**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 [https://www.geeksforgeeks.org/memory-access-methods/]. *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. This section covers the Go-implementation of linear abstract data such as array/slice, map, linked lists, stacks, and queues.

Next, let's look at how data structures are implemented in the Golang programming language.

**4.2. Linear data 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 and by the homogeneity of the data (homogeneous and heterogeneous).

4.2.1 Linear direct access structures

The basic linear direct access structures are homogeneous (arrays and slices) and heterogeneous (linked lists, stacks, queues). 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.).

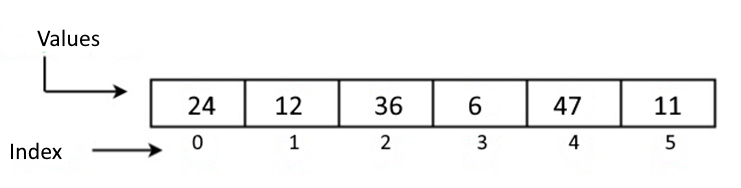


Figure 4.1. One-dimensional array

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)

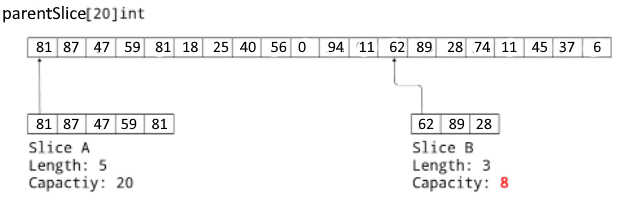


Figure 4.2. Create different slices of SliceA and SliceB

*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 sliceB := 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.

c) Structure

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 Smith49 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)

}

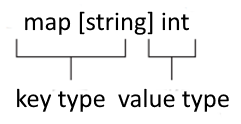
Результат.

First Name: Sam

Age: 55

d). 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["П. Петров"] = 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{

  "П. Wolf": 45,

    "В. Smith": 47,

}

**Addition of a new entry:**

employeeAge["К. Tompson"] = 34

**Output to the monitor:**

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

Other operations with the map.

**To get the value corresponding to the key:**

age := employeeAge["К. Маркин"]

**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.*.*

а) 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).

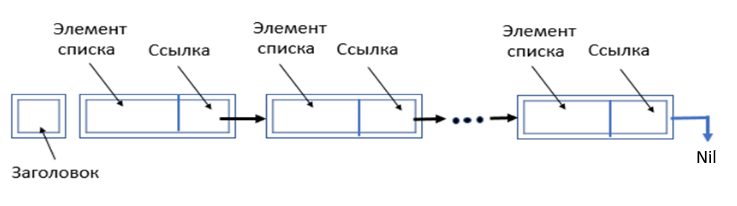


Figure 4.3. Simply linked list structure

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

}

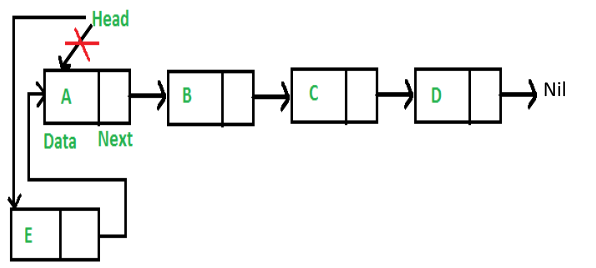
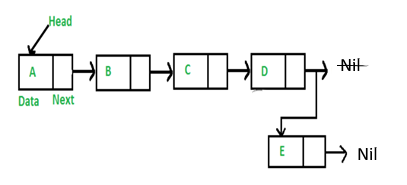
The "len" field in the linked list structure contains its length. The "headNode" field stores the memory address of the header (the first node in the linked list). Initialization (instance creation) of a structure of type LinkedList is performed as follows:

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

1. 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).

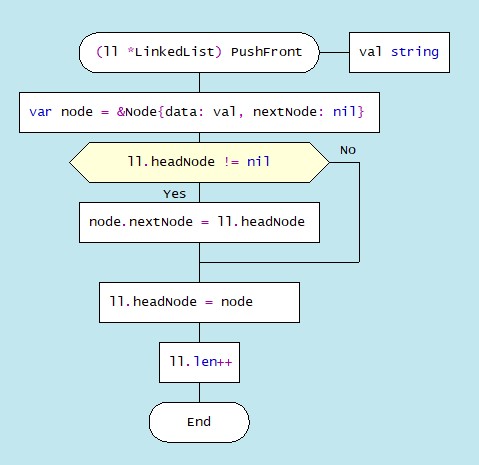


Figure 4.5. DRAKON-diagram of rhe method PushFront(val)

1. 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).

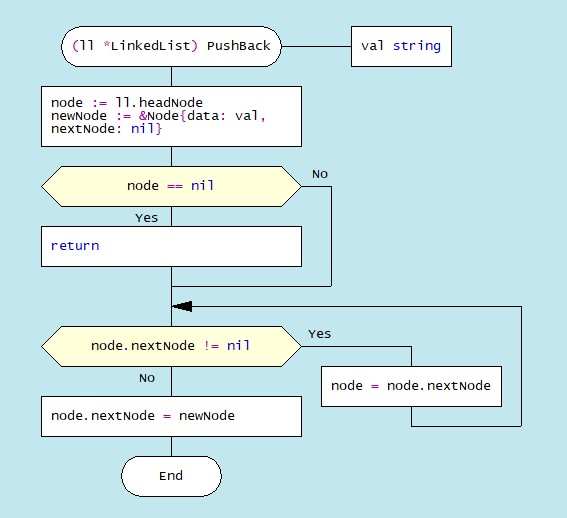


Figure 4.6. DRAKON-diagram of node insertion algorithm to the linked list end

1. 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.

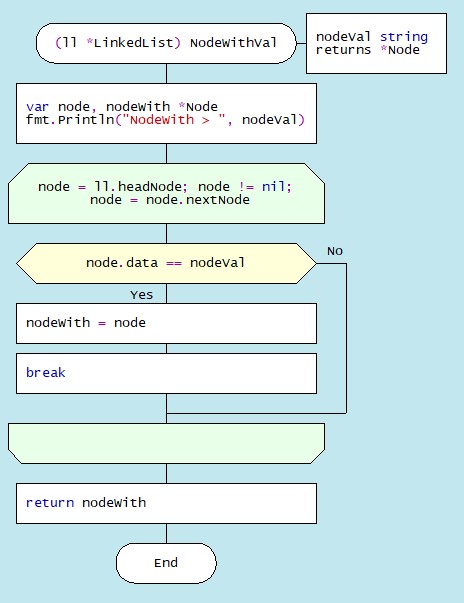
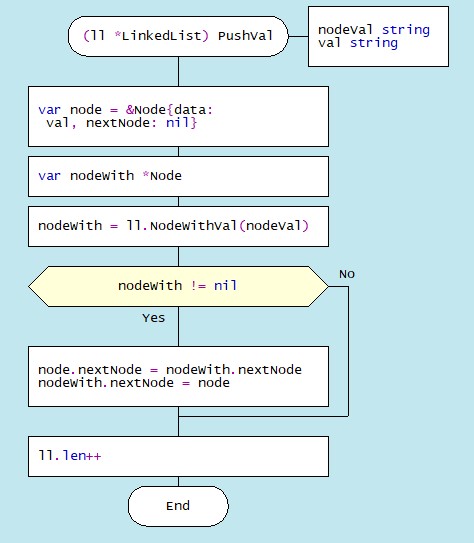
 

Figure 4.7. DRAKON-diagrams of node insertion algorithms NodeWithVal PushVal

1. 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).

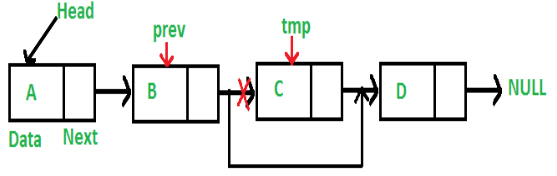


Figure 4.8. Illustration of removing a node from a list

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

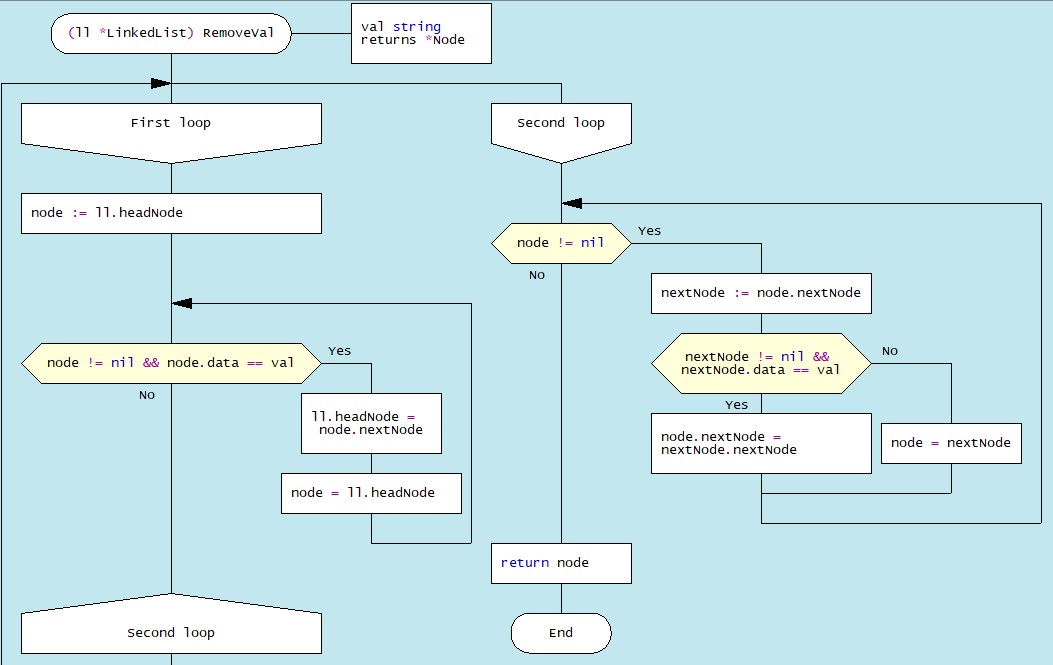


Figure 4.9. Drakon-diagram of removing a node from the RemoveVal list

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, .

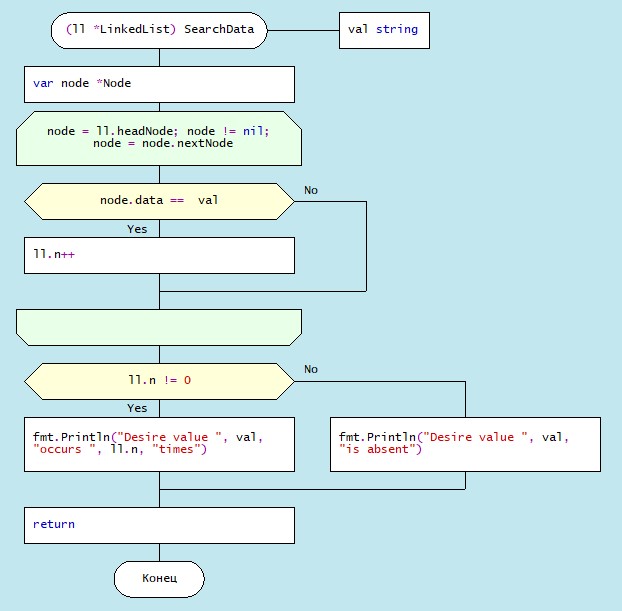


Figure 4.10. Drakon-diagram of the SearchVal() method

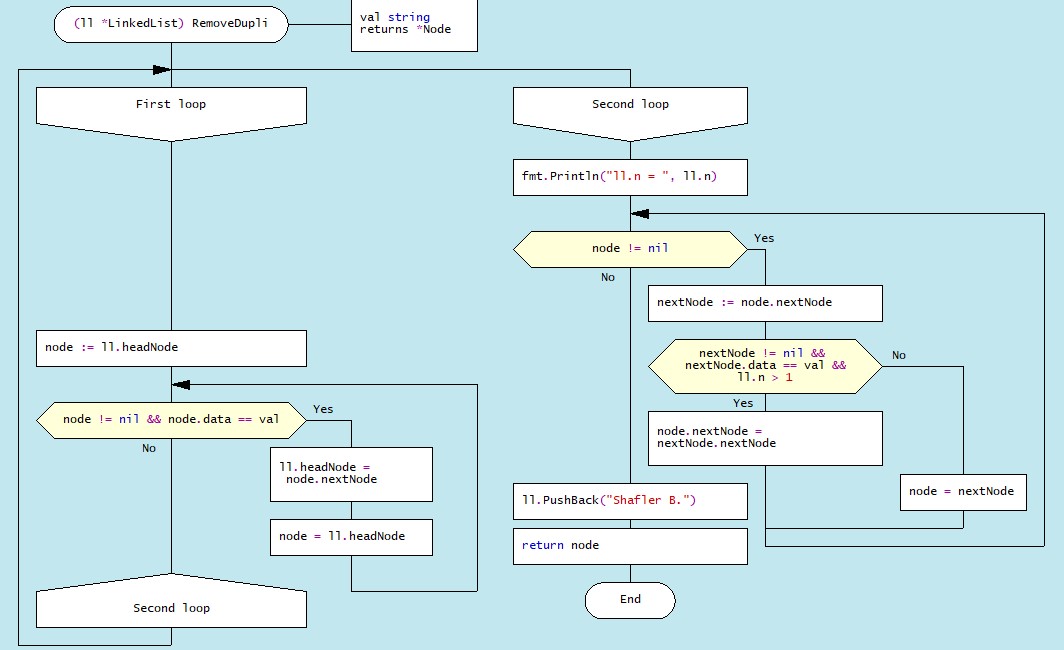


Figure 4.11. Drakon-diagram of the RemoveDupli() method

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.

**b). 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.):

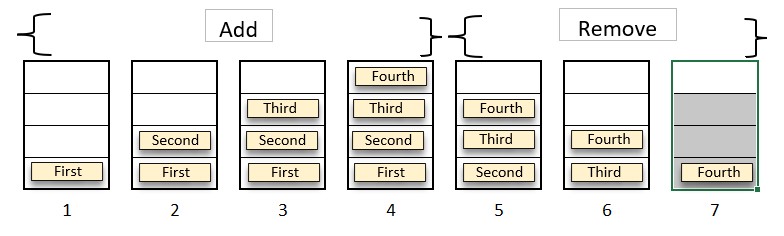


Figure.4.12. Last-in, first-out stack work

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 headstructmember 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())

**c). 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).

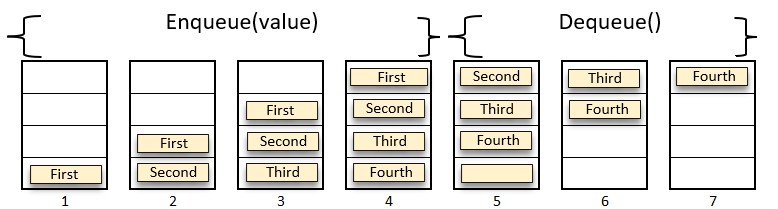


Figure 4.12. The work of the "queue" structure

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 linked 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

}