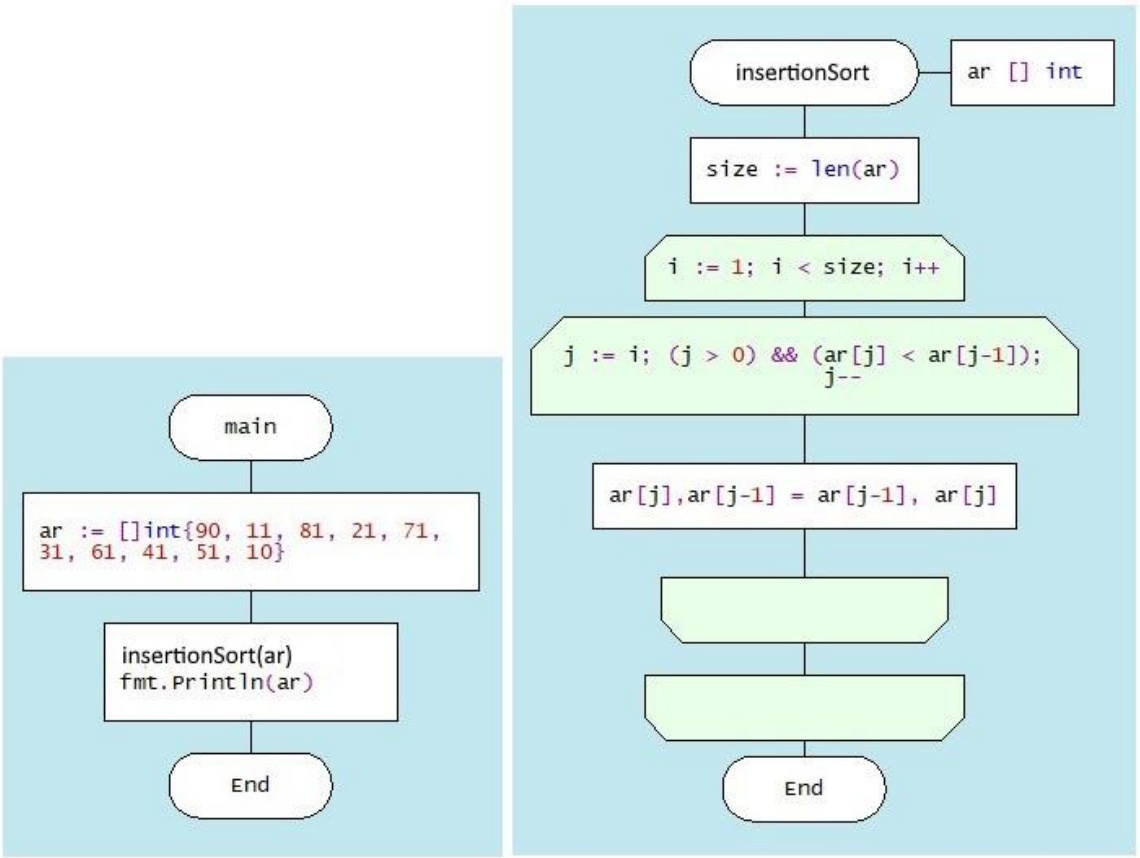


Figure 6.6. DRAKON-diagram of *insertionSort* algorithm

In this example, the items with the maximum value "advance" to the right, then a loop is executed, in which the neighboring items are compared and, if necessary, swap

places (marked with "_").

i = 1	i = 4	i = 8
[11 **90** 81 21 71 31 61 41 51 10]	[11 21 71 81 **90** 31 61 41 51 10]	[11 21 31 41 51 61 71 81 90 **10**]
[11 81 **90** 21 71 31 61 41 51 10]	[11 21 71 81 31 **90** 61 41 51 10]	[11 21 31 41 51 61 71 81 **10** 90]
i = 2	[11 21 *71* 31 81 **90** 61 41 51 10]	[11 21 31 41 51 61 71 **10** 81 90]
[11 81 **90** 21 71 31 61 41 51 10]	[11 21 31 *71* 81 **90** 61 41 51 10]	[11 21 31 41 51 61 **10** 71 81 90]
[11 81 21 **90** 71 31 61 41 51 10]	i = 5	[11 21 31 41 51 **10** 61 71 81 90]
[11 21 81 **90** 71 31 61 41 51 10]	[11 21 31 71 81 **90** 61 41 51 10]	[11 21 31 41 **10** 51 61 71 81 90]
i = 3	i = 7	[11 21 31 **10** 41 51 61 71 81 90]
[11 21 81 **90** 71 31 61 41 51 10]	[11 21 31 41 61 71 81 **90** 51 10]	[11 21 **10** 31 41 51 61 71 81 90]
[11 21 81 71 **90** 31 61 41 51 10]	[11 21 31 41 61 71 81 51 **90** 10]	[11 **10** 21 31 41 51 61 71 81 90]
[11 21 71 81 **90** 31 61 41 51 10]	[11 21 31 41 61 71 51 81 90 10]	[**10** 11 21 31 41 51 61 71 81 90]
	[11 21 31 41 61 51 71 81 90 10]	i = 9
	[11 21 31 41 51 61 71 81 90 10]	[10 11 21 31 41 51 61 71 81 90]

Figure 6.7. Insertion and replacement of slice items

The complexity of the insertion sort algorithm is determined by the number of comparisons and movements of elements that need to be performed to order the array. It depends on how the array is originally sorted.

The worst case scenario is when the array is sorted backwards. In this case, each element should be compared with all the previous elements and moved to the beginning of the array. The number of comparisons and moves is equal to

(n(n-1)/2,

it leads to O(n2)

Average case: when the array is partially sorted. In this case, each element should be compared on average with half of the previous elements and moved to the appropriate position. The number of comparisons and movements is equal to *n2/4*, it leads to O(n2).

The best case is when the array is already sorted. In this case, each element should be compared with only one previous element and left in its place. The number of comparisons and moves is equal to

$$(n - 1),$$

it leads to O(n).

The evaluation of the complexity of the insertion sort algorithm is presented in the table.

Time complexity:	
Worst case	$O(n^2)$
Average case	$O(n^2)$
Best case	O(n)
Space complexity:	O(1)
insertionSort algorithm is stable	

6.5. Quick Sorting

Quicksort is a highly efficient sorting algorithm whose general scheme consists of the following steps:

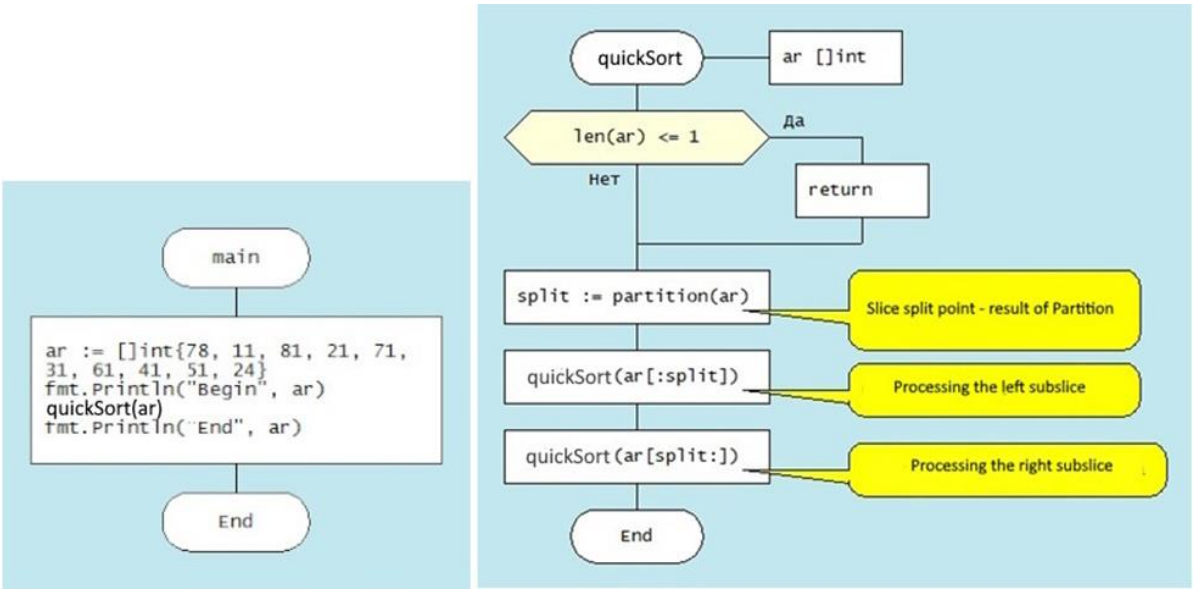
1. Selecting a reference item from a slice.

2. Redistributing items in a slice in such a way that items smaller than the reference one are placed in front of it, and those greater or equal - after it.

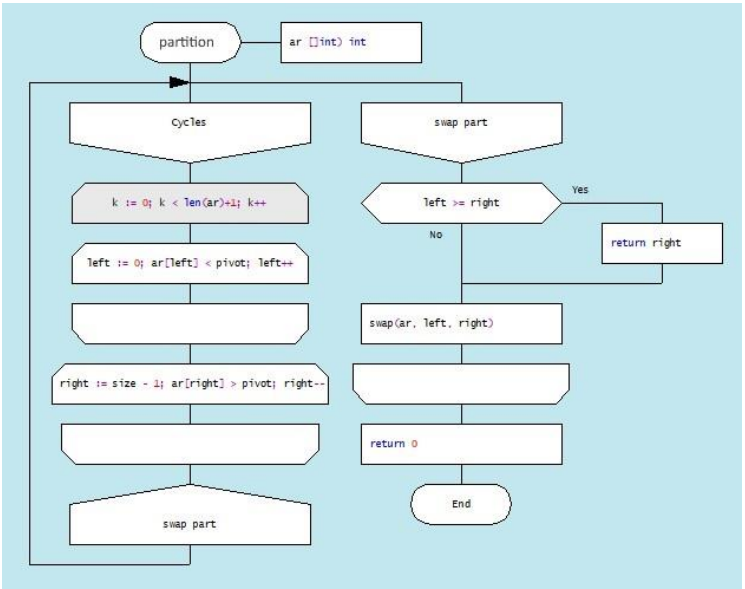
3. Recursively applying the first previous steps to slice fragments to the left and right of the reference item.

4. As a result, a fully sorted array is formed.

The DRAKON-diagram of the quicksort algorithm is shown in Figure 6.8.



a) b)



c)

Figure 6.8. DRAKON-diagram *quickSort* algorithm

a) function *main()* b) function *quickSort* c) function *partition*

Consider this algorithm in depth. The main function presents a collection of integer data. The pivot support item is selected with the slice item of 31.

78	11	81	21	71	31	61	41	51	28

Enter two pointers: left and right. At the beginning of the algorithm, they indicate the left and right end of the set respectively. In the left pointer algorithm, the left pointer is moved in 1 step towards the end of the slice until the current item value is less than the reference item value. The index of the first item, whose value is greater than *pivot* *(ar[left] > pivot)*, is fixed in the variable left. In the algorithm, the right pointer then moves from the end of the slice to the beginning until an item for which the condition *ar[right] <= pivot* is found. This fragment of the algorithm is executed until an item whose value exceeds that of the right item on the right is found on the left. In this case, these items are reversed. In this example, the first (left) item is larger than the supporting one (78 >31) and the last (right) item is smaller than the supporting one (24 < 31). In the second line of the table they switched. The process continues (Figure 6.9.):

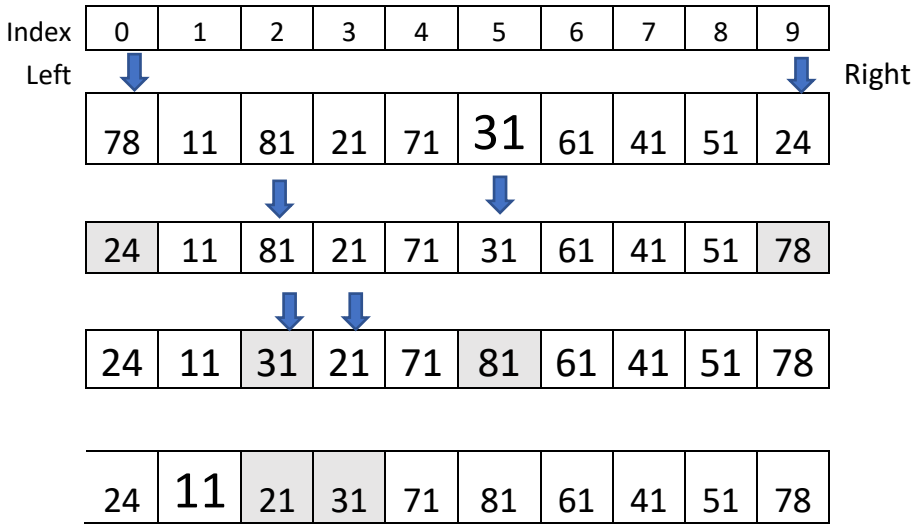
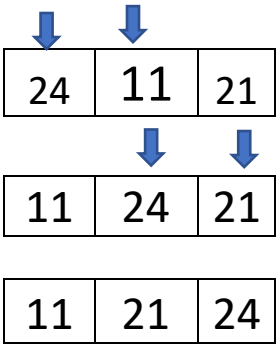


Figure 6.9. quickSort algorithm runtime

A new fragment of the source set is then formed in which the support item is overridden. In this fragment, pivot = 11.



After processing the left part of the set, the right part is processed in the same way. Eventually, the slice becomes sorted:

11	21	24	31	41	51	61	71	78	81

This algorithm is quite effective for large datasets as its average and worst complexity is $O(n^2)$, respectively.

The evaluation of the complexity of the Quick sort algorithm is presented in the table.

Time complexity:	
Worst case	$O(n^2)$
Average case	O(nlogn)
Best case	O(nlogn)
Space complexity:	O(nlogn)
Insertion sort algorithm is unstable	

6.6. Merge sorting

Merge sorting is performed by recursively splitting the collection into fragments until there is a fragment consisting of two items. The two items are easily compared and ordered according to the requirement: ascending or descending. The split is followed by an inverse merge, in which at one point (or in a loop) one item from each slice fragment is selected and compared. The smallest (or largest) item is stored in the result set, the remaining item remains valid for comparison with an item from another fragment in the following step (Figure 6.10.):

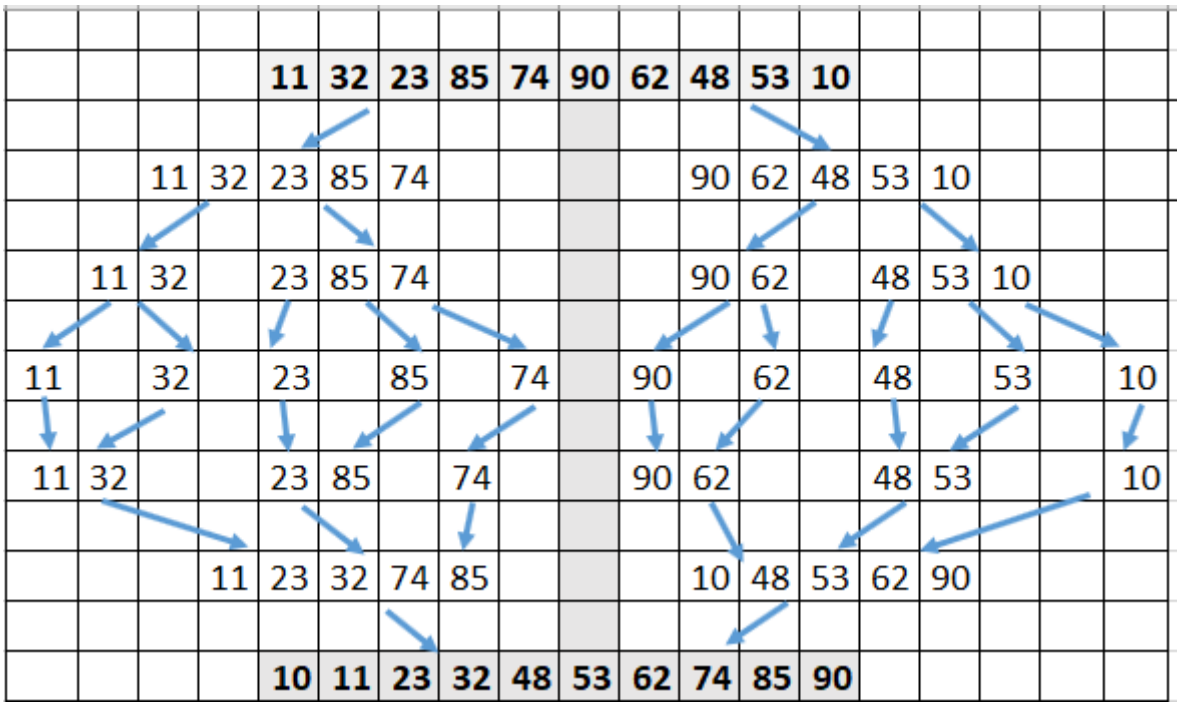
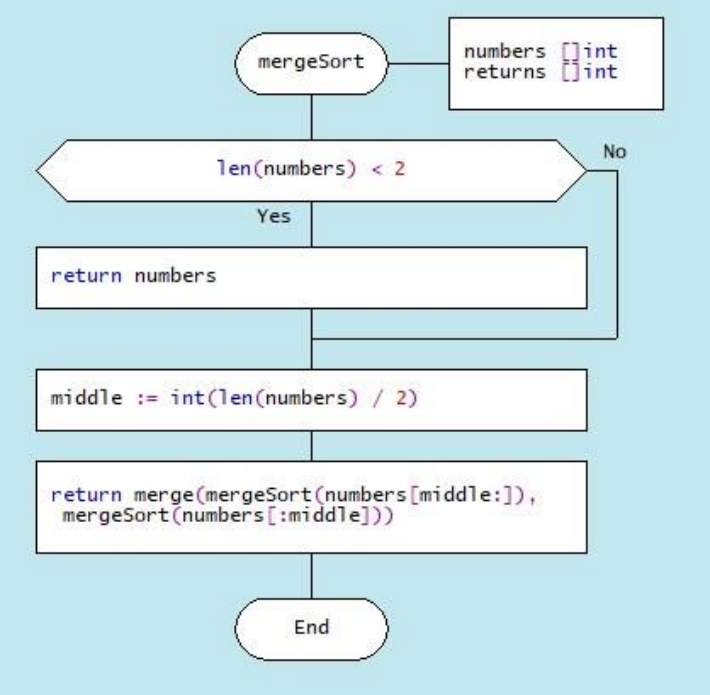
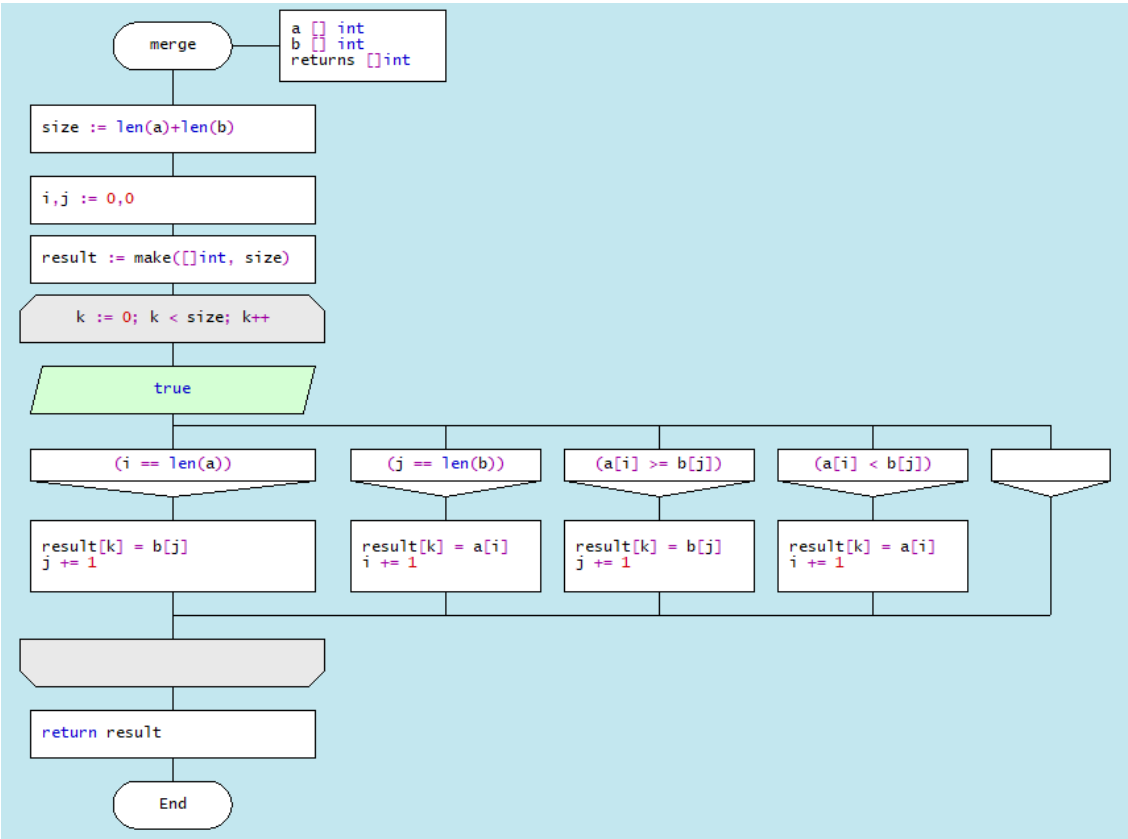


Figure 6.10. Visualizing the mergeSort algorithm

The DRAKON-diagram of the mergeSort algorithm is presented in Figure 6.11.



a)



b)

Figure 6.11. The DRAKON-diagram of the *mergeSort* algorithm:

a) function *mergeSort*; b) function *merge*

The evaluation of the complexity of the Merge sorting algorithm is presented in the table.

Time complexity:	
Worst case	O(nlogn)
Average case	O(nlogn)
Best case	O(nlogn)
Space complexity:	O(n)
Insertion sort algorithm is unstable	

6.7. Shell sorting

ShellSort is a sort of insertion sort. When sorting out, shellSort first compares and sorts between values that are separated from each other at some distance *d*. After that, the procedure is repeated for some smaller values *d* until the distance becomes *d=1* (that is, the usual sorting of inserts). The DRAKON-diagram of the algorithm is presented in Figure 6.12.

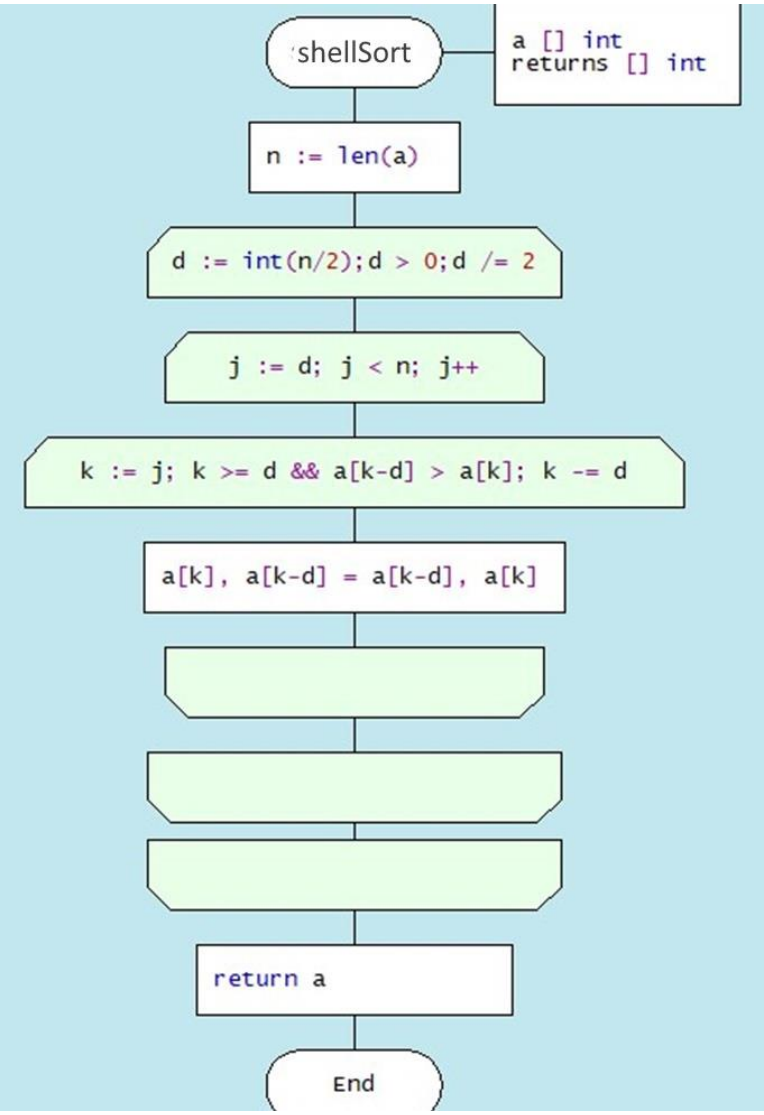


Figure 6.12. DRAKON-diagram of *shellSort* algorithm

Function shellSort uses a process that underlies many of the sorts presented: sequential segmentation, sorting of these segments, and finally grouping them into a sorted set. The process of partitioning occurs so that each item in the segment represents a fixed number of positions from each other. This creates uncertainty in the choice of this number of positions, in other words, the distance between the items in the segment (*d*). The simplest example is *d = n / 2, d2 = d/2 ... dn =1.*

Figure 6.13 shows the exchange of items under `a[k] < a[k-d]`, where d=5 at the first pass of the collection.

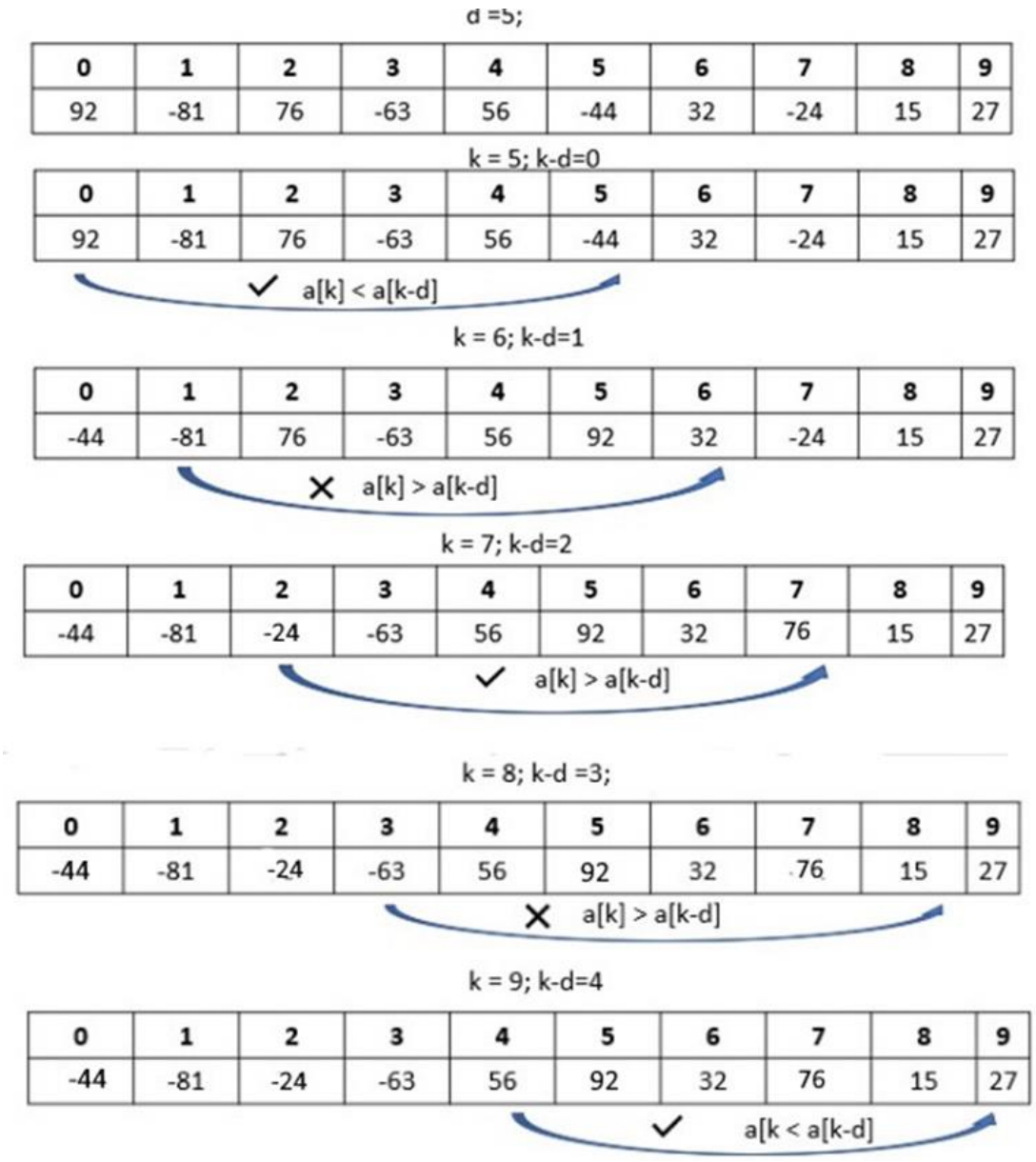


Figure 6.13. Fragment of algorithm of exchange of items of set

The evaluation of the complexity of the shell sorting algorithm is presented in the table.

Time complexity:	
Worst case	$O(n\log n)^2$
Average case	$O(n\log n^2)$

Best case	O(n)
Space complexity:	O(1)
Insertion sort algorithm is unstable	

6.8. Pyramid sorting (heapsort)

The pyramid sorting algorithm can be seen as an improved version of the choice sorting algorithm (`selectSort`): it divides the input data into sorted and unreported areas, and then successively reduces the unreported area, removing the largest item and moving it to the sorted area. An improvement is that the binary pile is used to find the highest value, not the linear search algorithm. This algorithm is executed using the notion of heap, which is a complete binary tree (see sub-section 1.3.). All nodes of a heap are either larger than its child items or smaller than its child items. A heap binary tree can be of two types: a minimum heap (MinHeap), in which the parent node is always smaller than the child nodes, and a maximum heap (MaxHeap), in which the parent node is always greater than or equal to the child nodes Figure 6.14).

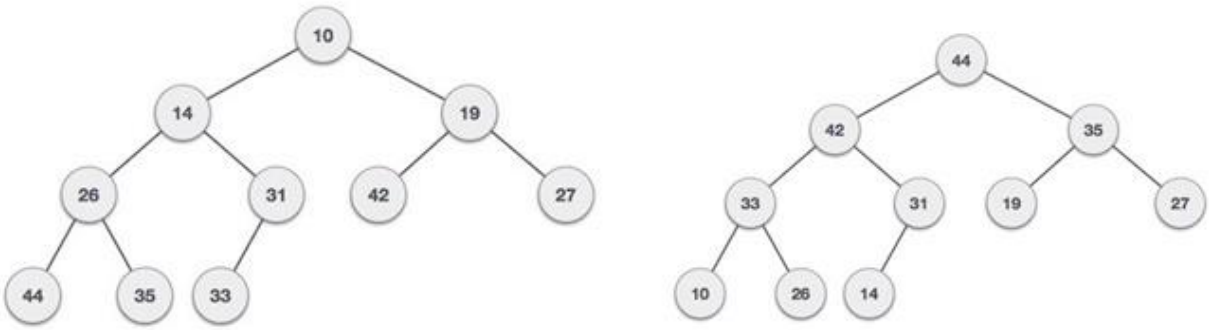


Figure 6.14. Binary tree examples

Construction of a binary tree from an array is simplified by the fact that the original array is actually an unordered heap (Figure 6.15.):

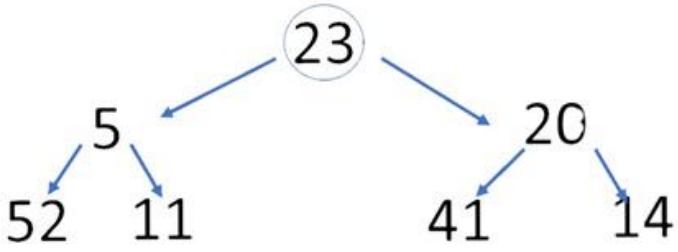


Figure 6.15. Binary tree example

The tree node sequence, starting with the root node, is performed by the formula:

$i_н$ = (array size / 2) – 1;

First, the algorithm swaps the nodes (20) and (41), then the nodes (5) and (52), then we present this process in the table:

23	5	20	52	11	41	14
23	5	41	52	11	20	14
23	52	41	5	11	20	14
52	23	41	5	11	20	14
14	23	41	5	11	20	52
41	23	14	5	11	20	52
20	23	14	5	11	41	52
23	20	14	5	11	41	52
11	20	14	5	23	41	52
20	11	14	5	23	41	52
5	11	14	20	23	41	52
11	5	14	20	23	41	52
5	11	14	20	23	41	52

The heap sorting algorithm uses three functions: *heap_Sort*, which performs node overwriting, *heapify*, which compares adjacent nodes, and swap, which swap two nodes. The sequence of the nodes in the heap is shown in Figure 6.16:

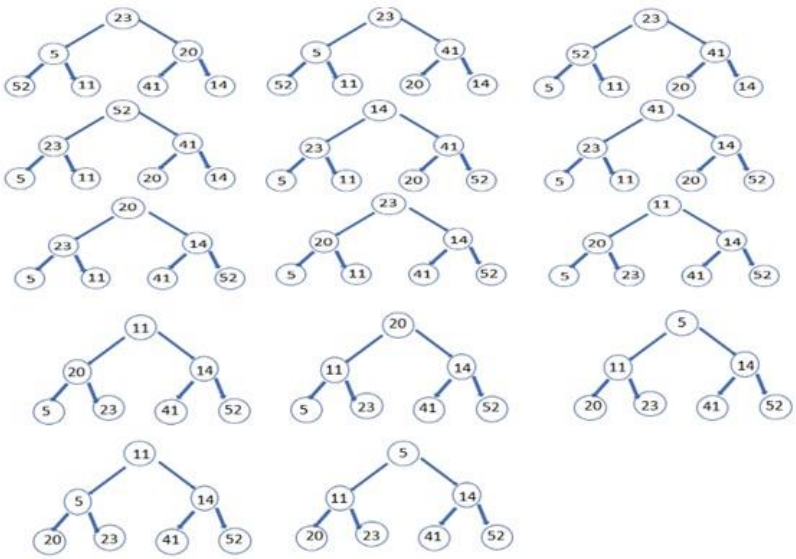
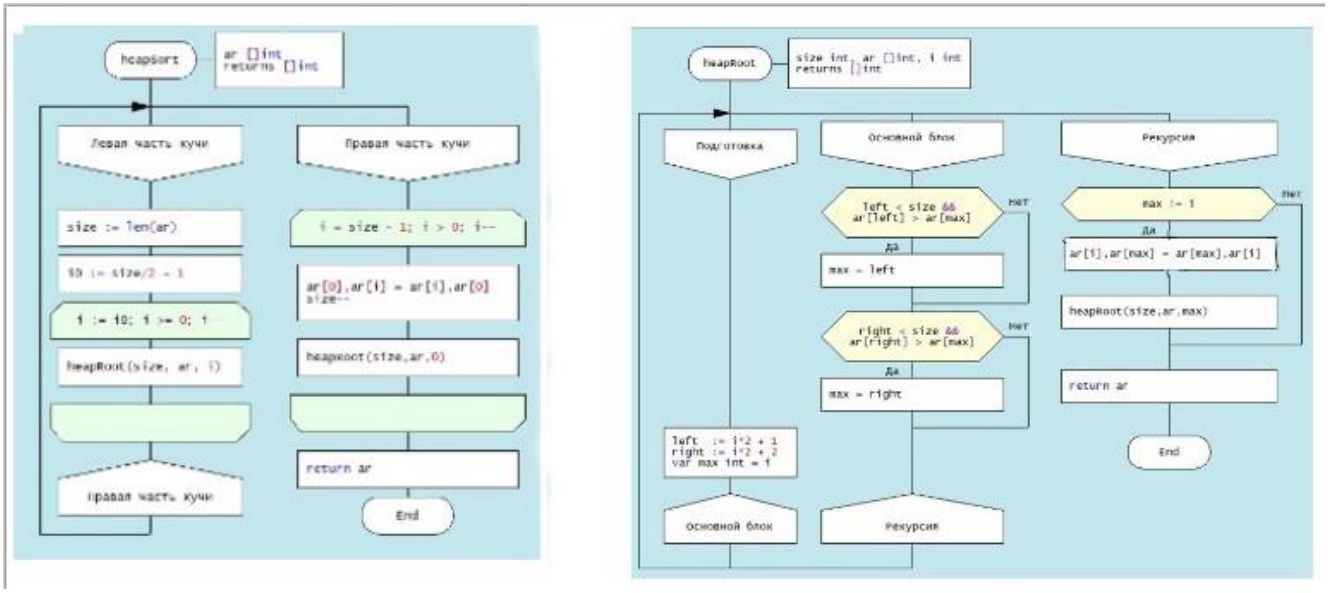


Figure 6.16. Sequence of node movement in heap

DRAKON-diagram of *heapSort* algorithm is presented in Figure 6.17:

.



a) function *heapSort* b) function *heapRoot*

Figure 6.17. DRAKON-diagram *heapSort* algorithm

The evaluation of the complexity of the heap sorting algorithm presented in the table

Time complexity:	
Worst case	O(nlogn).
Average case	O(nlogn).
Best case	O(nlogn).
Space complexity:	O(1)
Insertion sort algorithm is unstable	

6.9. Sorting comparison

The selection of a sorting algorithm is determined by the following factors:

- Time complexity;
- Spatial complexity;
- Stability/instability.

Knowing the strengths and weaknesses of each of the algorithms considered allows you to make a choice in favor of a particular sort. Each algorithm is unique and works best under certain conditions.

Sorting algorithm	Average	Best	Worst
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$
Quick Sort	O(n.log(n))	O(n.log(n))	$O(n^2)$
Merge Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))
Shell Sort	$n(\log(n)^2$	O(n)	$n (\log n)^2$
Heap Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))

Some common sorting algorithms are inherently stable, such as *`mergeSort`*, *`insertionSort`*, and *`bubbleSort`*. Others, such as *`quickSort`*, *`heapSort`* and *`selectionSort`*, are unstable. For example, we can use the extra space to maintain stability in *`quickSort`*.

SECTION 7. BASIC SEARCH ALGORITHMS

7.1. Main characteristics of 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.

7.2. 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).

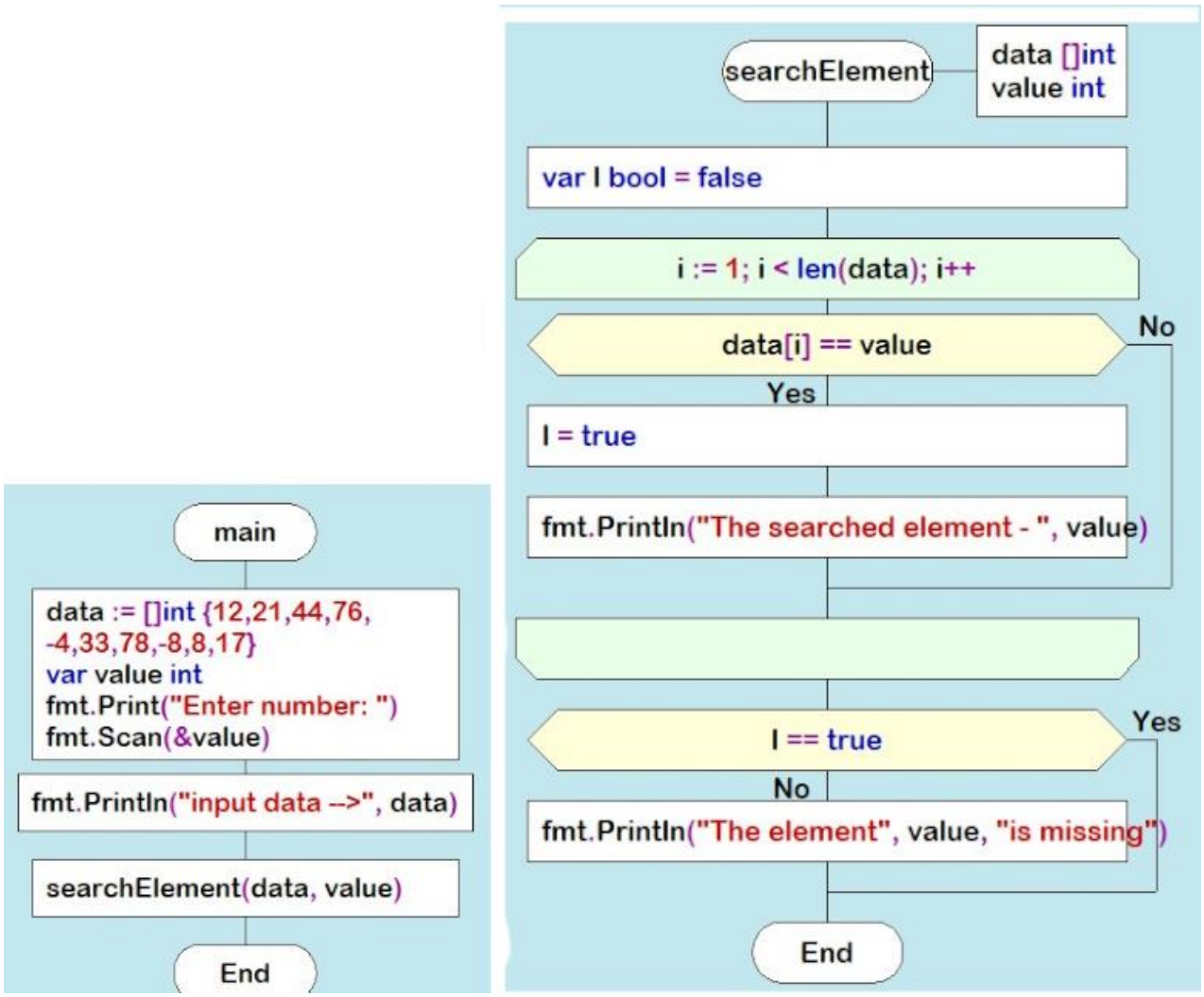


Figure 7.1. DRAKON-diagrams of linear search algorithm

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.

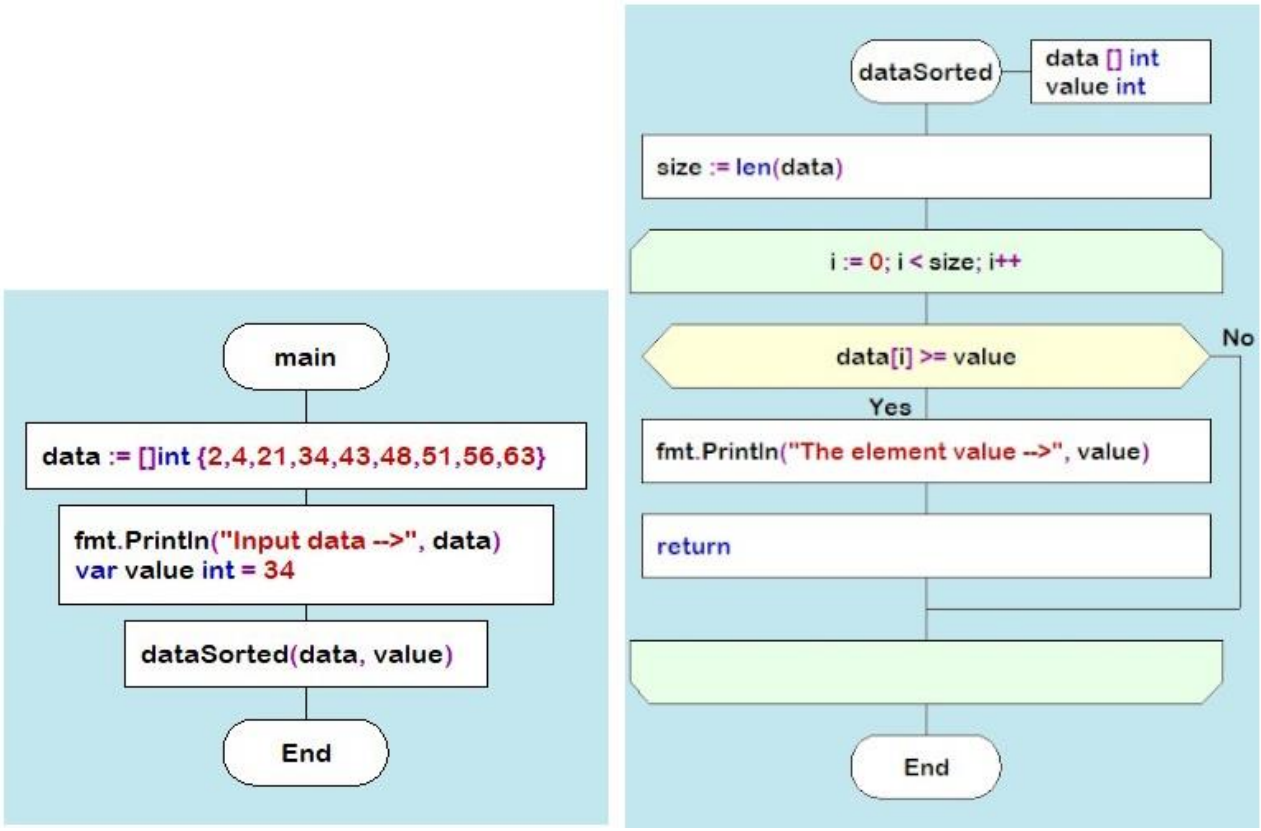


Figure 7.2. DRAKON-diagram of searching sorted slice algorithm

7.3. 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.

2. 3. The search is simply that the value of the middle element in the selected half is again determined and compared to the key.

3. 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):

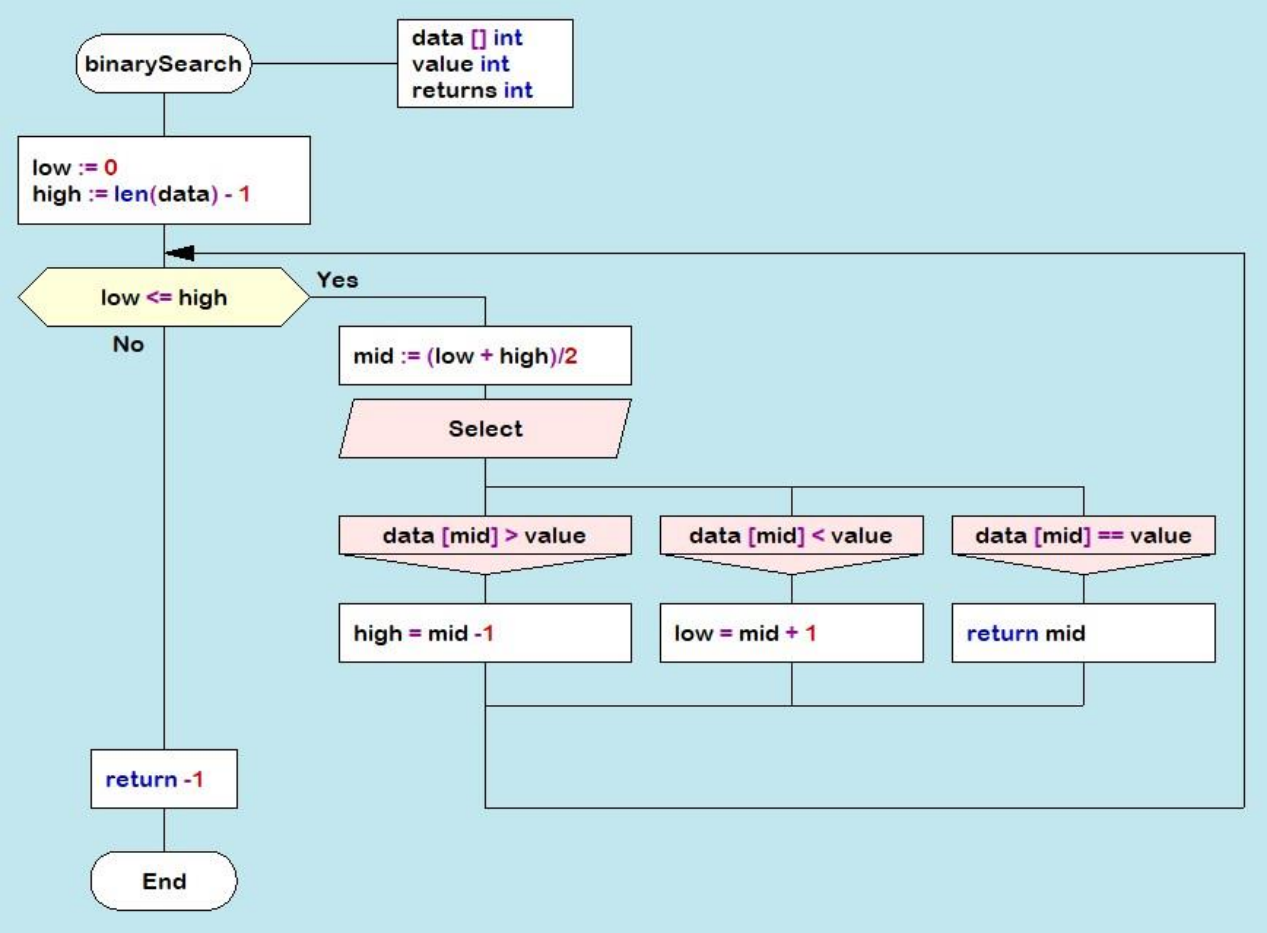


Figure 7.3. DRAKON-diagram of *binarySearch* algorithm

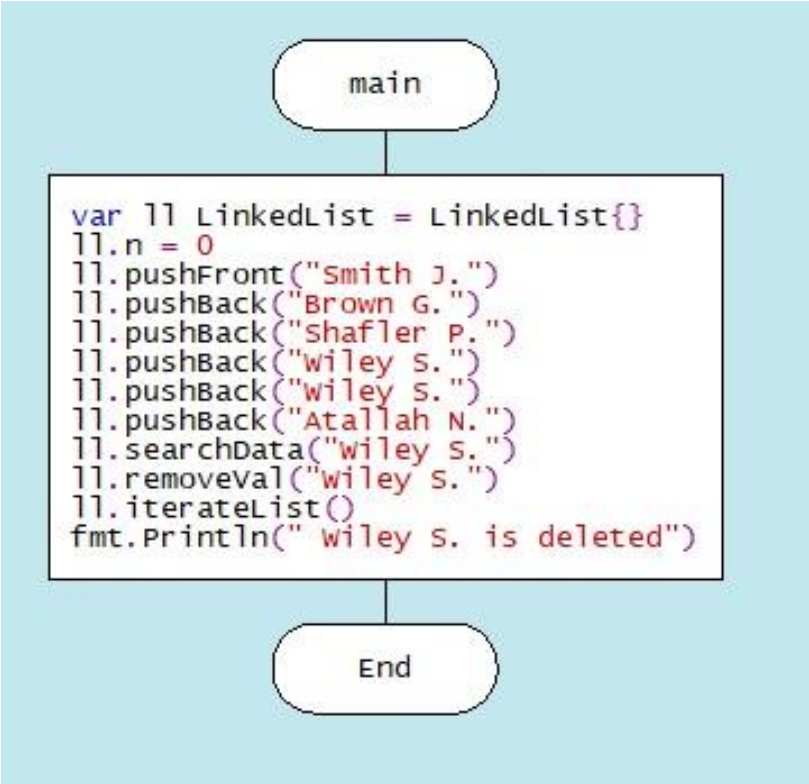
Time complexity of binary search algorithm belongs to class O(log n). The way to interpret this is that the asymptotic increase in the time taken by a function to perform a given input set of size n will not exceed log n.

Space complexity: O(1). That is, no extra space required.

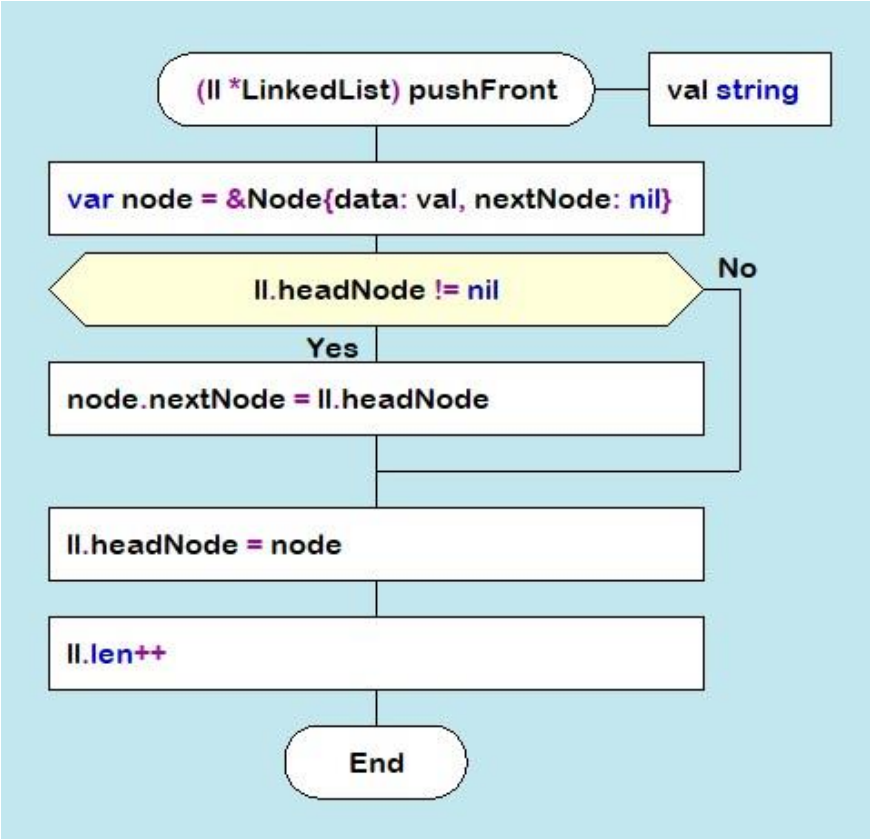
7.4. Searching in Single-Linked List

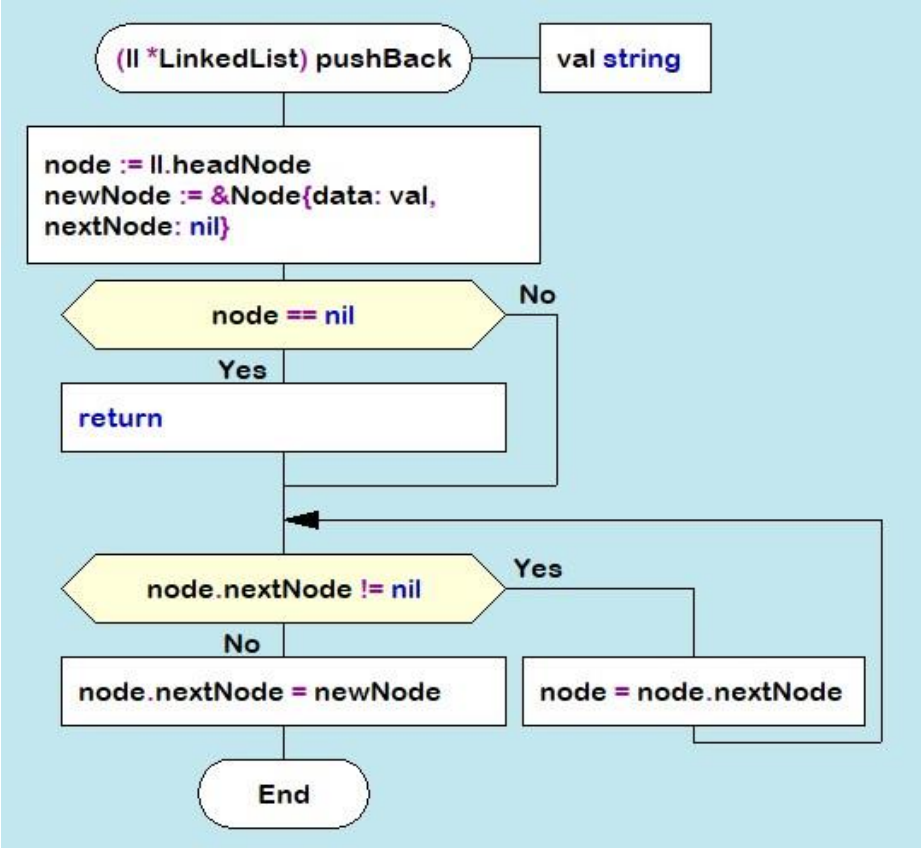
There are three possibilities for a Single-Linked List. First, the desired value is missing from the list, second, the desired value is encountered once and, third, the desired value is encountered repeatedly. You can also set the task of removing duplicates, i.e., nodes that are redundant.

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 Drakon-diagrams are presented in Figure 7.4 a,b,c,d:

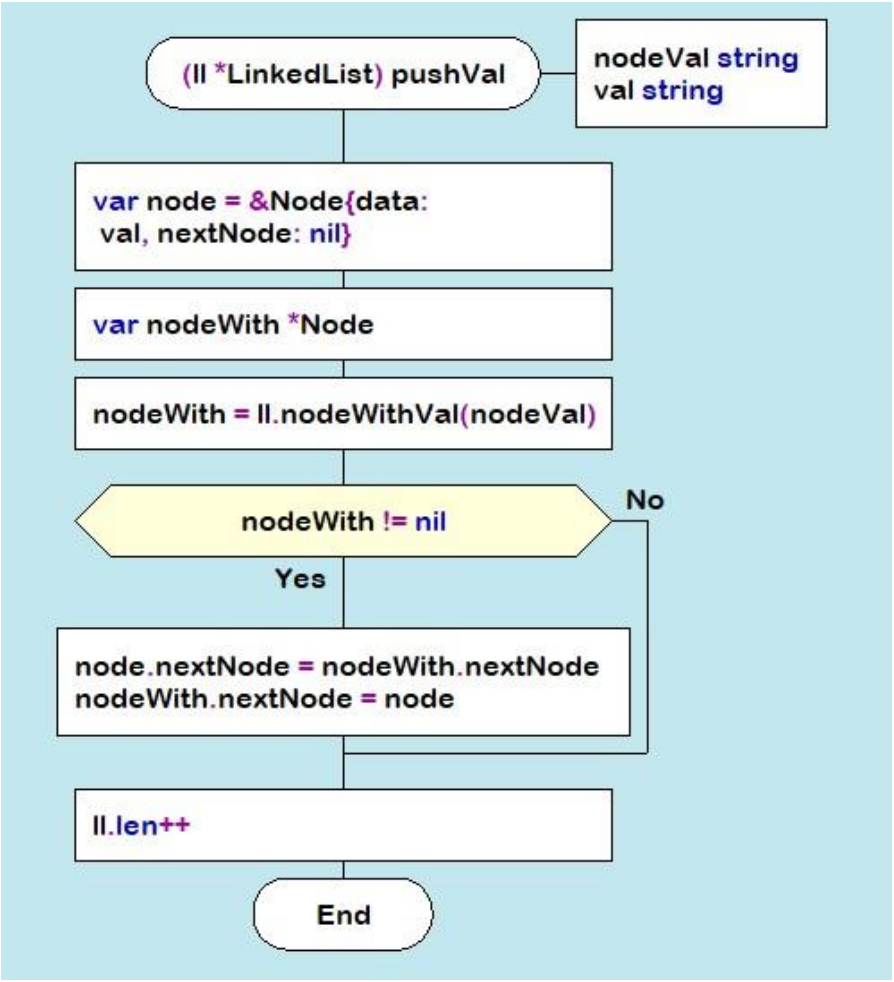


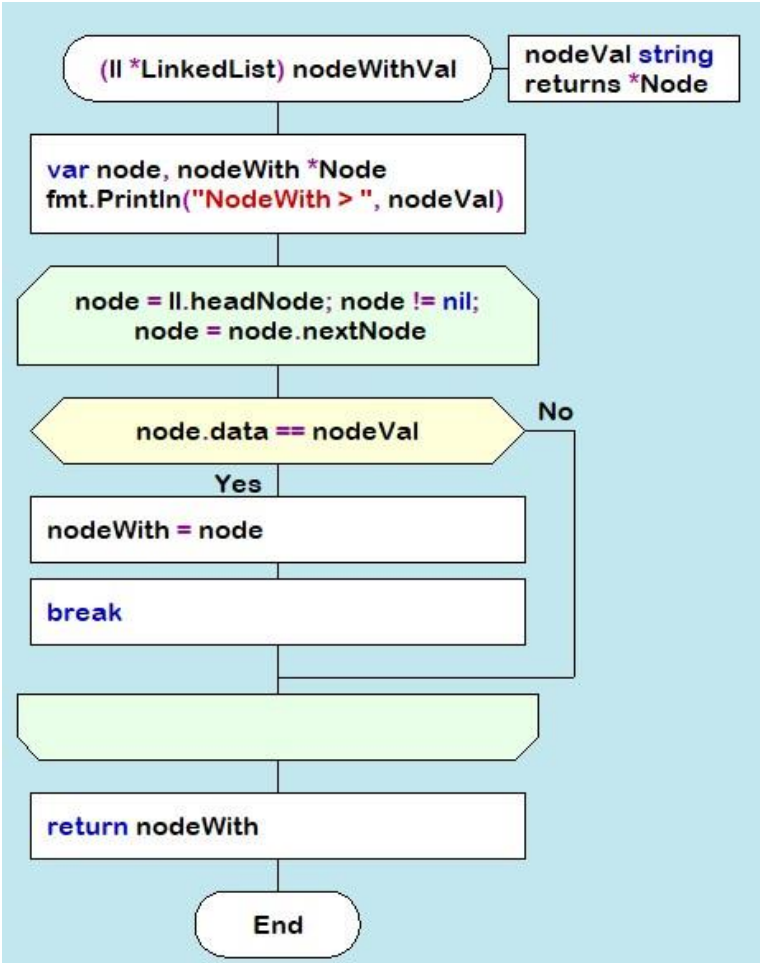
a) Function *main()*



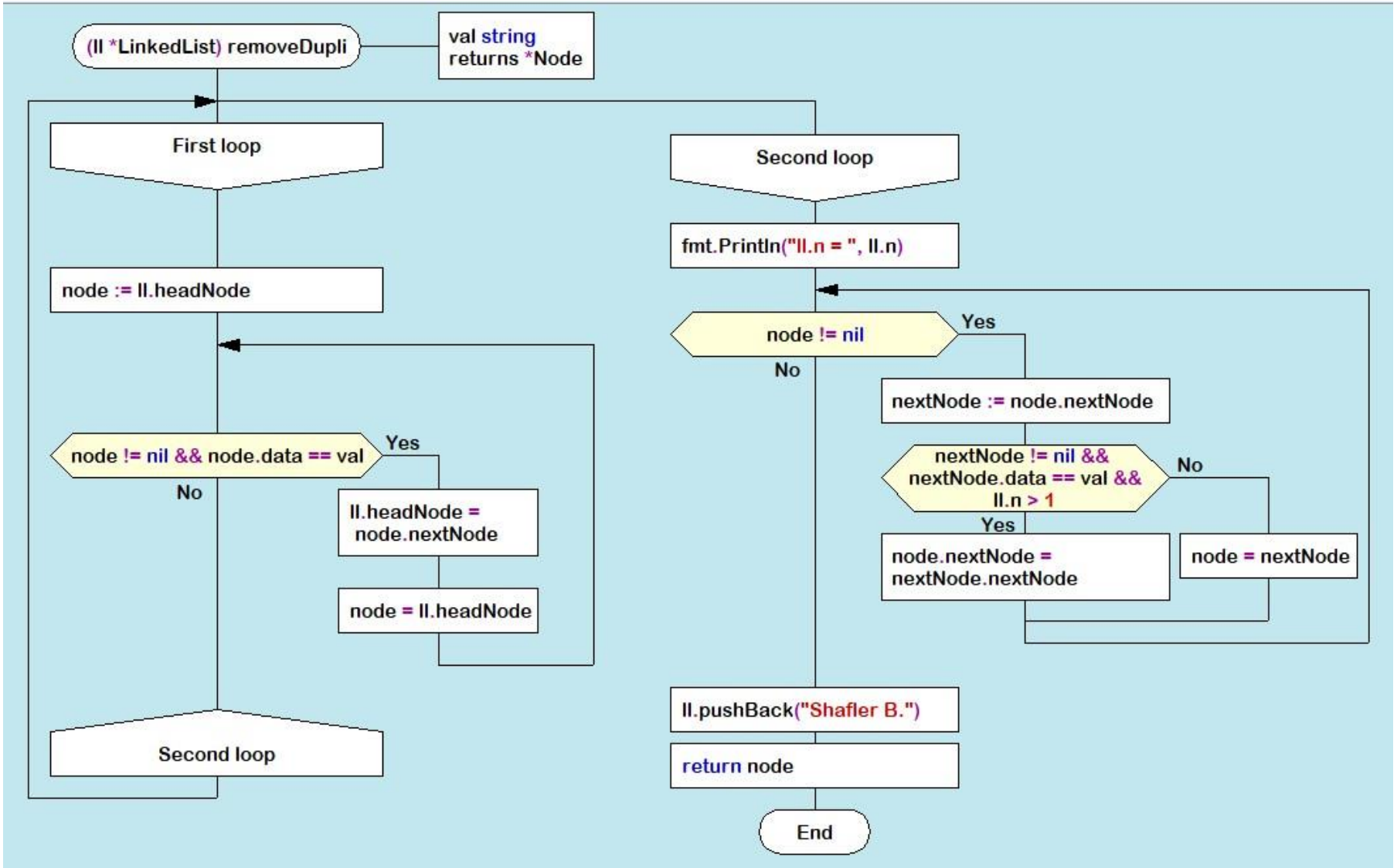


b) Functions inserting a node at the beginning and end of the list *pushFront* and *pushBack*

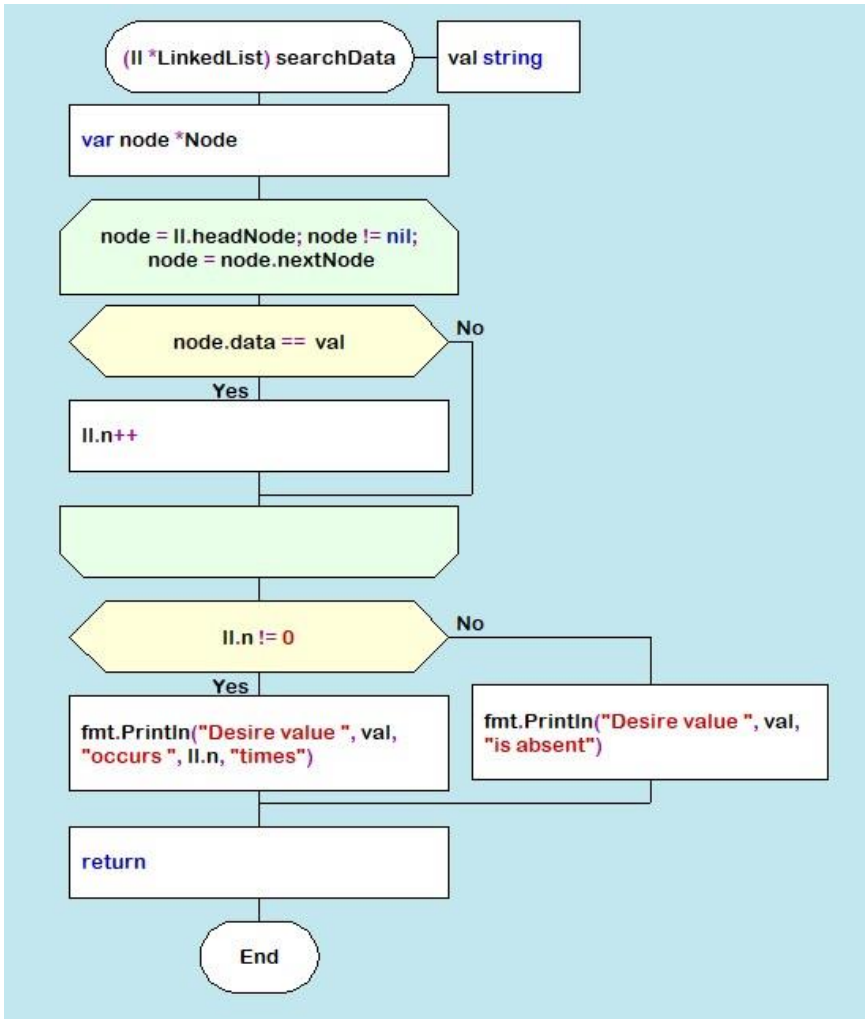


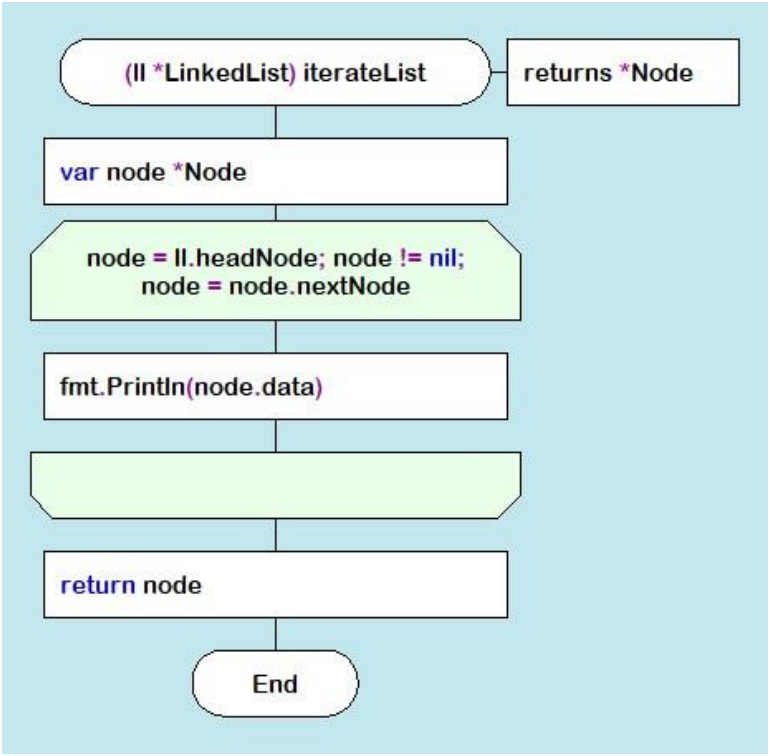


c) Funcrions inserting a node after a specified node *pushVal(val)* and *nodeWithVal(val)*



d) Function c) Function for deleting a specified node *removeDupli(val)*





e). Function of searching *searchData(val)* and list iteration *iterateList*

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

7.5. Hashing

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.

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.).

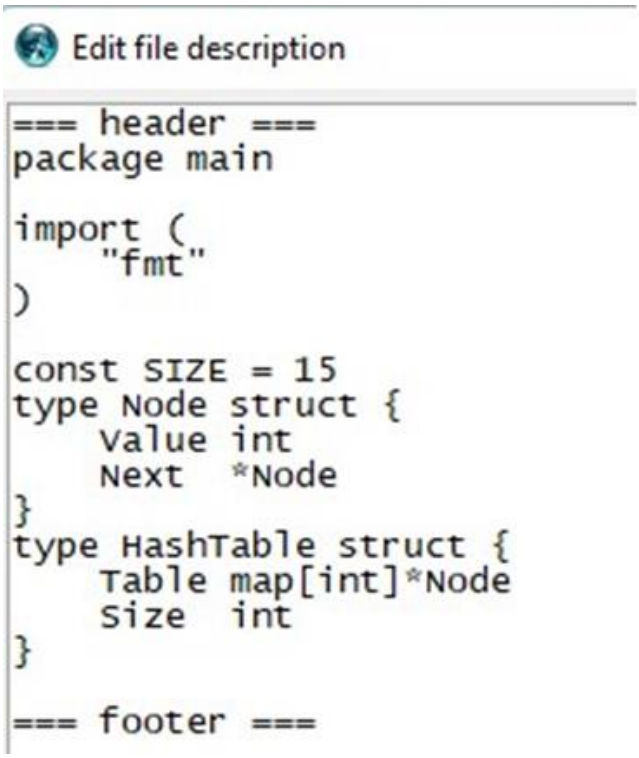


Figure 7.5. Type declaration *Node*, *HashTable* and constant *Size*

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

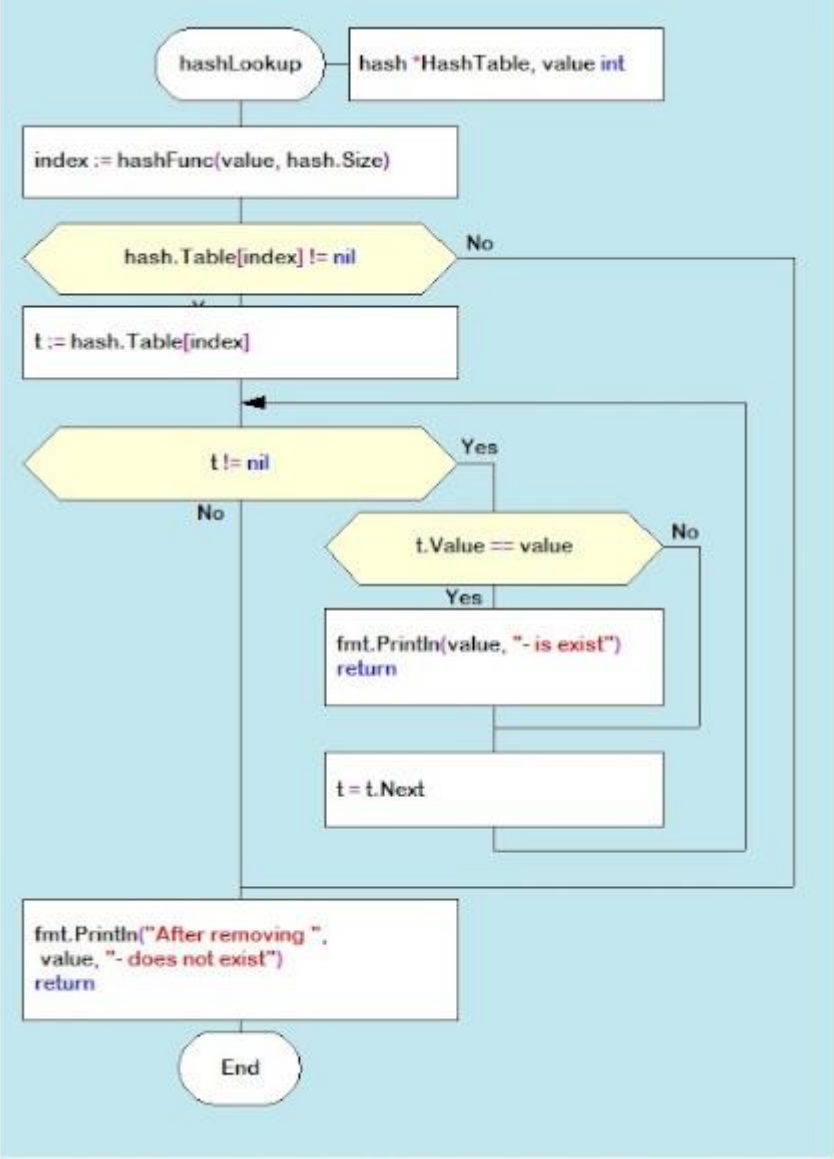
2. Compute the object's hash code by passing it through the hash function.

3. Get the bucket hash indices where the objects will be saved.

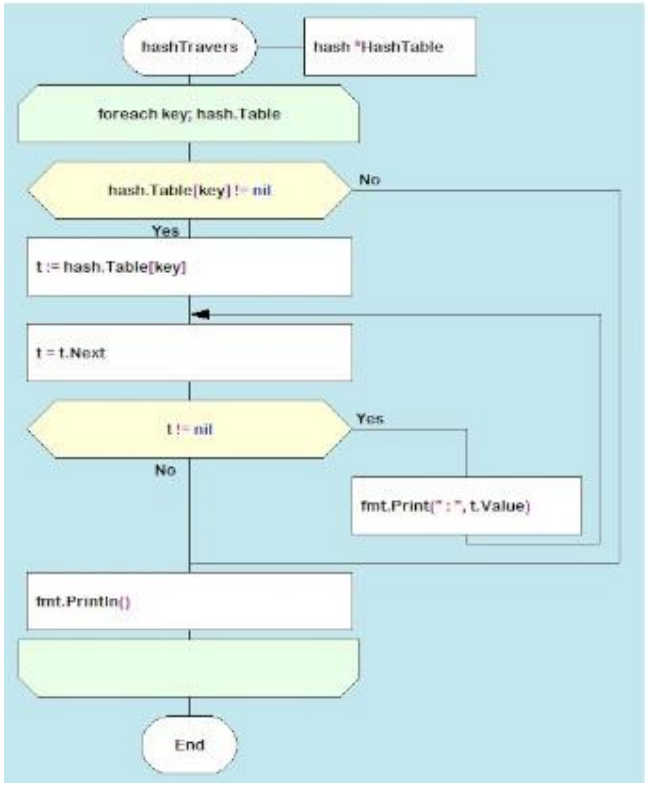
4. Save these objects in the designated bucket.

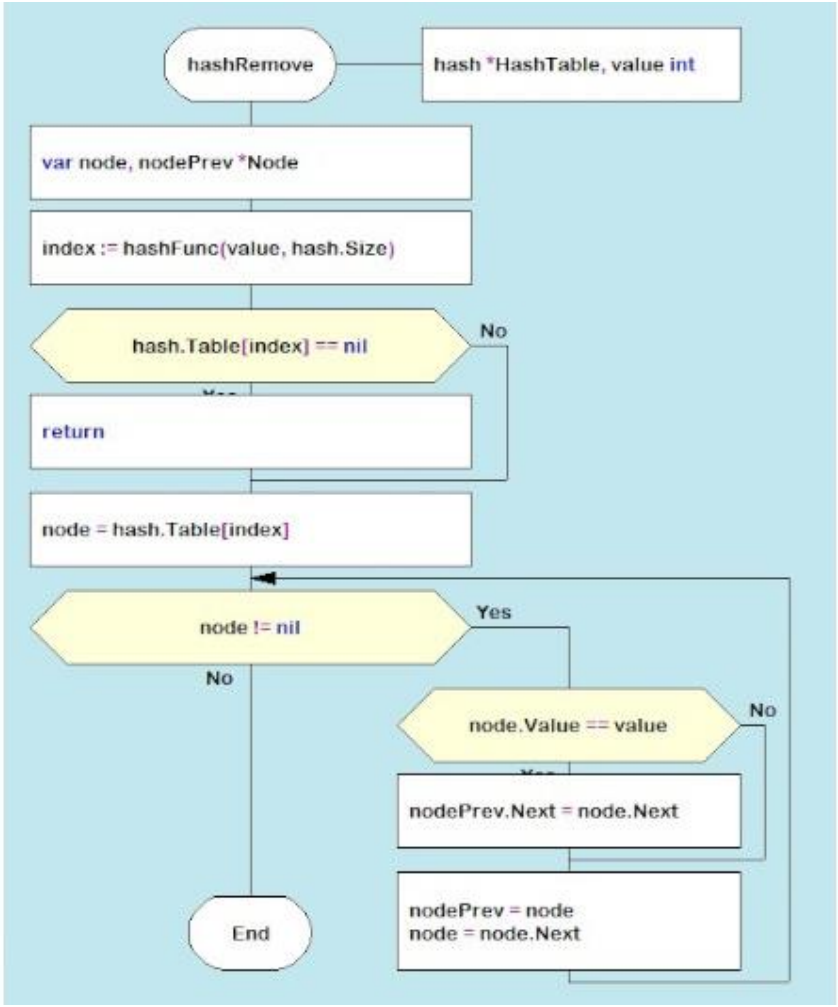
DRAKON-diagrams of algorithms for implementing the main functions of working with hash-tables are shown in Figure 7.6.

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.):

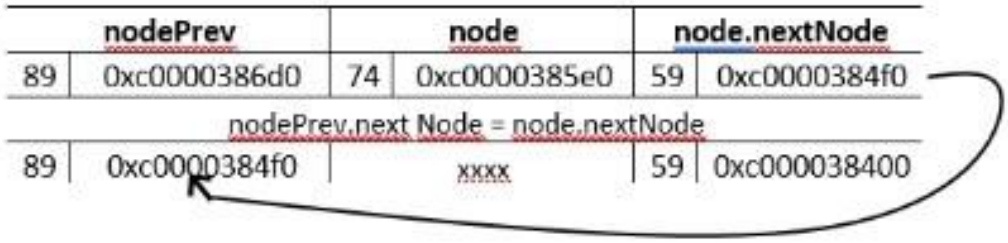


Figure 7.7. Deletion item from hash-table

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 (strc*onv.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:

- open addressing hashing with linear testing;

- chain hashing or so-called multidimensional hashing.

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

SECTION 7. BASIC SEARCH ALGORITHMS

7.1. Main characteristics of 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.

7.2. 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).

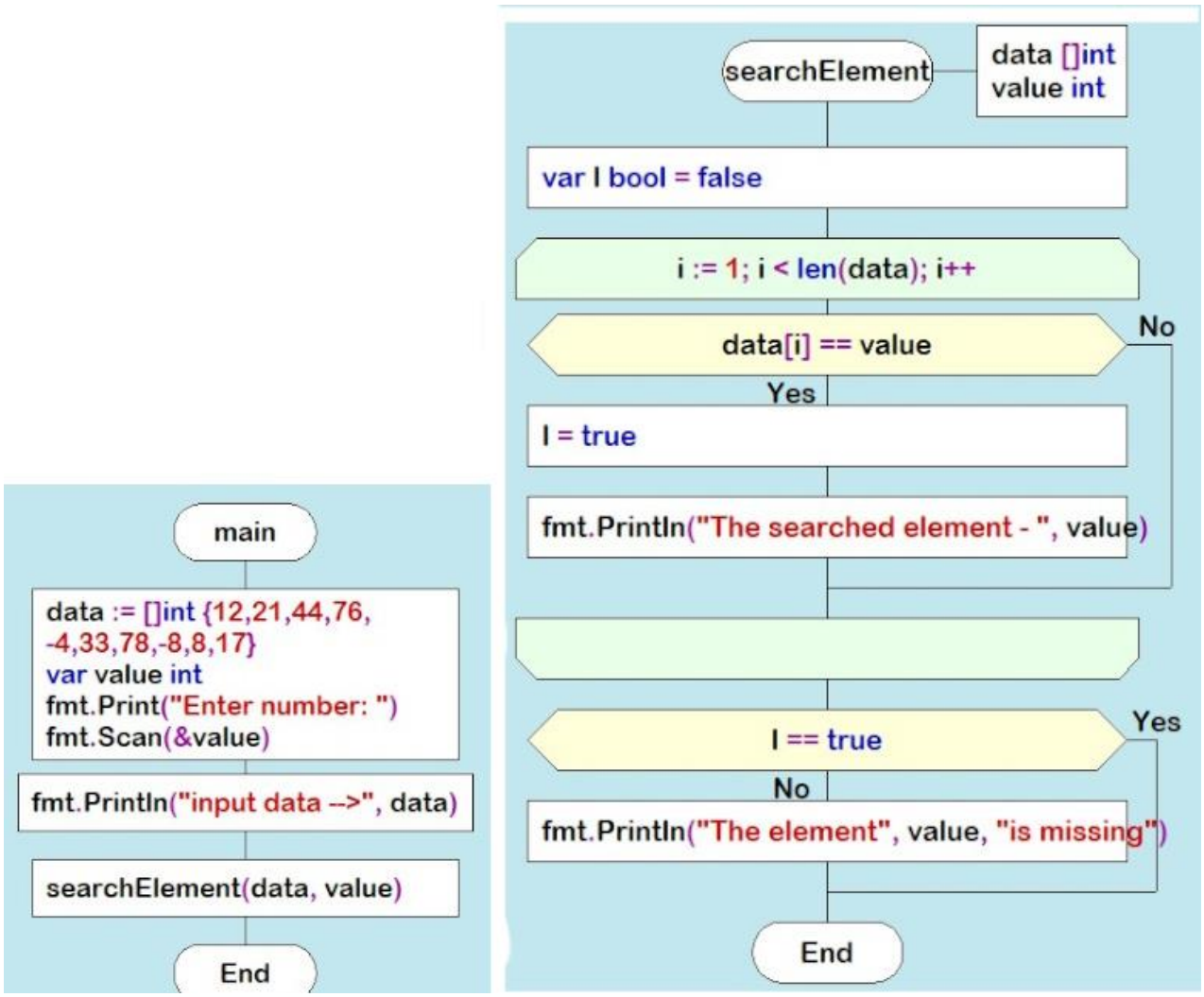


Figure 7.1. DRAKON-diagrams of linear search algorithm

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.

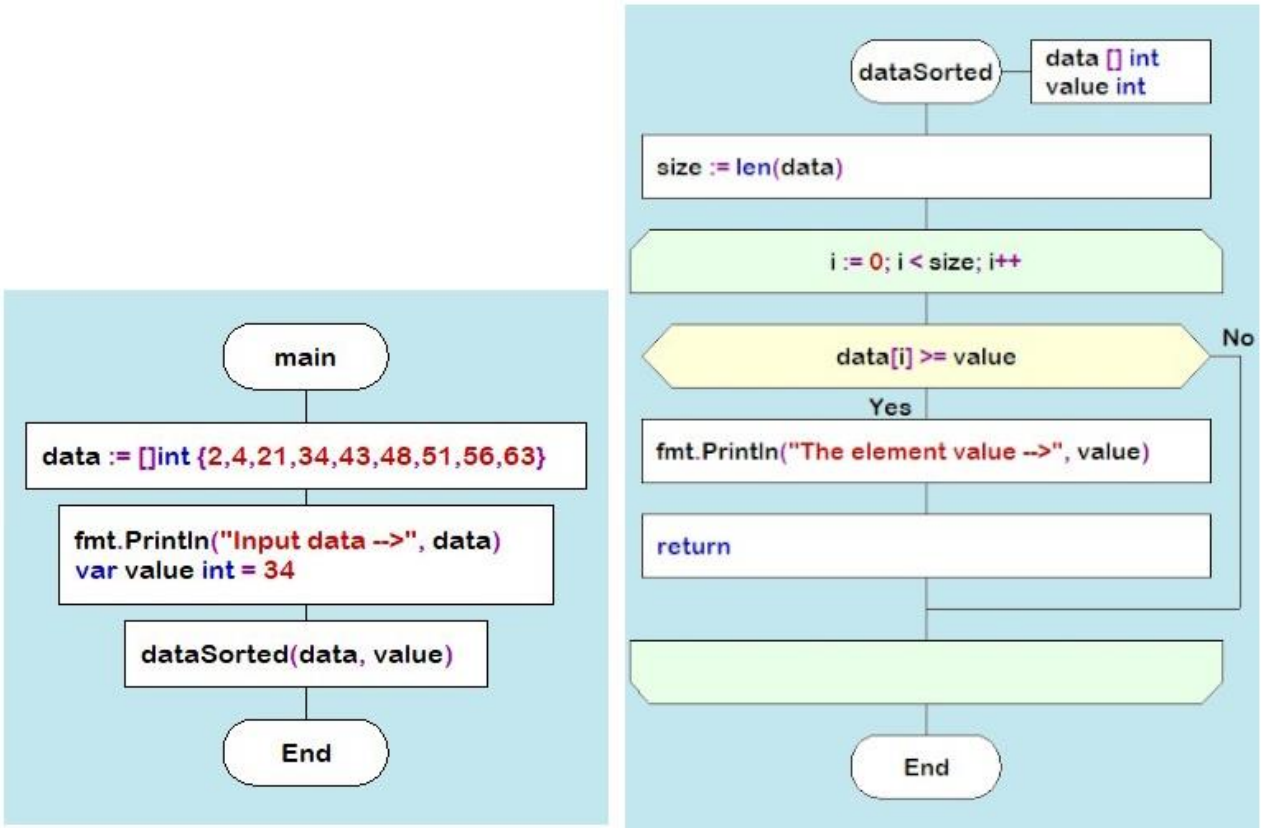


Figure 7.2. DRAKON-diagram of searching sorted slice algorithm

7.3. 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.

2. 3. The search is simply that the value of the middle element in the selected half is again determined and compared to the key.

3. 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):

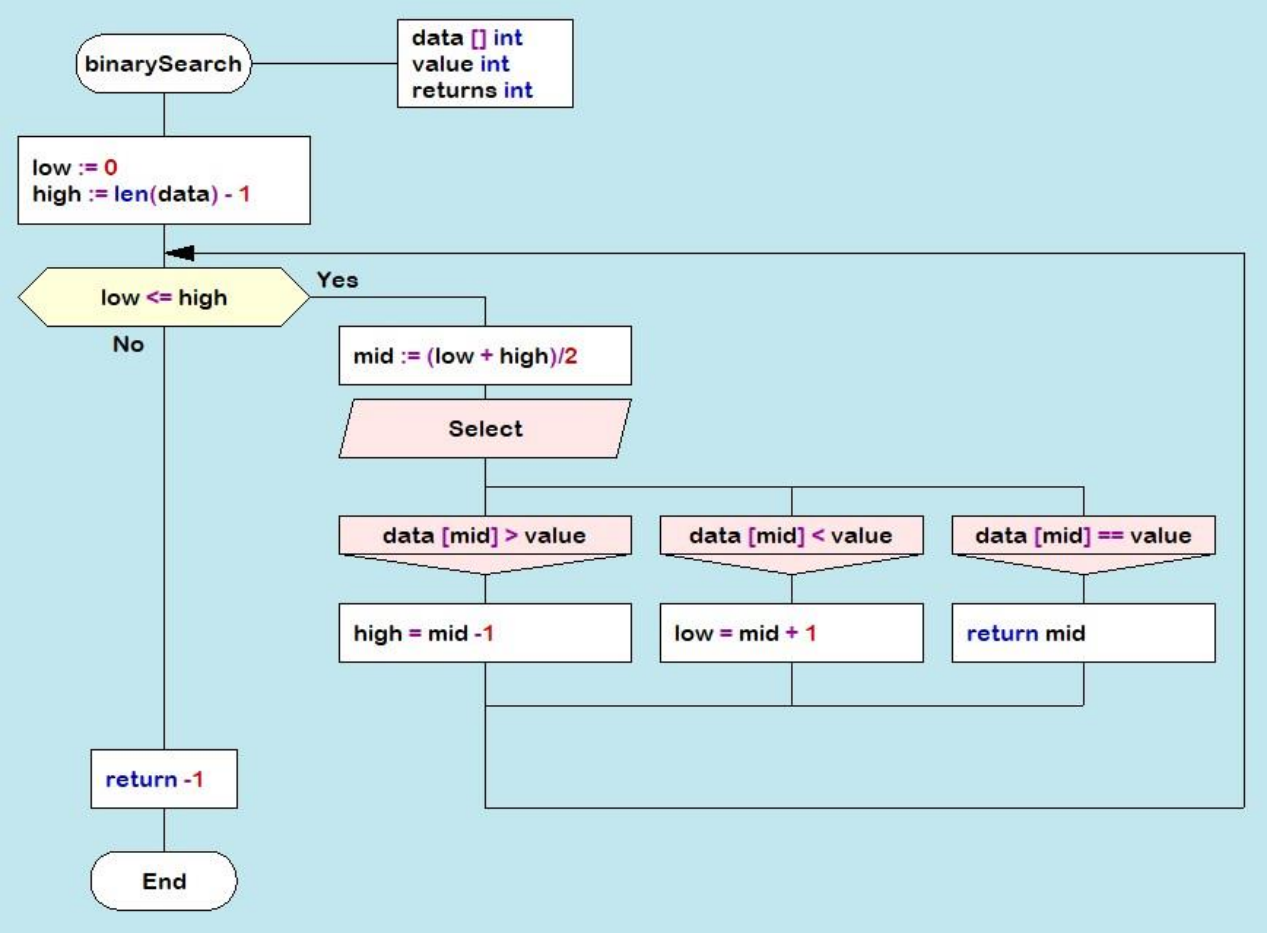


Figure 7.3. DRAKON-diagram of *binarySearch* algorithm

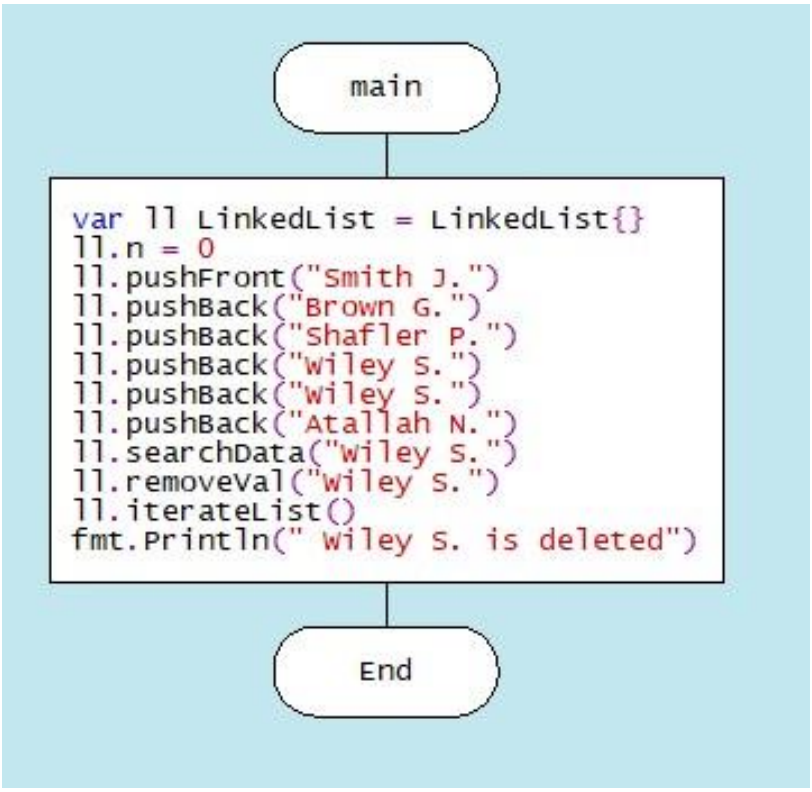
Time complexity of binary search algorithm belongs to class O(log n). The way to interpret this is that the asymptotic increase in the time taken by a function to perform a given input set of size n will not exceed log n.

Space complexity: O(1). That is, no extra space required.

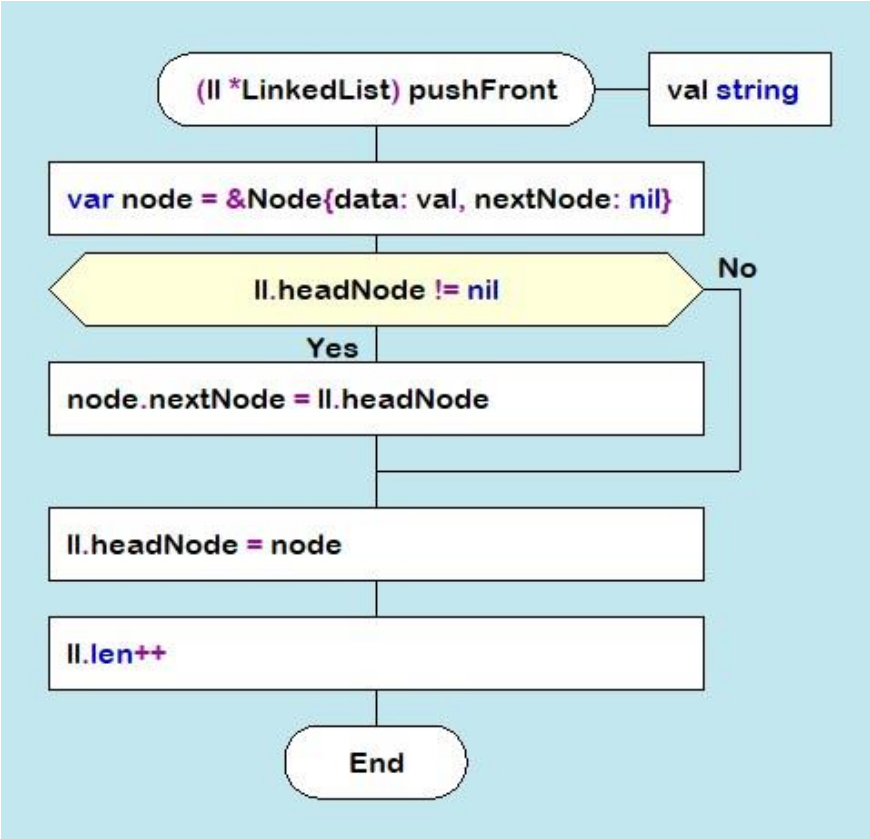
7.4. Searching in Single-Linked List

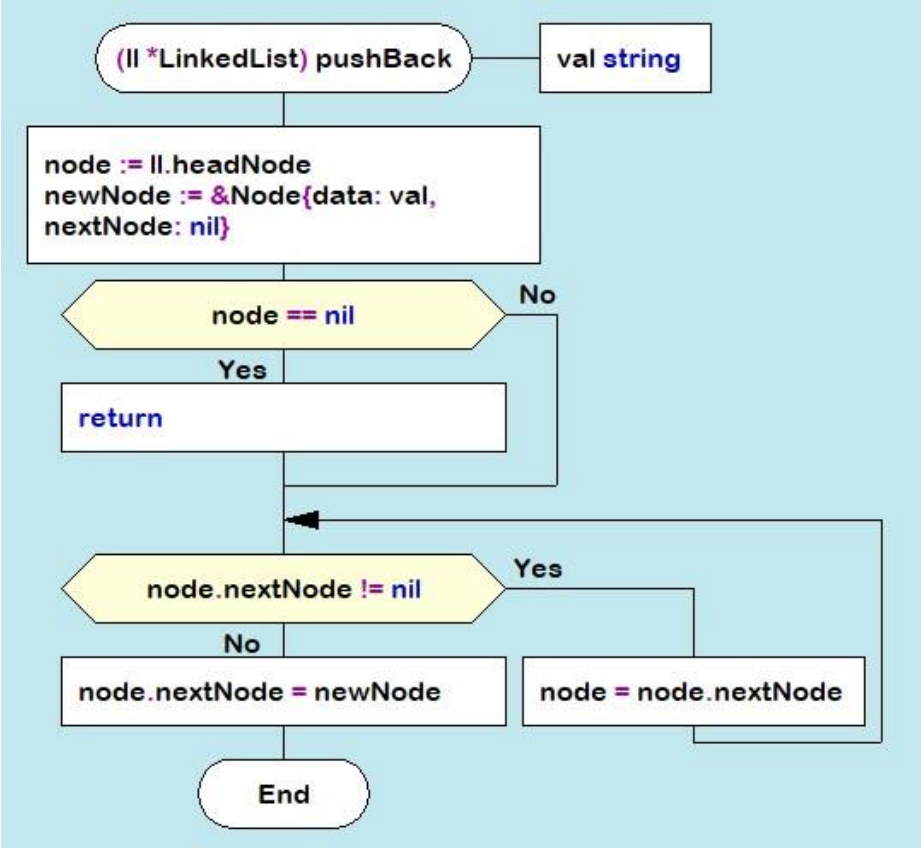
There are three possibilities for a Single-Linked List. First, the desired value is missing from the list, second, the desired value is encountered once and, third, the desired value is encountered repeatedly. You can also set the task of removing duplicates, i.e., nodes that are redundant.

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 Drakon-diagrams are presented in Figure 7.4 a,b,c,d:

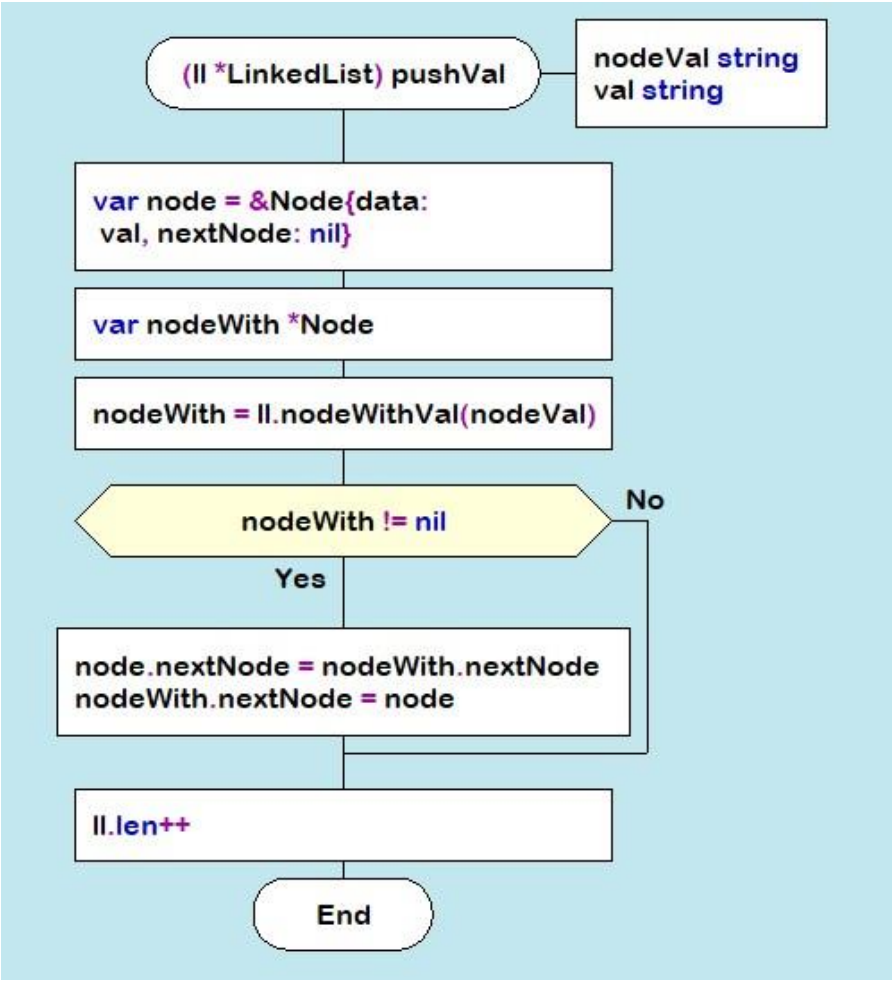


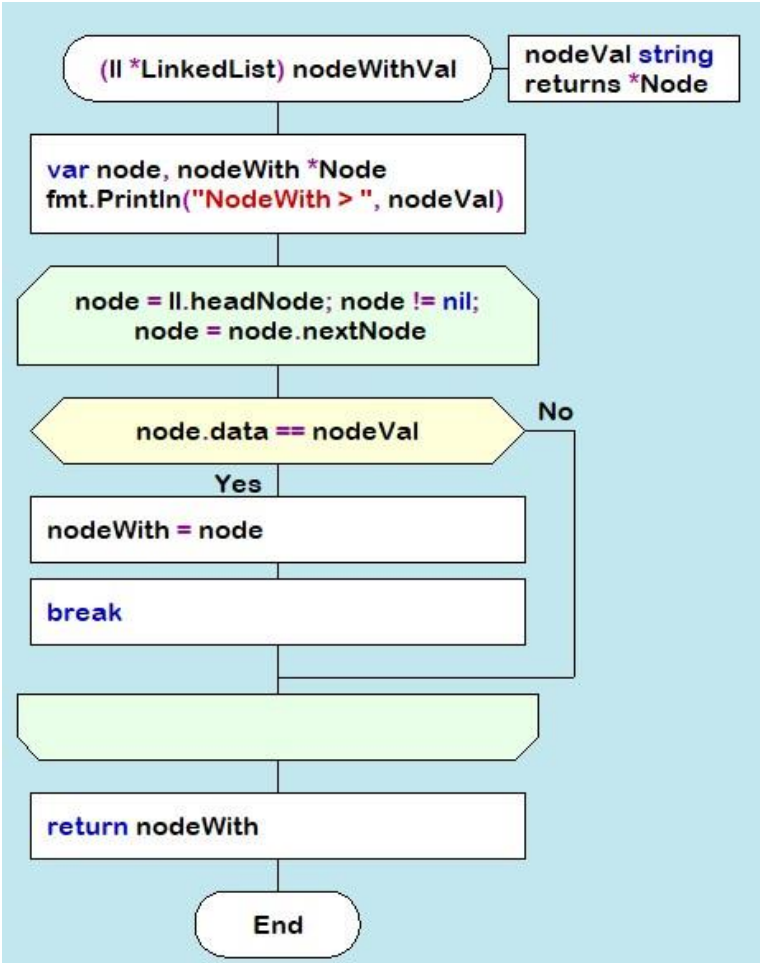
a) Function *main()*



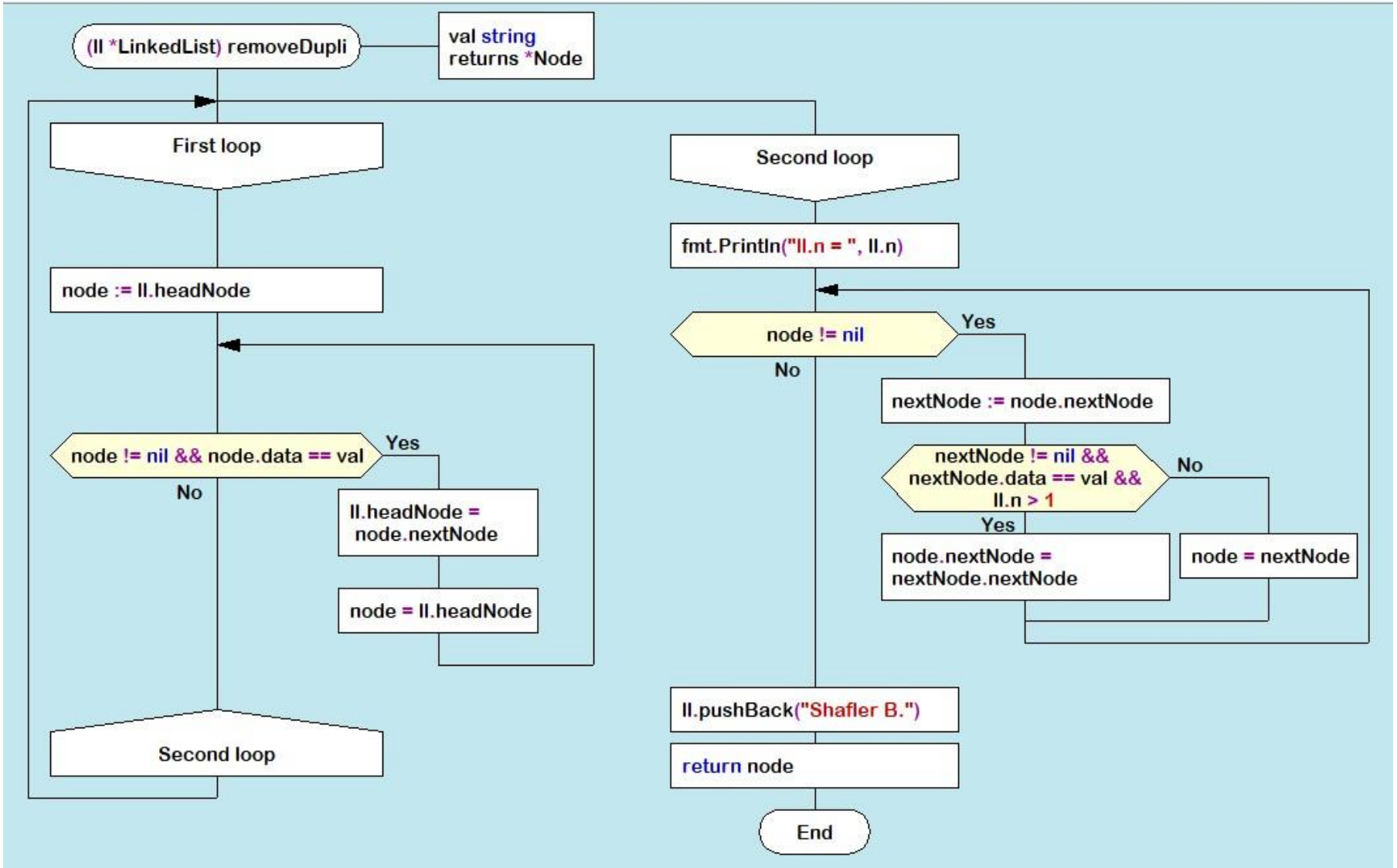


b) Functions inserting a node at the beginning and end of the list *pushFront* and *pushBack*

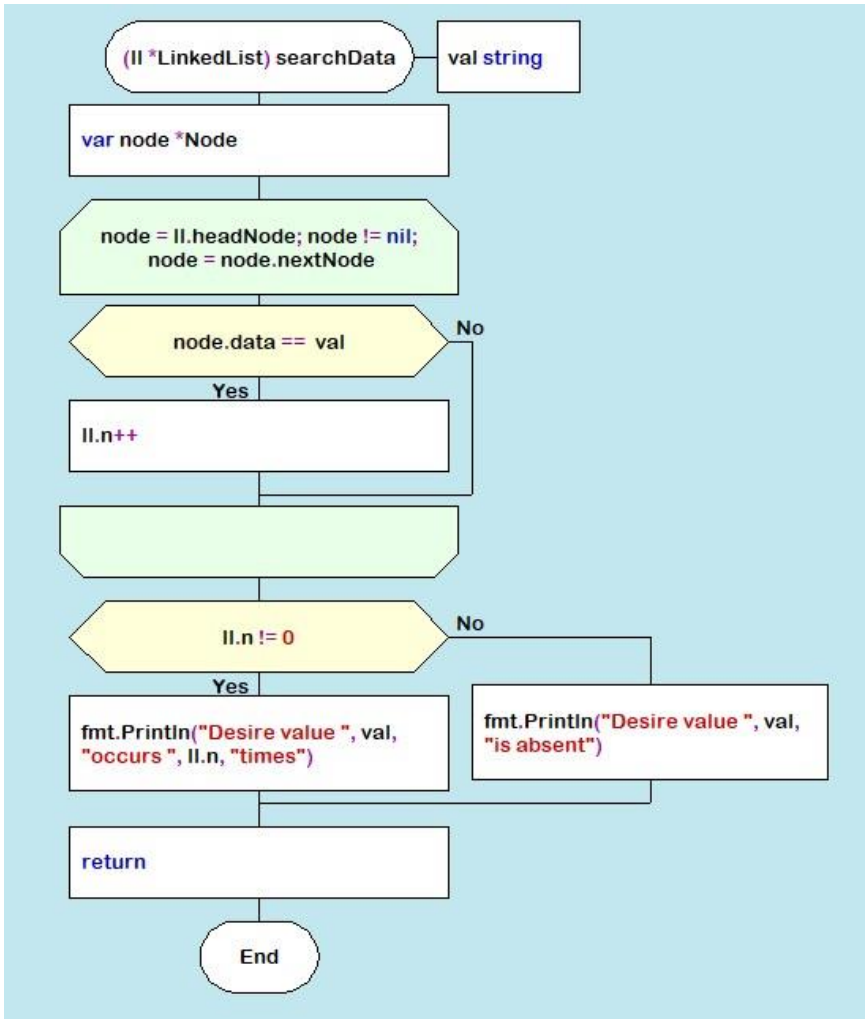


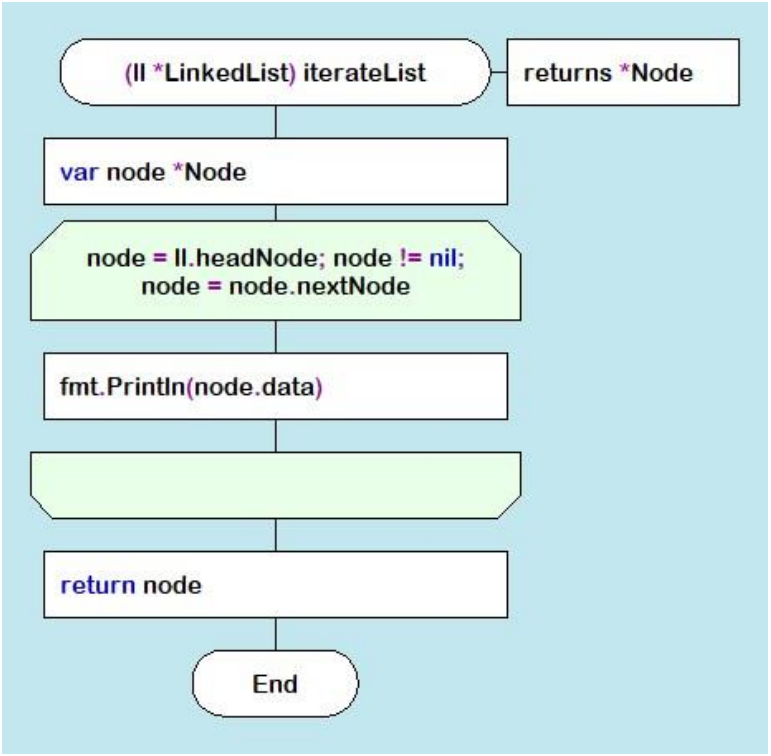


c) Funcrions inserting a node after a specified node *pushVal(val)* and *nodeWithVal(val)*



d) Function c) Function for deleting a specified node *removeDupli(val)*





e). Function of searching *searchData(val)* and list iteration *iterateList*

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

7.5. Hashing

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.

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.).

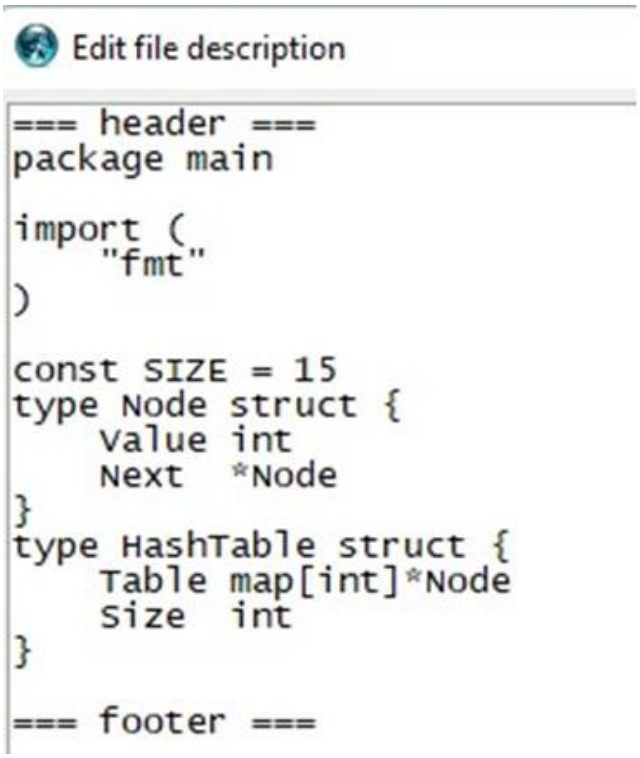


Figure 7.5. Type declaration *Node*, *HashTable* and constant *Size*

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

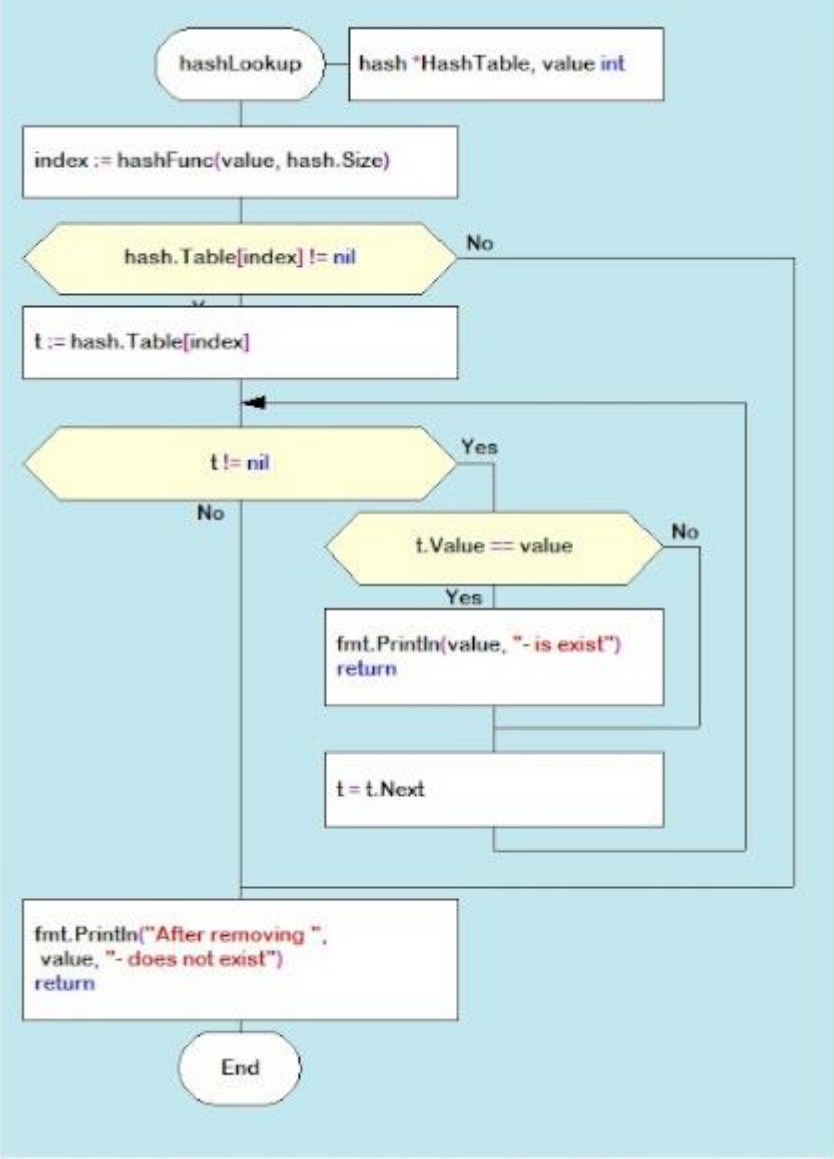
2. Compute the object's hash code by passing it through the hash function.

3. Get the bucket hash indices where the objects will be saved.

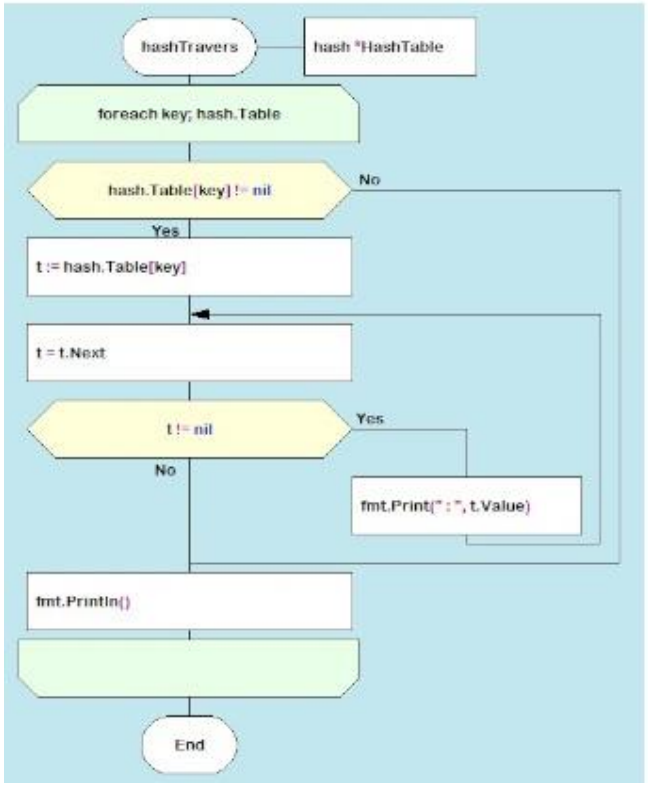
4. Save these objects in the designated bucket.

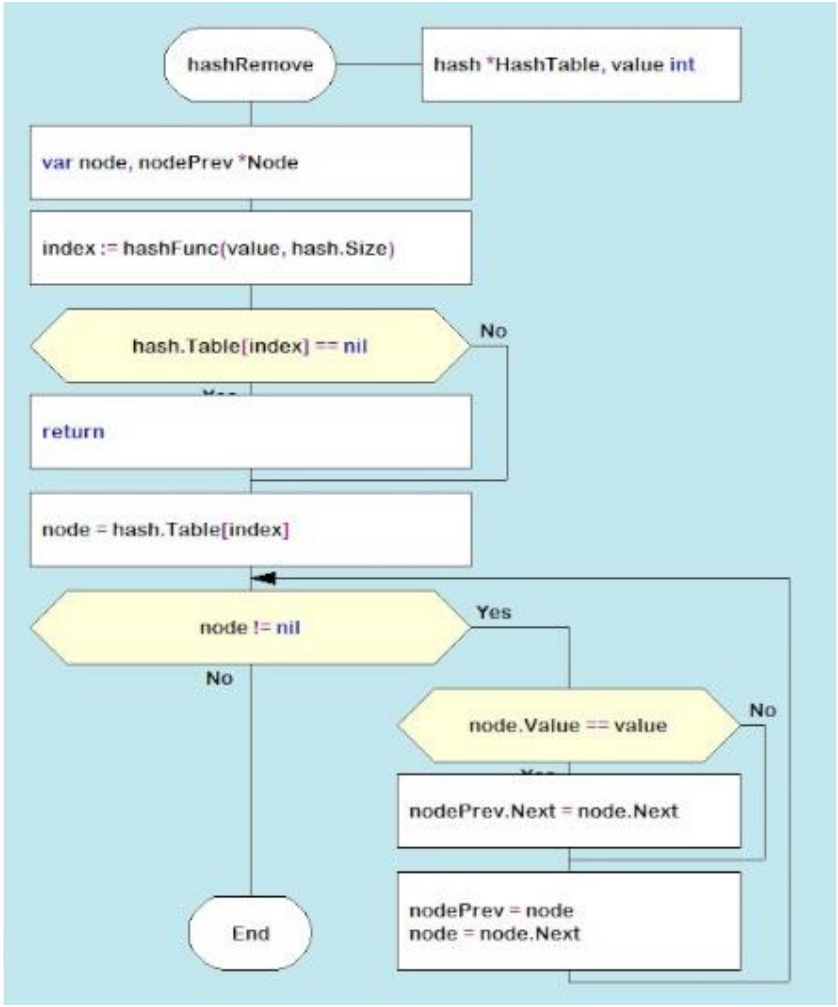
DRAKON-diagrams of algorithms for implementing the main functions of working with hash-tables are shown in Figure 7.6.

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.):

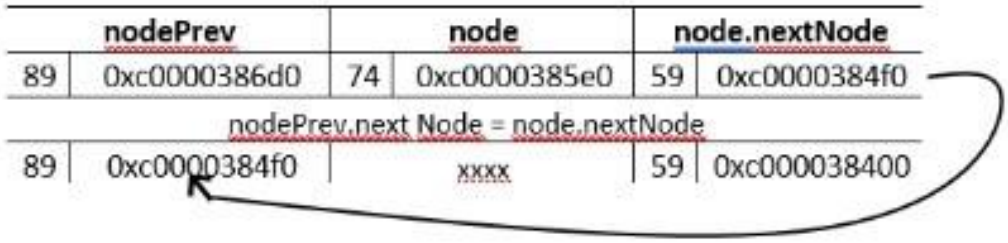


Figure 7.7. Deletion item from hash-table

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 (strc*onv.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:

- open addressing hashing with linear testing;

- chain hashing or so-called multidimensional hashing.

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

SECTION 9. GRAPHES

9.1. General information and basic terminology

This section covers data structures such as graphs. In mathematics, a graph is a structure consisting of vertices (nodes) and edges (connections) that connect these vertices. In general, graphs share many similarities with trees, and one can consider trees as a special case of graphs. However, the practical value of graphs in solving real-world problems is much greater. Many tasks can be reduced to considering a set of objects whose properties are described by the relationships between them. Such objects include electrical and electronic circuits, circuit boards, road maps, aviation routes, descriptions of constructions, and games. Among the tasks that can be solved using graphs are finding the shortest path between two vertices, solving the problem of maximum capacity of pipelines, road networks, or computer networks, distributing N workers to perform M different types of work, and choosing the most efficient method of problem solving, among others.

More strictly, a graph G is given by a set of vertices {V} and a set of edges {E} connecting all or part of these vertices. Thus, a graph G is completely defined as {V, E}. If edges are oriented, they are called arcs, and a graph with such edges is called an oriented graph (Figure 9.1 a). If the edges have no orientation, the graph is called an undirected graph:

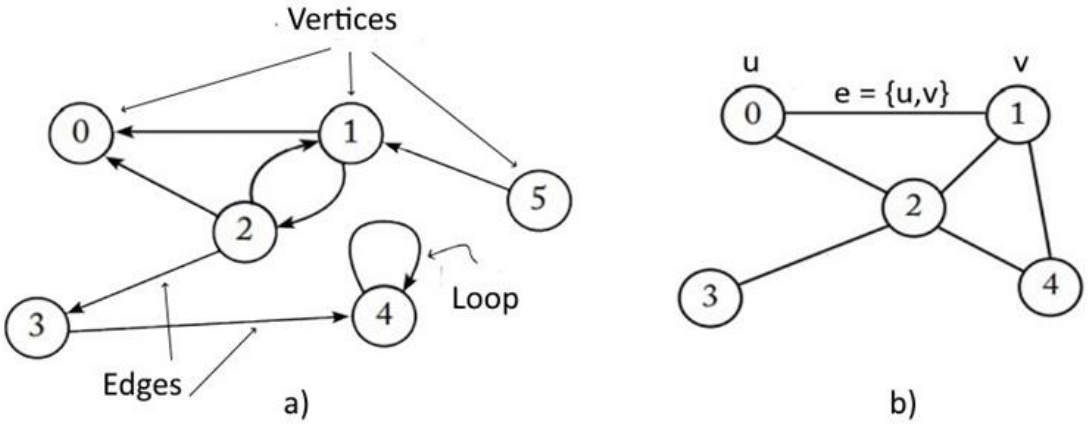


Figure 9.1. The view of a) -undirected; b) - directed graph

The elements of a graph are called vertices and edges. The number of vertices in the graph is called the order, and the number of edges is called the size of the graph. The endpoints of an edge ***e = {u,v}*** are called vertices ***(u,v)***, and two endpoints of the same edge are called adjacent. Two edges are called adjacent if they have a common endpoint. Two edges are called multiples if the sets of their endpoints are the same. An edge is called a loop if its ends coincide, that is, e = {u,u}. If the vertex is the beginning or end of the edge, then they (vertex and edge) are incident. The number of edges incident to a vertex is called the vertex degree (Figure 9.2).

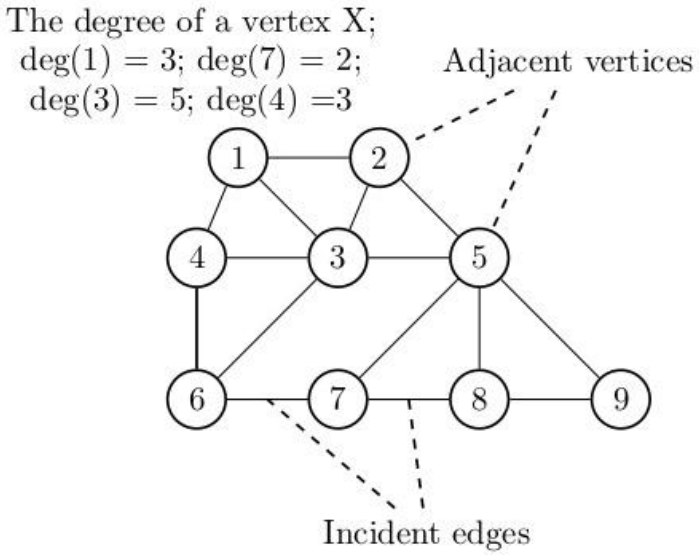


Figure 9.2. Basic graph parameters

9.2. Graph representation methods

Solving problems related to processing a data set organized in the form of graphs requires modeling them in computer programs. In principle, a set of vertices can be stored in an array and accessed by index. You can also store vertices using a simple-linked list or other data structure. In practice, two structures are typically used to model the structure of a graph: the adjacency matrix and the adjacency list. Adjacent in the sense that such vertices are connected by one edge.

An adjacency matrix is a two-dimensional array that indicates the existence of a relationship between two vertices. If the graph has V vertices, then the adjacency matrix is a V × V array. Figure Table 9.3. Undirected and directed graphs and the corresponding adjacency matrices are shown, In particular, for a directed graph, the adjacency matrix is constructed in such a way that 1 indicates the presence of a connection between the vertices, 0 - its absence. For instance, in an undirected graph, vertex (2) is adjacent to vertices (1), (3), (4), and (5), while in a directed graph, vertex (2) is adjacent to vertices (1), (3), and (4).

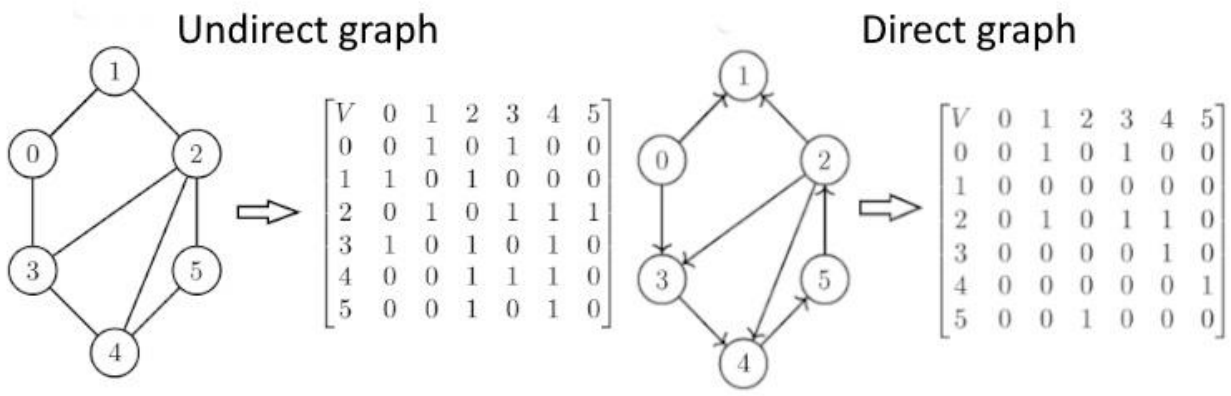


Figure 9.3. Graph representations in the adjacency matrix

The representation of a graph in the adjacency matrix is problematic. Foremost, when constructing a matrix, it is necessary to know in advance the number of vertices in the graph, which leads to the need to build a new matrix each time new vertices are inserted. In addition, the adjacency matrix consists mainly of zeros, resulting in inefficient memory usage, and if the graph contains V vertices, then memory for V^2 elements must be allocated.

In some cases, a more efficient way to represent a graph is to use a adjacency list. Undirected and directed graphs and their adjacency lists are depicted on Figure 9.4.

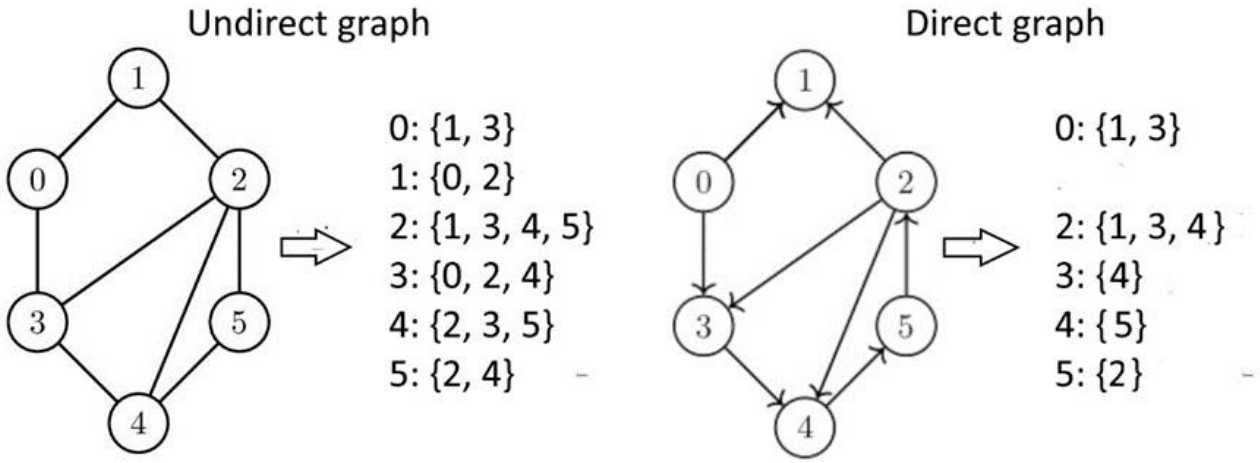


Figure 9.4. Graph representation in the adjacent list

The matrix and adjacency list for an undirected graph differ from a directed graph in that the '1' labels n the matrix are applied to all adjacent vertices (Figure 9.4.). In most practical cases, graph processing problems contain values that are associated with either the vertices of the graph or its edges. For example, in the problem of finding the optimal path between two points, the edges of the graph need to be loaded with such data as the distance between these vertices and the cost of travel per 1 km (Figure 9.5):

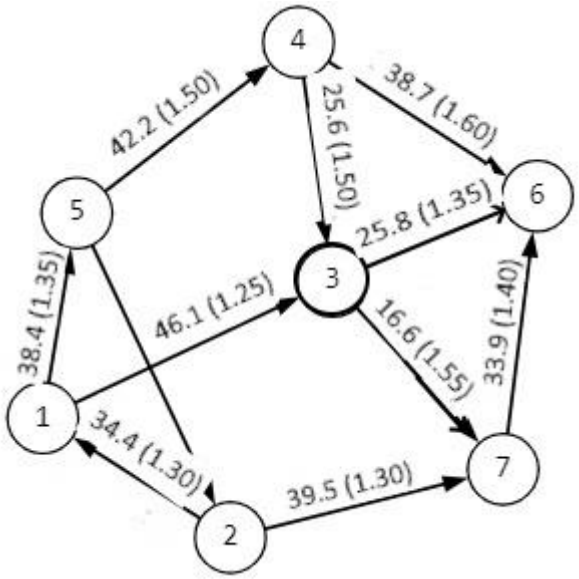


Figure 9.5. Loaded graph

9.3. Graph interface and algorithms

The interface of a graph as a data structure is a set of methods that define the basic operations available for working with a graph. This interface includes methods for getting the number of vertices and edges in a graph, accessing a specific edge between vertices, adding and removing vertices and edges, and methods for working with iterators to get information about incoming and outgoing edges for a particular vertex. The graph interface allows you to work with graph data structures and implement them in accordance with the requirements of a specific task.

In the Golang programming language, a graph interface can be implemented using structures and methods, the content of which is determined by a specific task and the choice of basic data structures from which new types will be created. In the simplest case, working with graphs requires the inclusion of fields in the graph structure to display vertices and/or edges. To create ***a Graph*** type With support for representing a graph using a list of contiguities, various basic structures of the Golang language are used (slices whose elements are Linked Lists, maps, two-dimensional slices [] []).

Let's look at some examples of the construction of a directed graph as shown in Figure 9.6. using the basic structures mentioned above (Table 9.1.).

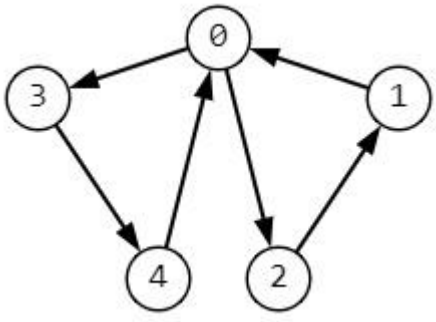


Figure 9.6. Directed graph for interface illustration

When choosing how to implement the graph interface using the Golang language, you should be guided by the following criteria:

1. Linked List:

- it is used when you want to store relationships between vertices in a graph as an adjacency list;

- suitable for implementing graphs with a large number of edges and a small number of vertices;

- allows you to efficiently add and remove edges between vertices.

2. Map:

- it is used when you want to store relationships between vertices of a graph in the form of a dictionary, where the keys are vertices and the values are their neighbors;

- suitable for implementing graphs with an arbitrary number of vertices and edges;

- allows you to quickly get a list of neighbors for a given vertex.

3. Slice:

- it is used when you want to store relationships between vertices in a graph as slices, where the slice indexes correspond to vertices and the values are their neighbors;

- suitable for implementing graphs with a fixed number of vertices;

- allows you to efficiently get a list of neighbors for a given vertex.

The specific implementation method depends on the specific requirements for the graph, such as the number of vertices and edges, the operations that will be performed most often (adding vertices, adding edges, finding neighbors, and so on), and the specifics of the problem in which the graph will be used.

Table. 9.1. Basic structures for creating graph interfaces

Linked List	Map	Slice

package main
import "fmt"
type Node struct {
value int
next *Node
}
type Graph struct {
vertices []*Node
}
	package main
	import "fmt"
	type Graph struct {
	vertices map[int][]int
	}
		package main
		import "fmt"
		type Graph struct {
		vertices [][]int
		}

Linked List	Map	Slice
package main import "fmt"	package main import "fmt"	package main import "fmt"

```
type Node struct {
value int
next *Node
}
type Graph struct {
vertices []*Node
}

func (g *Graph) addVertex(value int) {
newNode := &Node{value: value}
g.vertices = append(g.vertices, newNode)
}
func (g *Graph) addEdge(v1, v2 int) {
node1 := g.vertices[v1]
node2 := g.vertices[v2]
newNode := &Node{value: v2}
newNode.next = node1.next
node1.next = newNode
newNode = &Node{value: v1}
```

```
type Graph struct {
vertices map[int][]int
}

func (g *Graph) addVertex(vertex int) {
g.vertices[vertex] = []int{}
}
func (g *Graph) addEdge(v1, v2 int) {
g.vertices[v1] = append(g.vertices[v1], v2)
g.vertices[v2] = append(g.vertices[v2], v1)
}
func (g *Graph) getNeighbors(vertex int) {
fmt.Println(g.vertices[vertex])
}
func main() {
graph := NewGraph()
```

```
type Graph struct {
vertices [][]int
}

func (g *Graph) addVertex() {
g.vertices = append(g.vertices, []int{})
}
func (g *Graph) addEdge(v1, v2 int) {
g.vertices[v1] = append(g.vertices[v1], v2)
g.vertices[v2] = append(g.vertices[v2], v1)
}
```

```
	newNode.next =
node2.nextnode2.next = newNode
	func (g *Graph) getNeigh-
bors(vertex int) {
	node := g.vertices[vertex]
	for node != nil {
	fmt.Println(node.value)
	node = node.next
	}
	}
	func main() {
	graph := Graph{}
	graph.AddVertex(0)
	graph.AddVertex(1)
	graph.AddVertex(2)
	graph.AddVertex(3)
	graph.AddVertex(4)

	graph.AddEdge(0, 1)
	graph.AddEdge(0, 2)
	graph.AddEdge(0, 3)
	graph.AddEdge(1, 4)
	graph.AddEdge(2, 4)
	graph.getNeighbors(4)
	}
```

```
	graph.adVertex(0)
	graph.addVertex(1)
	graph.addVertex(2)
	graph.addVertex(3)
	graph.addVertex(4)
	graph.addEdge(0, 1)
	graph.addEdge(0, 2)
	graph.addEdge(0, 3)
	graph.addEdge(1, 4)
	graph.addEdge(2, 4)

	graph.getNeighbors(0)
	}
```

```
	func (g *Graph) getNeighbors(vertex
int) {
	fmt.Println(g.vertices[vertex])
	}
	func main() {
	    graph := Graph{}
	    graph.addVertex()
	    graph.addVertex()
	    graph.addVertex()

	    graph.addEdge(0, 1)
	    graph.addEdge(1, 2)

	    graph.getNeighbors(0)
	}
	func main() {
	graph := Graph{}
	graph.addVertex()
	graph.addVertex()
	graph.addVertex()
	graph.addVertex()
	graph.addVertex()

	graph.addEdge(0, 1)
	graph.addEdge(0, 2)
	graph.addEdge(0, 3)
```

		graph.addEdge(1, 4) graph.addEdge(2, 4) graph.getNeighbors(0) }

9.4. Basic graph algorithms

Due to the widespread use of graphs, there are a large number of algorithms for processing them. Among the problems solved within the framework of graph theory are:

- Definition of the graph and its properties;
- Actions with graphs;
- Routes, chains and cycles, contours;
- Calculation of graph characteristics;

Let's have a look at the main algorithms used for these tasks, implemented, as before, within the framework of DRAKON + Golang hybrid programming. Once the graph has been constructed and its properties have been checked, the problem of traversing the graph through its vertices can naturally arise. The general task is formulated as follows: to traverse the graph, starting from a given vertex and moving along the edges to other vertices, in order to visit all vertices.

There are two main ways of traversing graphs: depth-first traversal (dfs) and breadth traversal (bfs), which ensure that all connected vertices are traversed. The difference between depth-first and breadth-first search is that the result of a depth-first search algorithm is a route that can sequentially traverse all the vertices of the graph that are accessible from the initial vertex. This is fundamentally different from breadth-first search, which traverses all the vertices of one level first, and then traverses the vertices of the next levels sequentially.

9.4.1. Depth-first traversal (DFS)

a). Graph Formation

Recall that the depth-first search algorithm consists of sequentially traversing the vertices of the graph accessible from the initial vertex. To implement this algorithm, the graph will be represented using an adjacency list (Table 9.1). A description of the types of variables used in the graph processing program is shown in Figure 9.7.

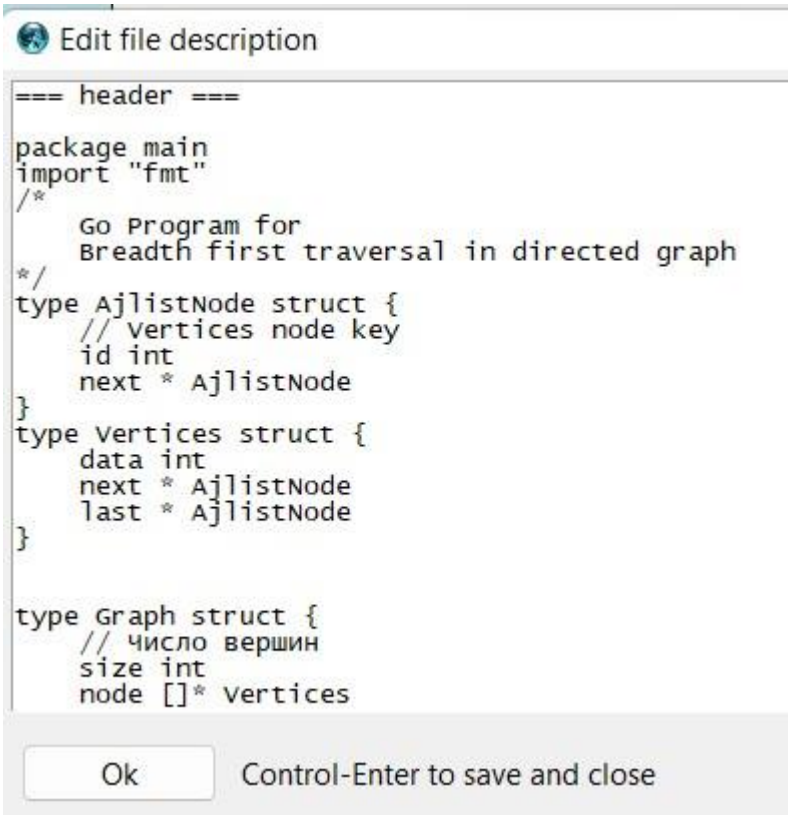


Figure 9.7. Description of depth-in variable types

The following structures are used in this description:

1. ***AjlistNode*** **is** a structure that represents a node in a linked list to represent the adjacency of a graph. It contains the following fields:

- ***id int*** is an integer value that represents the vertex identifier of the graph.

- ***next *AjlistNode*** - pointer to the next node in the linked list.

2. ***Vertices*** is a data structure that represents the vertex of a graph in a graph processing program. It contains the following fields:

- ***data int*** is an integer value that represents the data associated with the vertex of the graph.

- ***next *AjlistNode*** is a pointer to the first node in a linked list of adjacent vertices.

- ***last *AjlistNode*** is a pointer to the last node in a linked list of adjacent vertices.

3. ***Graph*** **–** the structure describes a graph consisting of nodes of the ***Vertices*** type and having two fields***:***

- ***size*** represents the number of nodes in the graph (graph size);

- ***node*** is a slice of pointers to these nodes.

The full Drakon-diagram of the graph depth-first traversal algorithm is presented in Figure9.8.

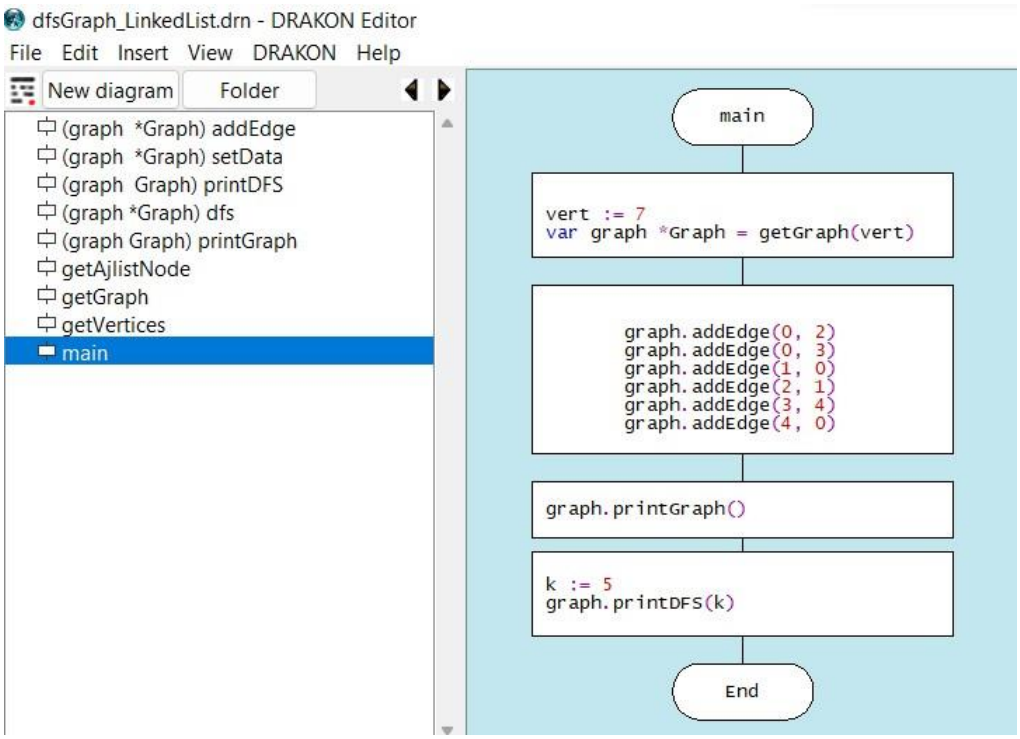


Figure 9.8. The full Drakon-diagram of algorithm in depth

In the main() function, a graph variable of the ****Graph*** pointer type is created, which describes a graph consisting of data structures of type ***Vertices***, where ***size*** represents the number of nodes in the graph, and node is a slice of pointers to these nodes (Figure 9.9.).

After creating a new instance of the graph, the ***addEdge(start, last)*** method is executed, which adds an edge between the `start` and last `nodes`. To create a node for an adjacency list, the method uses the ***var edge *AjlistNode*** variable, which initializes a unique identifier (***id)*** and a pointer to the next node (Figure 9.10.)..

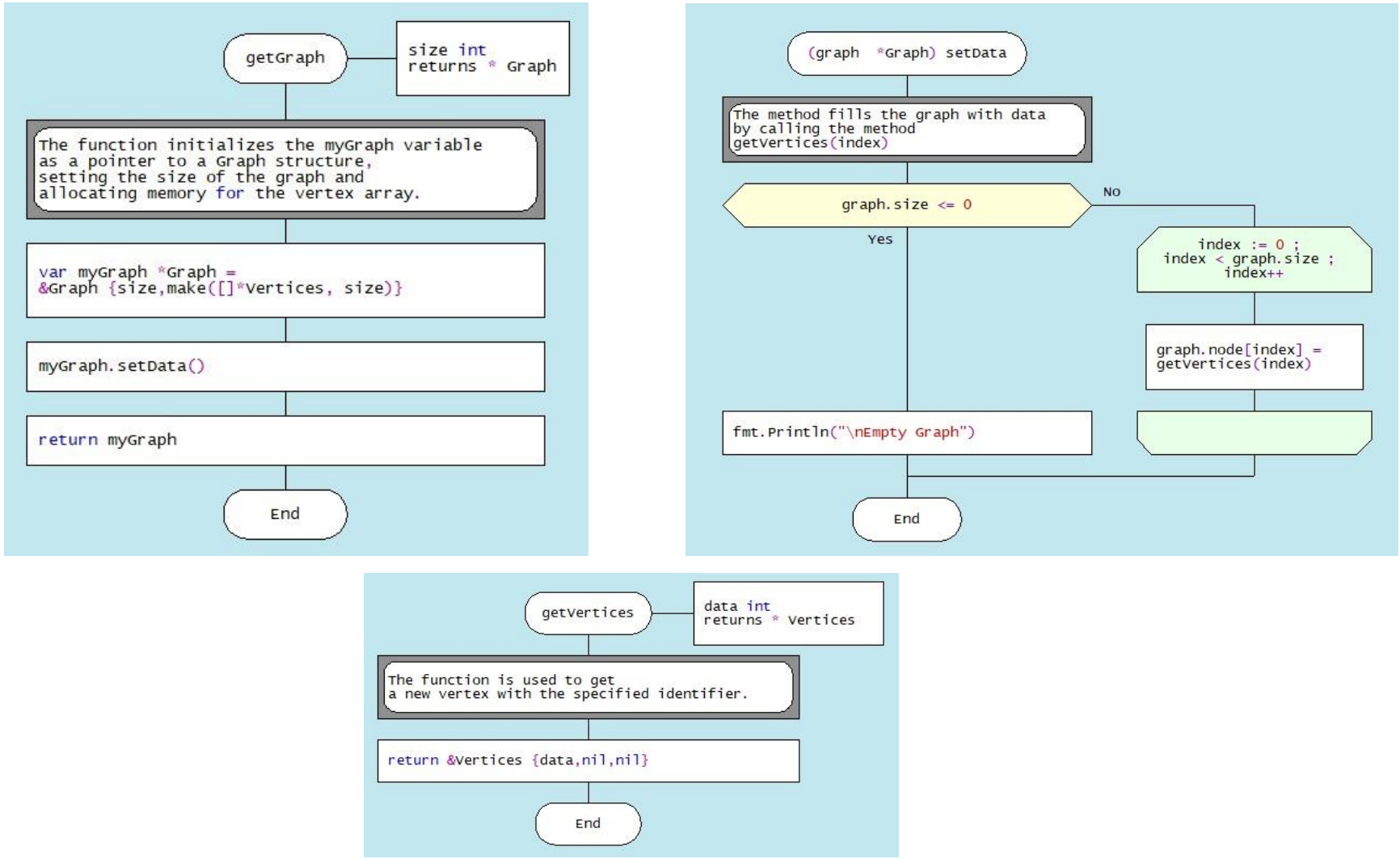
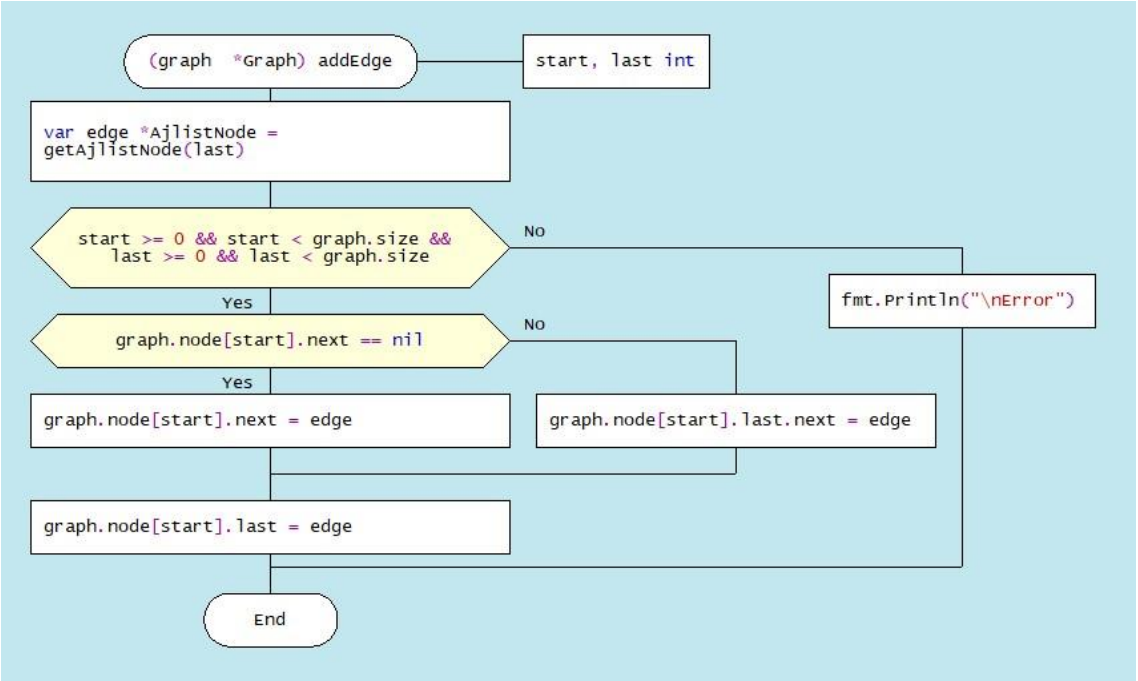


Figure 9.9. Drakon-diagrams of graph formation methods



Method addEdge((start, last int)

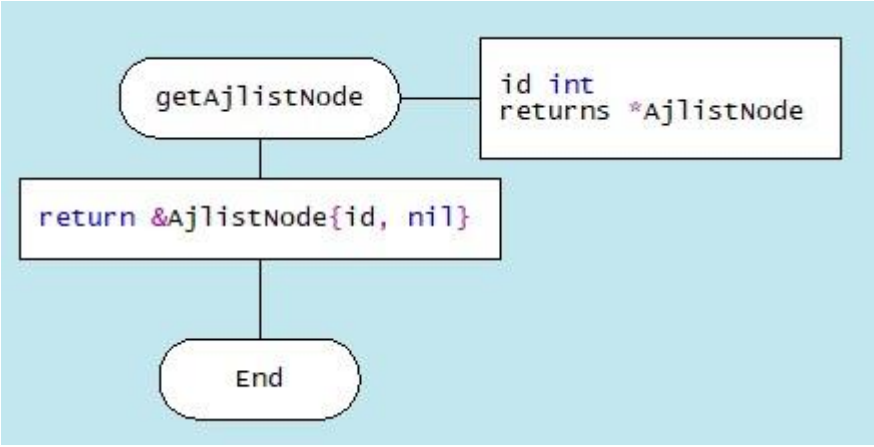


Figure 9.10. Drakon-diagrams of filling a graph with data

The ***printGraph()*** method allows you to the adjacency list of the graph, where each node is connected to its neighbors (Figure 9.11).

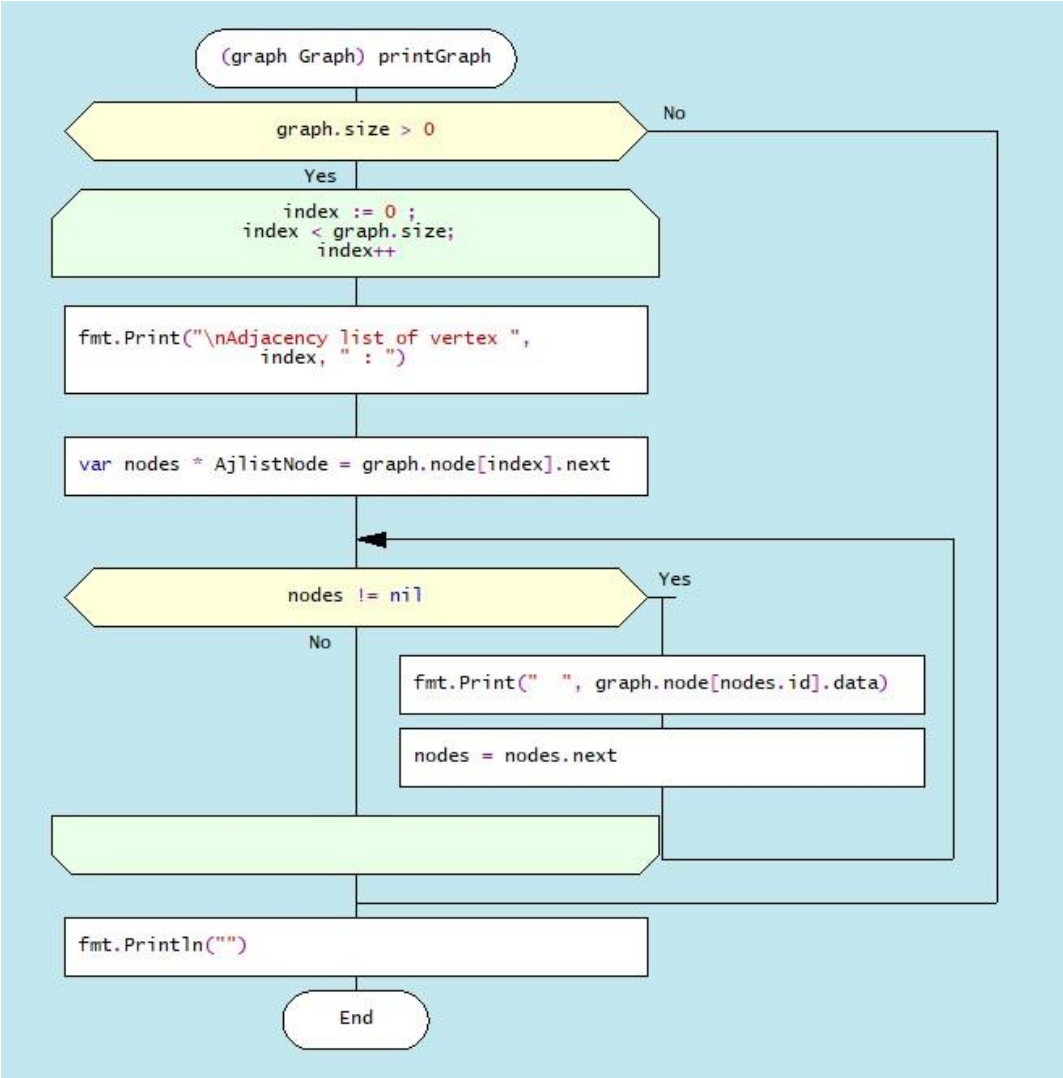


Figure 9.11. Drakon- diagram of method ***printGraph***

b). Depth-first traversal algorithm

The depth-first traversal of the graph is justified in situations where it is necessary to investigate the unknown structure of a real object represented as a graph. If the graph is oriented, the depth search builds a tree of paths from the starting vertex to all the tops accessible from it. The algorithm for traversing the graph in depth can be represented as follows. Suppose an observer in one of the vertices of a graph is given the task of traversing all its vertices. While at this vertex, the observer sees the edges emanating from that vertex. In the case of a sufficiently complex graph structure, the observer runs the risk of passing through some vertices several times and, in the end, getting stuck in a loop. To avoid such a situation, the observer should mark all the visited vertices and should not go to the peak that he has already visited. Then the algorithm might look like this:

- Select the vertex from which the graph traversal begins;
- Go to any adjacent vertex that has not been visited before;
- Run the depth-first traversal algorithm from this vertex;
- Return to the starting vertex; Repeat the process for all previously unvisited adjacent vertices.

Thus, in order to implement the algorithm, you will need to note which vertices the researcher was in and which were not. The mark will be made in the slice ***visit***, where ***visit[i] == True*** for visited vertices, and ***visit[i] == false*** for unvisited vertices. The mark "about visiting a vertice" is put when entering this vertex.

A depth-first search starts by visiting the original vertex start**,** and then recursively visits all adjacent vertices. The visit to the vertex is recorded in the logical slice visit. The depth-first traversal algorithm is based on a recursive function that retrieves a vertex from the stack and checks it for attendance. If the vertex has already been visited, the traversal continues to search the adjacency list until it reaches the dead-end vertex. An illustration of a graph traversal in depth is presented Figure9.12.

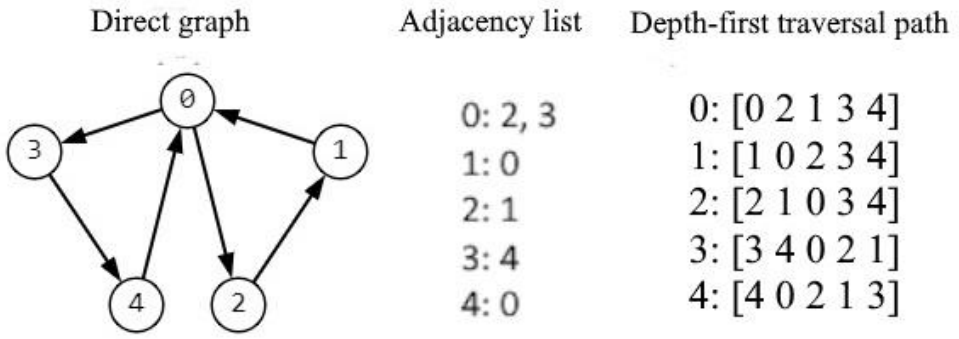


Figure 9.12. Illustration of depth-first traversal of a directed graph

The depth-traversal algorithm for undirected and directed graphs is broadly similar, but its execution may vary due to differences in the directionality of the edges and arrows in the graph. This difference lies in the fact that in undirected graphs it is possible to relate vertices in both directions without explicitly taking into account the direction. In directed graphs, each edge is counted only once in the direction from

the start vertex to the end vertex. The Drakon-diagram of the algorithm for traversing the graph in depth is shown in Figure 9.13.

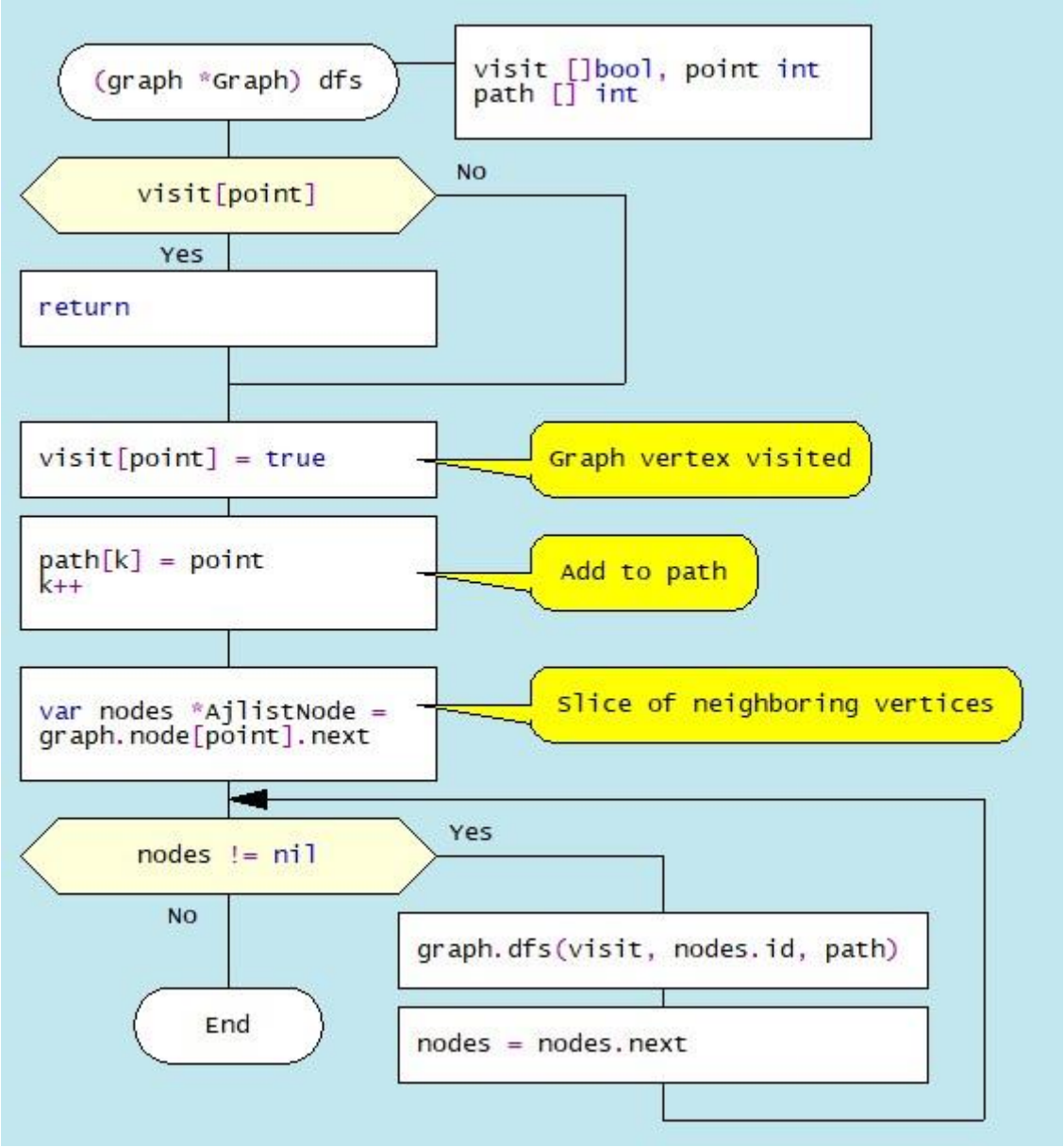


Figure 9.13. Depth-first traversal algorithm Drakon-diagram ***dfs***

9.4.2. The breadth-first search algorithm

Graph breadth-first traversal is widely used in a variety of fields, such as computer networks, web search, image processing, and other areas that require analysis of the data structure represented as a graph. A breadth-first traversal visits all vertices that are on the same level, and then moves on to the next level, and so on. Recall that a graph level is a group of vertices that are at the same distance from the initial ver-

tex. As an example, consider the algorithm for traversing a graph in the width of an undirected graph (Figure 9.14),

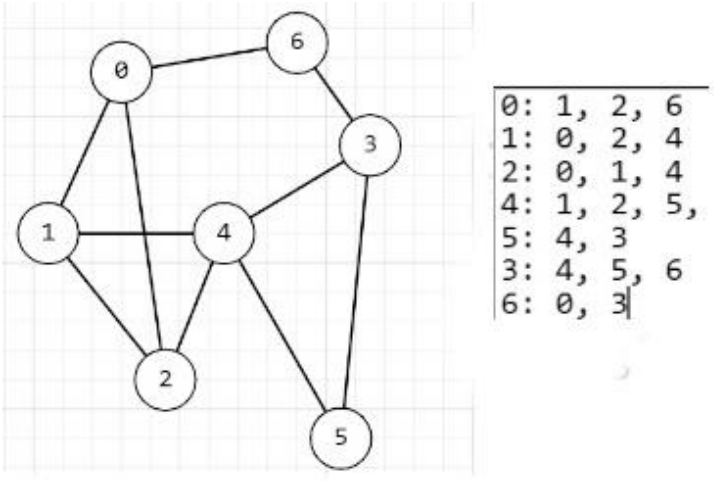


Figure 9.14. Undirected graph and Its adjacency list

In this case, unlike the previous depth-first traversal algorithm, to traverse the graph in width, we create an empty map [int] bool to track the visited vertices and an empty queue [] slice to add the visited vertices (Figure 9.15).

```
=== header ===

package main

import "fmt"

type Graph struct {
        nodes map[int][]int
        size int
}
var path [] int

=== footer ===
```

Figure 9.15. Description of variable types

The graph is generated using the ***NewGraph*** and ***addEdge(start, last) methods*** (Figure 9.16).

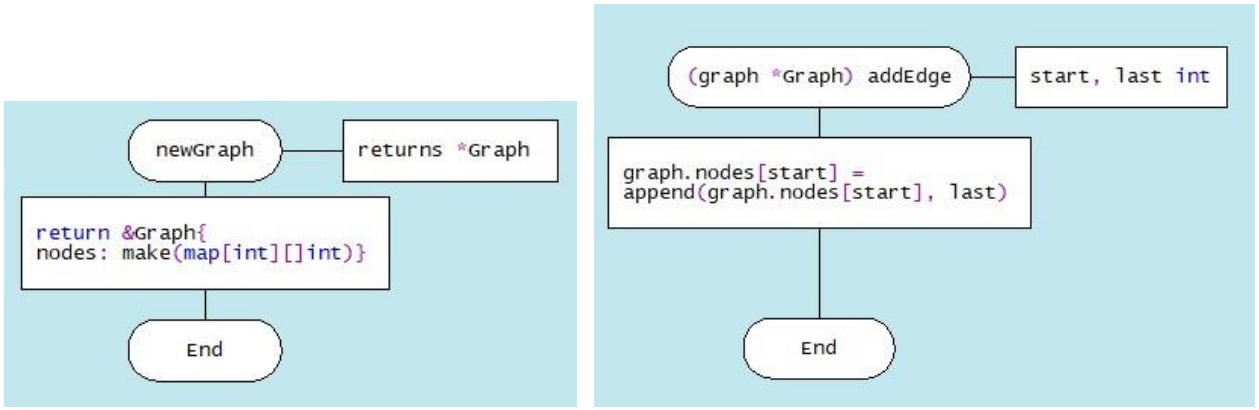


Figure 9.16. Graph generation algorithms

In the first step of the algorithm, the start vertex is added to the queue. At each width traversal step, a vertex is retrieved from the top of the queue. If this vertex has already been visited (is in the visited state), the program moves on to the next iteration of the loop, interrupting the current iteration. If the summit has not yet been visited, it is displayed and marked as visited. Then, each neighbor of that vertex that has not yet been visited is added to the queue Then the traversal continues until the queue is empty. If the queue is empty, the algorithm ends. The Drakon-diagram of the graph traversal algorithm is shown in Figure 9.17.

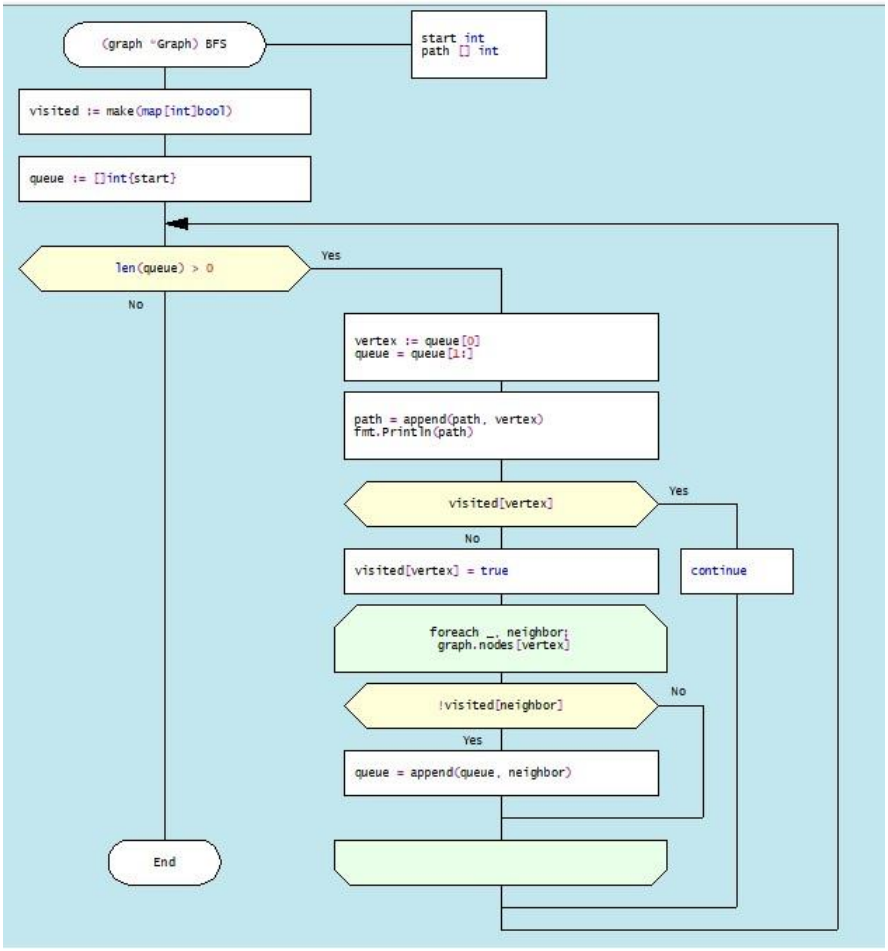
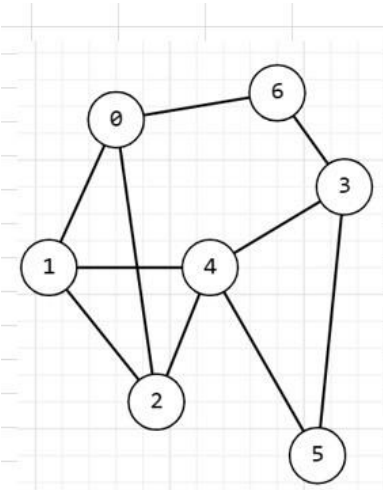


Figure 9.17. DRAKON-diagram of Breadth-first traversal algorithm

The results of traversal of the directed graph in depth and width are shown in Figure 9.18.



Vertex	adjList[vertex]	queue Before	Queue After	Result
2	1-0-4	1-0-4	[1] 0-4	**2-1**
1	0-**2**-4	0-4	[0] 4-0-4	**2-1-0**
0	**1**-6-**2**	4-0-4	[4] **0**-4-6	**2-1-0-4**
4	**1**-**2**-3-5	0-4-6	**(0-4)*** [6]-3-5	**2-1-0-4-6**
6	**0**-3	3-5	[3] 5-3	**2-1-0-4-6-3**
3	5-**4**-**6**	5-**3**	[5] 3-5	**2-1-0-4-6-3-5**
5	**3**-**4**	**3**-**5**	[]	**2-1-0-4-6-3-5**

* - visited vertices

Figure 9.18. The process of forming a graph traversal path

9.4.3. Removing a vertex from a graph

A vertex is removed from the graph by copying all vertices up to the vertex to be deleted, then the vertex to be deleted is skipped, and then the remaining vertices are copied. The Drakon-diagram of the vertex removal algorithm is shown in Figure 9.19.

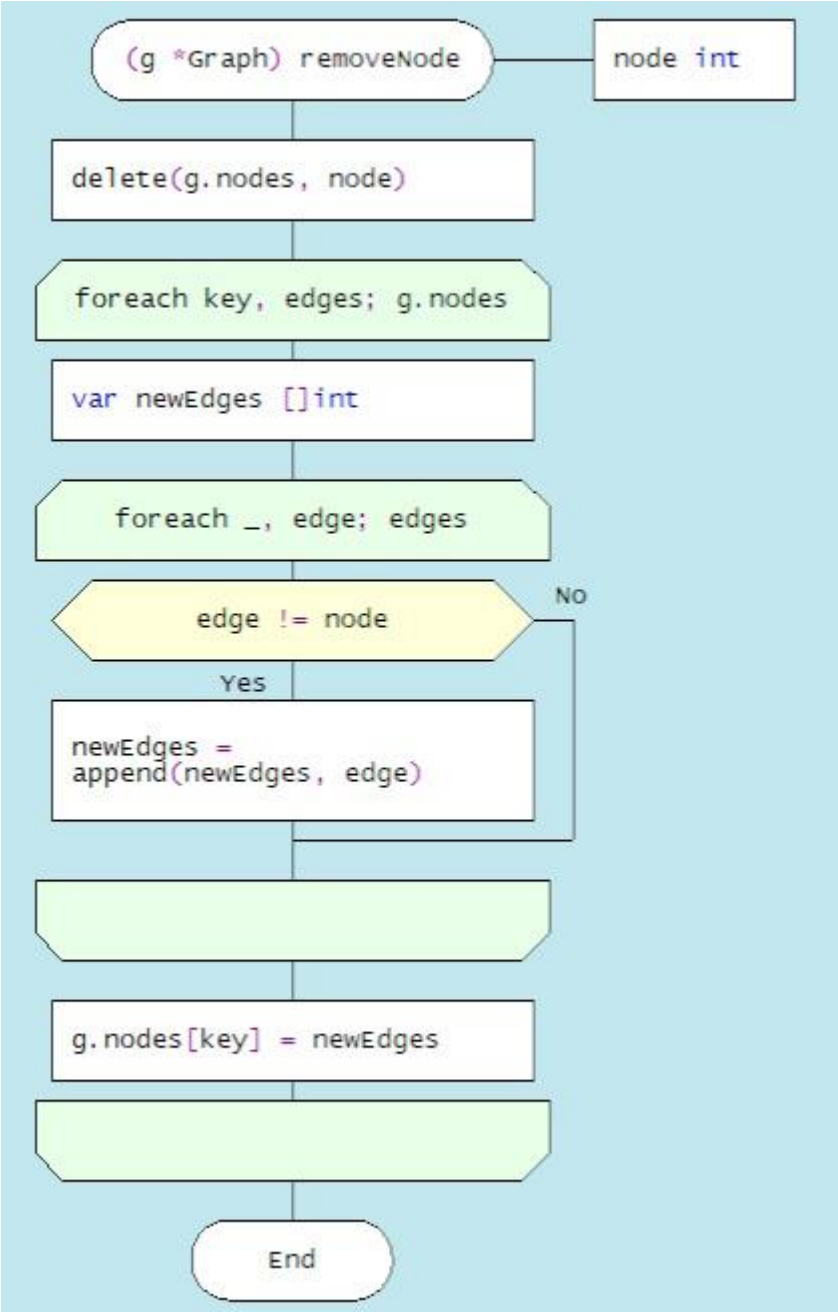
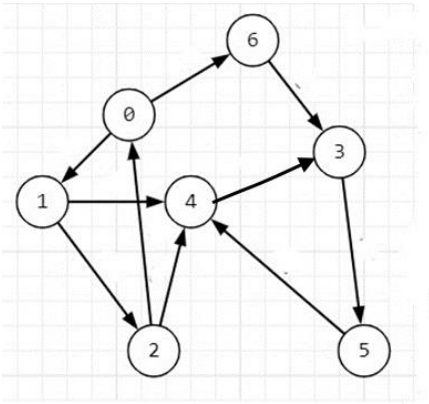


Figure 9.19. Drakon-diagram of the algorithm for removing a vertex

9.5. Choosing a path between vertices in a directed graph

The choice of a path between two vertices in a directed graph, which is of practical importance in the sense of finding the optimal path between two settlements connected by one-way roads, can be carried out according to various criteria. Let's set the task as follows: in order to determine the optimal path, find the distances of all possible paths between two geographical points, as well as the total cost of travel, taking into account the different costs per 1 km on different sections of the road.

To solve this problem, it is necessary to create a weighted graph, more specifically, to load the edges between all vertices with the values of the distances between the points and the fare per 1 kilometre (Figure 9.20).



Distance, km	Price per 1 km
["0-1"] = 10.45	["0-1"] = 1.50
["0-6"] = 13.57	["0-6"] = 1.80
["1-2"] = 18.83	["1-2"] = 2.00
["1-4"] = 14.23	["1-4"] = 3.00
["2-0"] = 43.67	["2-0"] = 1.20
["2-4"] = 27.93	["2-4"] = 1.30
["3-5"] = 47.33	["3-5"] = 1.00
["4-3"] = 29.27	["4-3"] = 3.30
["5-4"] = 57.69	["5-4"] = 1.00
["6-3"] = 33.31	["6-3"] = 1.60

Figure 9.20. Weighted directed graph with source data

Creating a new instance of the graph (the *newGraph* method), adding edge information (*the addEdge* method), and calling *the findPaths* method (the allPaths *method*) is shown in Figure 9.21.

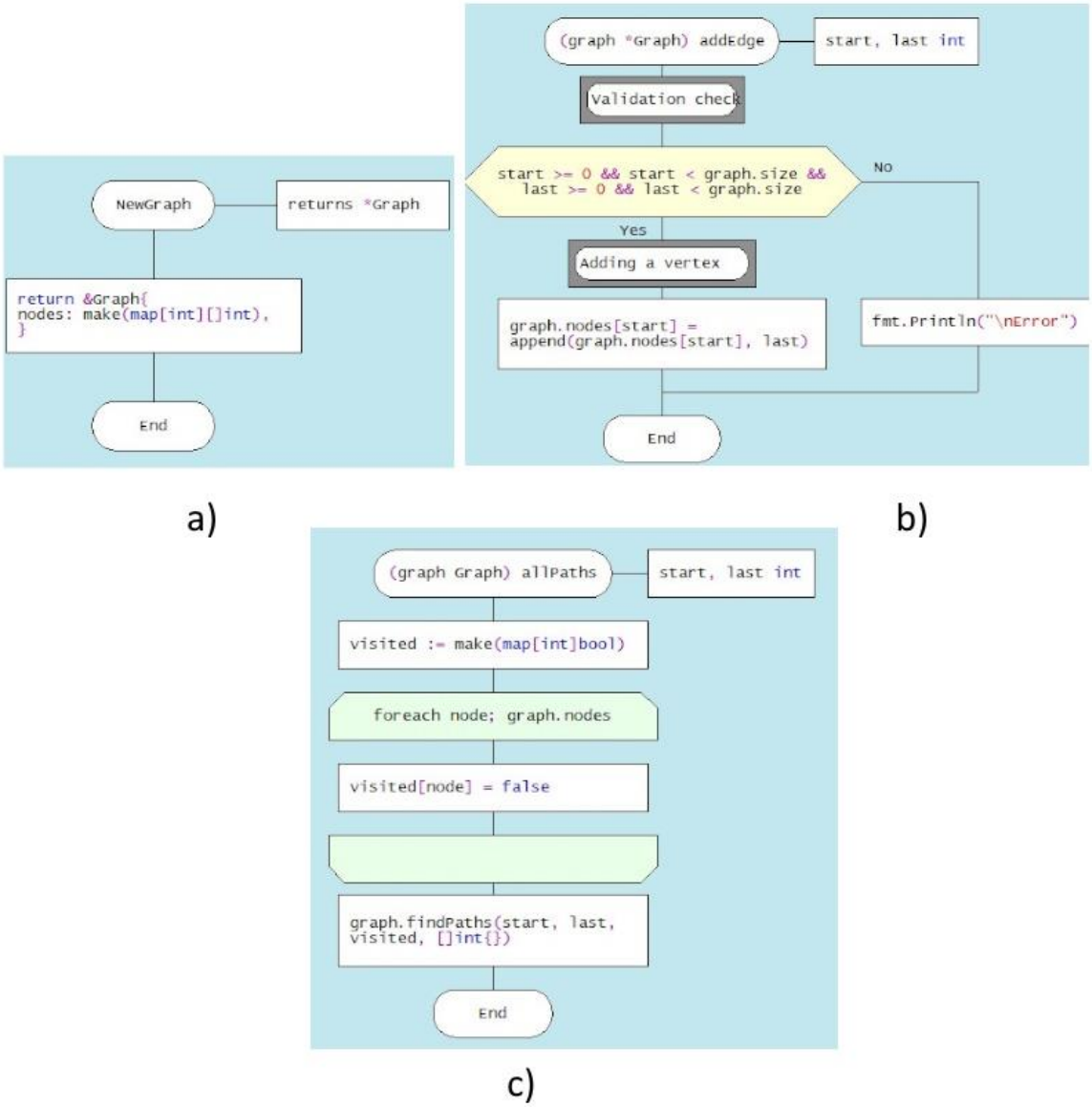


Figure 9.21. Drakon-diagrams of methods

a). ***newGaph, b). addEdge, c). allPaths***

The algorithm for finding all possible paths between two vertices is based on the sequential analysis of the list of vertex connections and recursive access to the ***findPath*** module, which implements a traversal of nodes with fixing a visit to each vertex. The DRAKON-diagram for the ***findPath*** module is shown in Figure 9.21. The algorithm consists of three parts: in the first part, the correctness of the module parameters (the names of the initial and final vertices) is set, and the fact of visiting the current vertex is checked. The second part traverses through the nodes of the graph that have not yet been visited, forming a string consisting of visited vertex names and a separator between them, for example, “2-0-1-4-3-5” (Figure 9.22). In the

third part, when the final vertex is reached, that is, when the condition ***“start == last”*** is met, the minimum path between these vertices is determined. In addition, a possible path string, for example, "2-0-1-4-3-5", is divided into sections of the type "2-0" in order to form keys for maps containing the distances between the corresponding vertices and the cost of travelling 1 km, these areas.

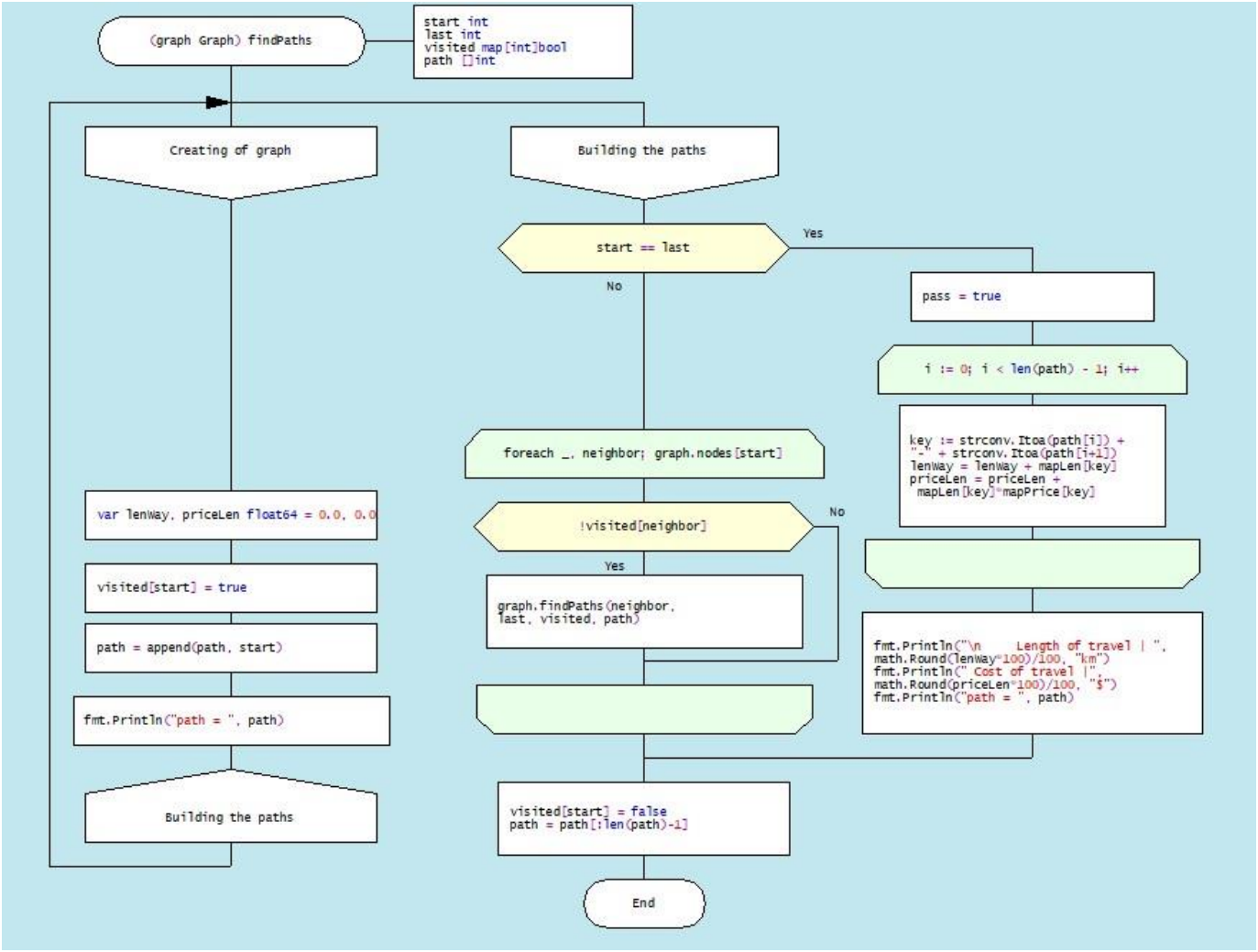
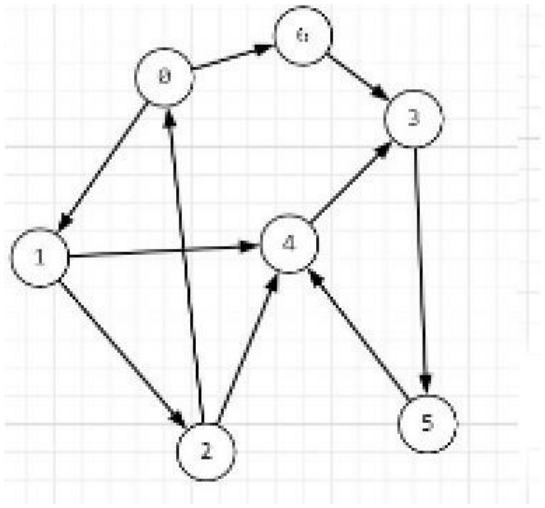


Figure 9.22. DRAKON-diagram of method ***findPath***

Consider the problem of finding all possible paths between two vertices of the graph (2) and (5) shown in Figure 9.23.:



The adjacency list

```
0: 1, 6
1: 2, 4
2: 0, 4
3: 5
4: 3
5: 4
6: 3
```

Figure 9.23. Graph and the adjacency list

All the paths of the directed graph between vertices (2) and (5) are shown in Figure 9.24.

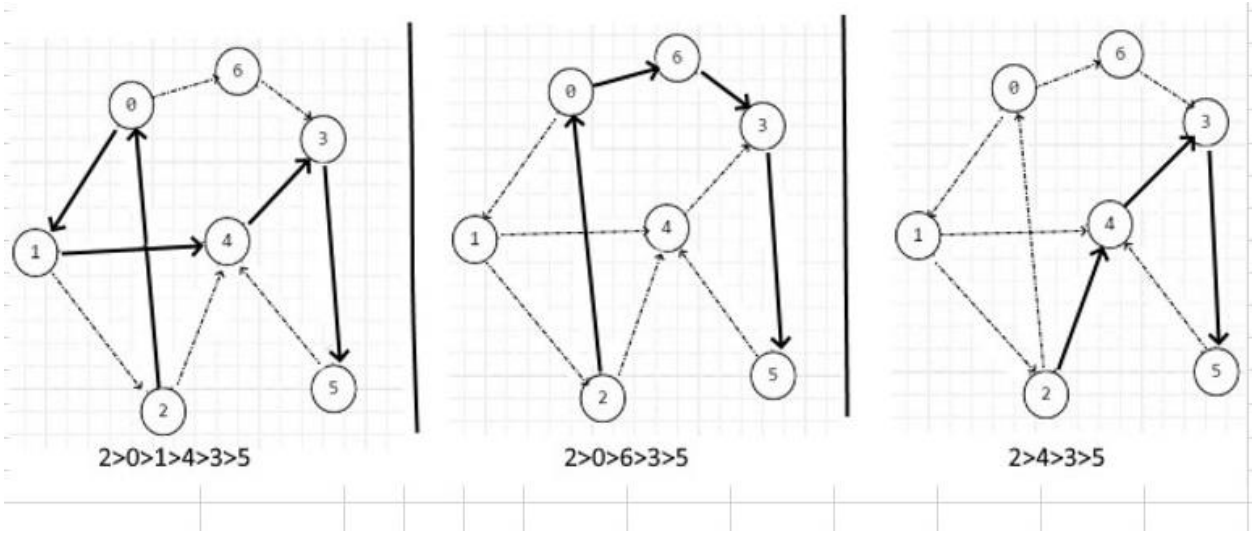


Figure 9.24 Possible paths to traversal of vertices of a directed graph

The results of passing through the vertices in algorithm implementation process is shown in Table 9.2.

Table.9.2. Implementation process ***findPath*** (first traversal)

Vertices	Neighbors	Path	Visited vertices
start ≠ last			
2	(2, 4)	2	TRUE
0	(1, 6)	2 0	TRUE
1	(2,4)	2 0 1	TRUE
4	(3)	2 0 1 4	TRUE
3	(5)	2 0 1 4 3	TRUE
5	(4)	2 0 1 4 3 5	TRUE

start = last → Recursion → (path = path[:len(path) – 1]		
Vertices	Path	Visited vertices (true); Unvisited vertices (false)
5	2 0 1 4 3 5	0:true 1:true 2:true 3:true 4:true 5:true 6:false
3	2 0 1 4 3	0:true 1:true 2:true 3:true 4:true 5:false 6:false
4	2 0 1 4	0:true 1:true 2:true 3:false 4:true 5:false 6:false
1	2 0 1	0:true 1:true 2:true 3:false 4:false 5:false 6:false
0	2 0	0:true 1:false 2:true 3:false 4:false 5:false 6:false

The results of the comparison of fares from the top (2) and (5) are shown in Table 9.3.

Table 9.3. Distance and fare

Travel	Length of travel, km	Cost of travel, $
2-0-1-4-3-5	144.95	254.69
2-0-6-3-5	137.88	177.46
2-4-3-5	104.53	180.23

Table 9.3 shows that the second trip (2-0-6-3-5) is the most cost-effective.

9.6. Dijkstra's algorithm

The Dijkstra algorithm is designed to find the shortest paths from a given vertex to all other vertices, provided that the edges of the graph are not negative. The algorithm was named after the Dutch scientist E. W. Dijkstra in 1956. This algorithm is widely used in various fields of science, technology and social life. In GPS systems, an algorithm is used to find the fastest route to a destination. In telecommunications networks, it is used to determine the optimal path for data to be transmitted from source to receiver. In robotics, an algorithm can be used to plan a robot's path to reach a goal in the most efficient way. In computer games, to determine the path of characters or objects. In network planning, an algorithm can be used to find the optimal delivery path for goods.

To implement Dijkstra's algorithm, vertex structures *of type* ***Vertex*** are created to describe vertices and ***Graph*** to describe the graph (Figure 9.25)

Edit file description

```
=== header ===
package main

import (
        "fmt"
        "math"
)

type Graph struct {
        vertices []*Vertex
}

type Vertex struct {
        key      int
        adjacent []*Vertex
        lengths  []float64
        dist     float64
}
=== footer ===
```

Figure 9.25 Description of vertex and graph structures

Here, ***key*** is the key of the vertex of the graph; ***adjacent*** – adjacency list for the vertex; ***lengths*** - a set of edge lengths to vertices from the adjacency list; ***dist*** *is* the total length of the path.

Next, an instance of the graph is created in the main program and vertices are instantiated in the loop (Figure 9.26):

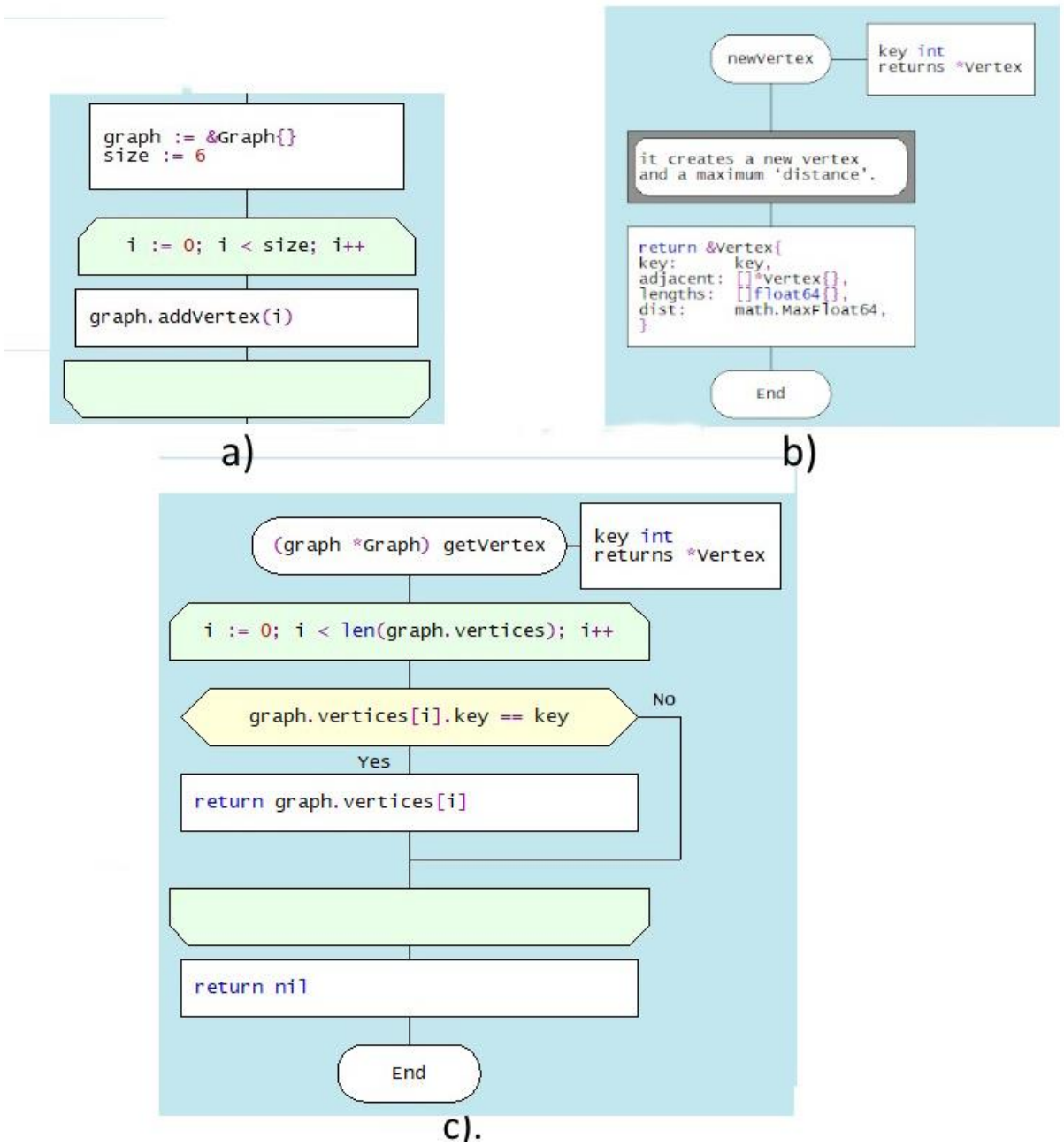


Figure 9.2**6**. Drakon-diagrams of graph vertex formation methods

a). instantiating the graph and vertex templates

b). creating a New Vertex

c). populating the vertex with data

After creating the vertices of the graph and populating them with data (keys, neighboring vertices, and distances), edges are formed using the ***addEdge*** method (Figure 9.27):

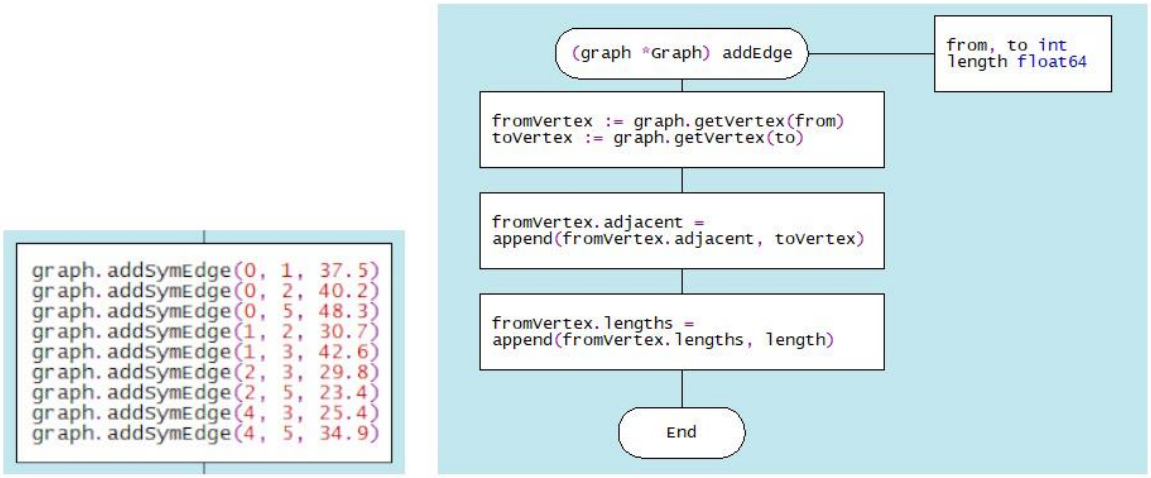


Figure 9.27. Drakon-diagrams of the *addEdge* graph edge formation method

The algorithm of the ***addEdge(from, to, length)*** method is implemented as follows. First, the method gets two vertices, between which you need to add an edge (***graph.getVertex(from), graph.getVertex(to)***), where ***from*** and ***to*** are vertex keys. The method then adds the ***toVertex*** vertex *to* the list of adjacent vertices for the vertex ***fromVertex*** (***append(fromVertex.adjacent, toVertex***)). As a result, an edge is formed between these two vertices ***fromVertex*** and ***toVertex***. In the last step, the method adds the edge length to the list of edge lengths for the *vertex* ***fromVertex*** (***append(fromVertex.lengths, length)***). This means that the length of the edge between the vertices ***fromVertex*** and ***toVertex*** is now known.

Note that the main program uses the ***addSymEdge (u,v,L)*** method, which creates symmetric data for each edge in the undirected graph (Figure 9.28):

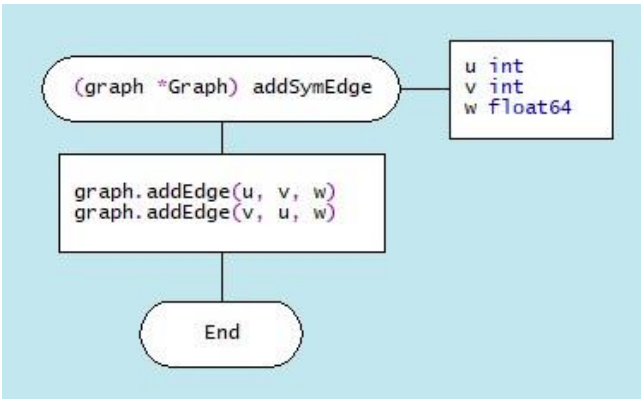


Figure 9.28. Drakon-diagram of the ***addSymEdge*** edge symmetrization method

Next, we call the Dijkstra method (startKey int), where startKey int is the initial vertex, and output the final results (Figure 9.29):

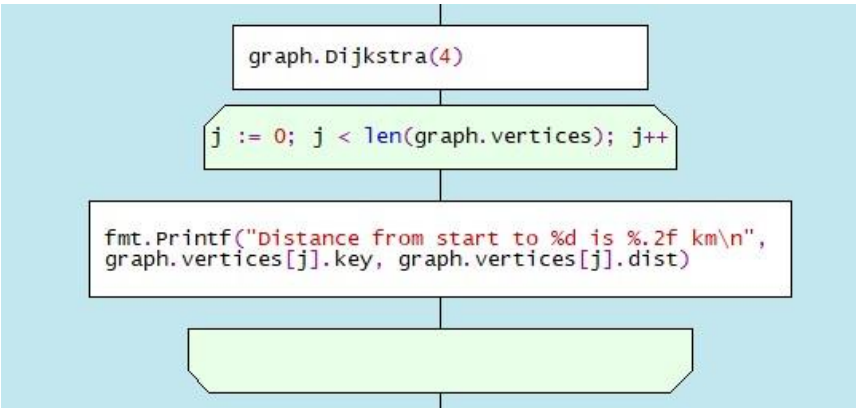


Figure 29. Drakon-diagram of the Deikstra method and output of the results

The dragon diagram of the ***Dijkstra(startkey)*** method is shown in Figure 9.30.

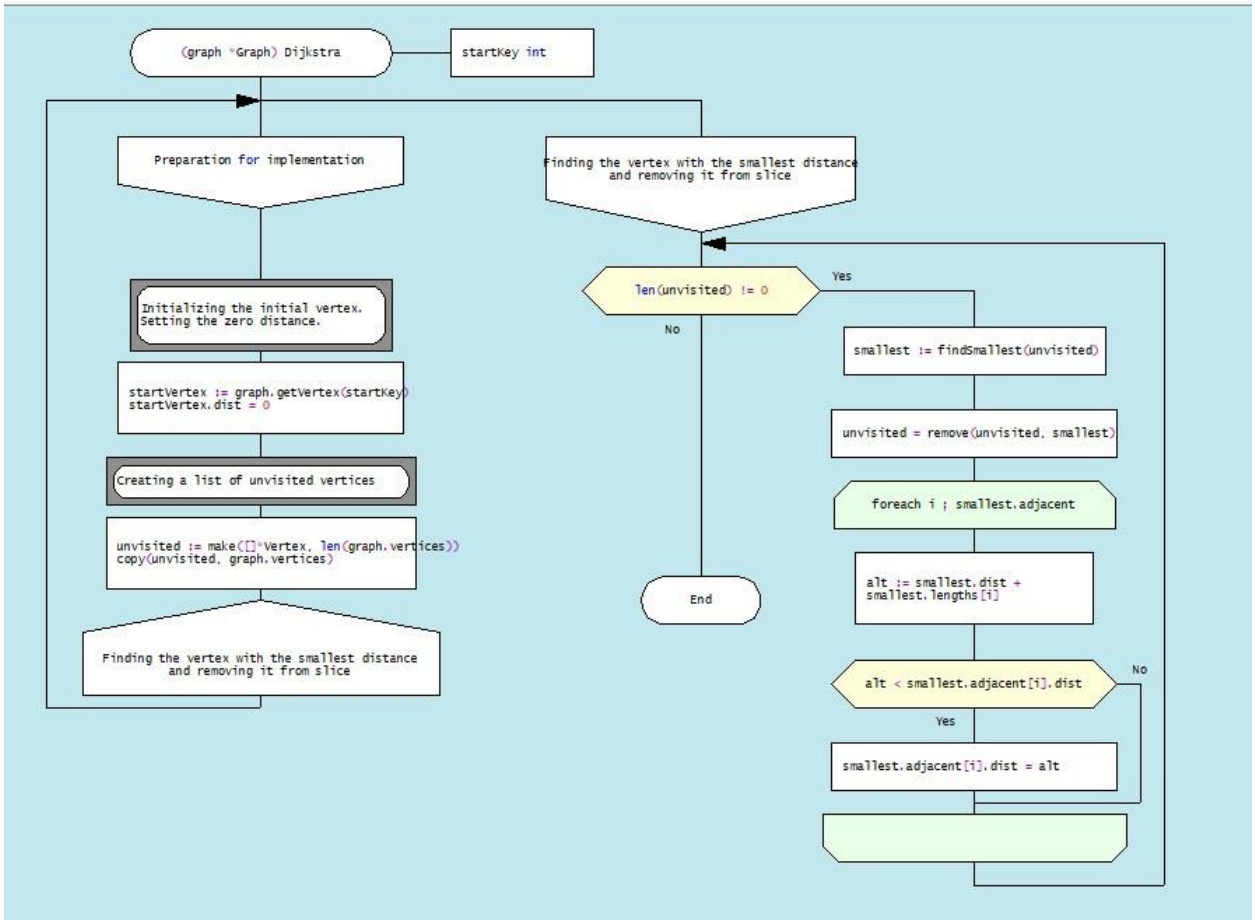


Figure 9.30 Drakon-diagram of method ***Dijkstra***

Let's take a look at the basic steps of implementing this algorithm. First, initialization (***startVertex := graph.getVertex(startKey)***; ***startVertex.dist = 0***) takes place, which results in getting the initial vertex from the key and setting its

distance to 0. In the second step, a list of unvisited vertices is created, which is then copied with the copy command in order to be able to manipulate this list during the processing of vertex data without changing the graph itself.

Next, a loop ***for len(unvisited) != 0*** is performed to traverse all unvisited vertices in order to find the vertex with the shortest distance using the ***method smallest := findSmallest(unvisited)*** (Figure 9. 31.)

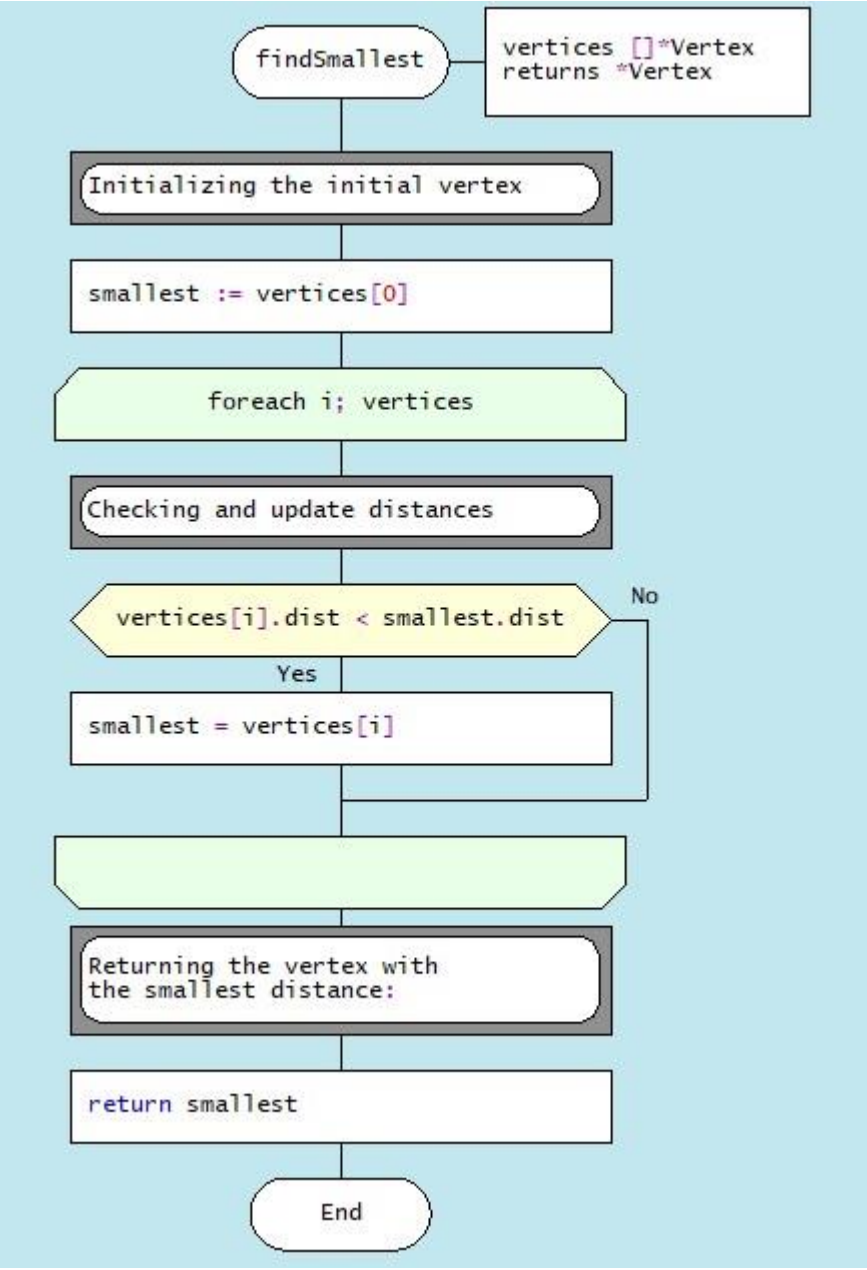
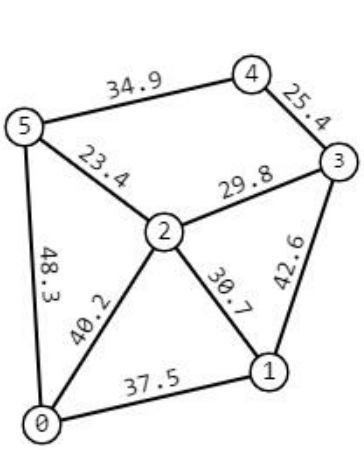


Figure 9.31. Drakon-diagram of mrthod ***findSmallest(unvisited)***

The current vertex is then removed from the list of unvisited vertices (***unvisited = remove(unvisited, smallest)***). Finally, in a loop across all vertices, the

distances to adjacent vertices are updated as a result of the shorter path being detected.

As an example, consider the application of Dijkstra's algorithm to finding the shortest paths from a given vertex for the undirected graph shown in Figure 9.32.



Adjacency list

```
0: 1, 2, 5
1: 0, 2, 3
2: 0, 1, 3, 5
3: 1, 2, 4
4: 3, 5
5: 0, 2, 4
```

Figure 9.32. Undirected loaded graph

The results of calculations of the shortest paths from vertex 0 to the rest of the vertices of the graph (Figure 9.33.):

Distance from start to 0 is 0.00 km

Distance from start to 1 is 37.50 km

Distance from start to 2 is 40.20 km

Distance from start to 3 is 70.00 km

Distance from start to 4 is 83.20 km

Distance from start to 5 is 48.30 km

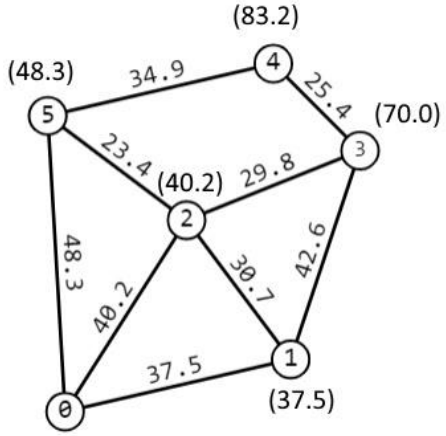


Figure 9.33. Shortest distances from vertex 0 are given in parentheses

And, finally, about estimating the complexity of Dijkstra's algorithm. This estimate can vary depending on the data structure used to represent the graph. In the case of a ***Linked List***, the time complexity of Dijkstra's algorithm is estimated to be O(V^2), where V is the number of vertices in the graph. This is because you have to look at all vertices each time you retrieve a minimal item from the priority queue.

When using the ***map*** *data structure,* the time complexity of Dijkstra's algorithm is *O((V+E) log V),* where E is the number of edges in the graph. This is because it takes O(log V) time to retrieve the minimum element from the priority queue implemented with map.

In both cases, the spatial complexity will be *O(V + E)* because you need to store all the vertices and edges of the graph. However, it's worth noting that ***a map*** usually takes up more space than a ***linked list*** because of the additional information it stores (keys and values).