**РАЗДЕЛ 4 РЕАЛИЗАЦИЯ АБСТРАКТНЫХ ТИПОВ ДАННЫХ В GOLANG**

**4.1. Признаки структурированности данных**

Напомним, - абстрактный тип данных представляет собой математическую модель данных и различные опера­торы, определенные в рамках этой модели. Алгоритмы обработки данных могут быть разработаны в терминах АТД, но для их реализации в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов (методов), поддерживаемых данным языком программирования.

В практической плоскости структура данных - это данные, структурированные таким образом, чтобы обеспечить их эффективное использование пользователями, что предопределяет необходимость определенного упорядочения данных, в первую очередь, на уровне хранения в компьютерной памяти. Уменьшение пространства и повышение уровня его организованности в совокупности с уменьшением временнóй сложности различных задач является основной целью работы со структурами данных.

Значительный опыт развития компьютерной техники и вычислительных технологий позволил классифицировать структуры данных по различным признакам. Во-первых, **по признаку сложности** различают *простые*структуры и *интегрированные*. Критерием простоты является неделимость данного, то есть в компьютерной реализации – простая цепочка битов. К простым, базовым структурам относятся переменные различных типов: целые, вещественные числа, логические, строковые.

Интегрированные (композитные, сложные) – это структуры данных, составными частями которых являются другие структуры данных, включая простые и интегрированные. Многие базовые интегрированные структуры данных предопределены конкретным языком программирования: массивы, срезы, структуры и т.д. Такие структуры могут создаваться пользователями под конкретную проблемную задачу, используя базовые интегрированные структуры.

По **способу представления** структуры данных делятся на логическую и физическую. *Логическая структура*- это абстрактная схема расположения данных, которую представляет себе пользователь или программист. *Физическая структура* – это способ (схема) конкретного размещения данных в памяти вычислительной машины. В общем случае логическая и физическая структуры одних и тех же данных не совпадают. В логической (абстрактной) структуре данные как правило располагаются смежно по отношению друг к другу, в то время как в компьютерной реализации эти данные могут располагаться на различных участках памяти.

Важным признаком структуры данных является **наличие связей между элементами** структуры. По этому признаку различают *несвязные и связные.*  
Несвязные структуры характеризуются отсутствием связей между элементами структуры, в то время как связные структуры характеризуются наличием связи. К несвязным структурам относятся массивы, строки, стеки, очереди; к связным - связные списки.

Во многих случаях при работе с данными может играть роль такая их характеристика как ***изменчивость****,* то естьизменение числа элементов и (или) связей между элементами структуры. По этому признаку различают *статические, полустатические и динамические* структуры.  
К статическим относят массивы, множества, записи, таблицы,  
к полустатическим  - стеки, очереди, деревья, к динамическим - линейные и разветвленные связные списки, графы, дерева.

***По характеру упорядоченности*** элементов в структуре различают  *линейные и нелинейные* структуры данных. Линейные структуры в зависимости от характера взаимного расположения элементов в памяти разделяют на структуры с последовательным распределением элементов в памяти  (векторы, строки, массивы, стеки, очереди) и структуры с произвольным связным распределением элементов в памяти (односвязные и двусвязные линейные списки). Нелинейные структуры - многосвязные списки, дерева, графы.

Одним из определяющих признаков структур данных является ***способы доступа к данным***. В методе доступа важен механизм поиска – алгоритм, определяющий путь доступа, который возможен в рамках заданной структуры памяти, и количество шагов вдоль этого пути для нахождения искомых данных. Среди методов доступа к данным выделяют две основные группы: *последовательный* и *прямой* [https://www.geeksforgeeks.org/memory-access-methods/].

*Последовательный доступ* означает, что доступ к группе элементов осуществляется в заранее определенной упорядоченной последовательности. Примером последовательного доступа является односвязный список. *Прямой доступ* к различным элементам структуры данных осуществляется путем указания уникального адреса этих элементов.

Наконец следует отметить такой признак структур данных как *однородность*. К однородным относятся структуры, содержащие множество простых данных одного типа (числовые, логические, строковые и т.д.). Неоднородные структуры объединяют данные различных типов. Примерами однородных структур являются массивы, срезы, стеки. К неоднородным структурам относятся записи и множества.

Далее рассмотрим как реализуются структуры данных в языке программирования Golang.

* 1. **Линейные структуры данных**

Линейные структуре данных - это структуры, в которых элементы данных расположены в последовательном порядке. Линейные структуры можно различать по способам доступа к отдельным элементам той или иной коллекции данных и по признаку однородности данных (однородные и неоднородные).

* + 1. Линейные структуры прямого доступа

К базовым линейным структурам прямого доступа относятся однородные (массивы и срезы) и неоднородные (записи, словари, хэш-таблицы).

а) Массив (Array)

Массив представляет собой коллекцию данных, принадлежащих к одному типу. Например, коллекция целых чисел 24, 12, 36, 6, 47, 11 образует массив (рис. 4.1.).

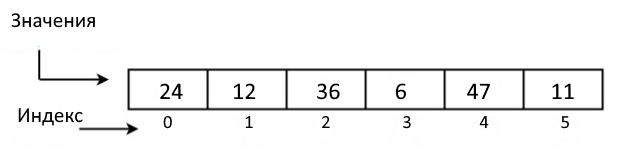


Рис.4.1. Одномерный массив

Смешивание значений разных типов, например массива, содержащего как символы, так и целые числа, в Go не допускается. Существуют различные способы объявления массивов:

var array\_name[]Type

или

var array\_name[length]Typle{item1, item2, item3, ...itemN}

Кроме того, в языке Go массивы могут быть объявлены в сокращенном виде:

array\_name:= [length]Type{item1, item2, item3,...itemN}

В языке Go можно создать многомерный массив, используя следующий синтаксис:

array\_name[Length1][Length2]..[LengthN]Type

Однако данные в виде массива в Go используются достаточно редко. Намного более удобна такая коллекция данных как *срез*.

**б) Срез (Slice)**

Срез это коллекция данных переменной длины, в которой хранятся элементы однородного типа. Срез можно рассматривать как фрагмент массива.

Синтаксис среза (T – тип) данных:

[]T

или

[]T{}

или

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

Срез состоит из трех компонентов: *указателя, длины и емкости*. Для создания среза в таком виде используется функция make. Для примера на рис.4.2. создание среза parentSlice выглядит так:

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

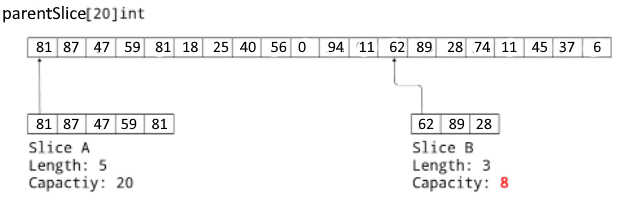


Рис.4.2. Создание разных срезов SliceA и SliceB

*Указатель* (*Pointer*) указывает на первый элемент массива, доступный через срез (который не обязательно совпадает с первым элементом массива). *Длина* (*Length*)— это количество элементов среза; она не может превышать *емкость (capacity)*, которая, как правило, представляет собой количество элементов между началом среза и концом базового массива. Эти значения возвращаются встроенными функциями 1еп и cap. На базе одного массива можно создать несколько срезов с разными значениями указателя, длины и емкости. На рис. 4.2. показан первичный срез/масив parentSlice, на базе которого создаются два среза разной длины, начинающиеся с разных мест. Для этого используется запись sliceA := parentSlice[:5] и sliceB := parentSlice[12:15]. Здесь емкость среза SliceA равна 20, а sliceB равна 8, поскольку Go определяет это значение как разность между длиной первичного массива (20) и индексом первого элемента среза Slice B (12).

Для работы со срезами используются такие функции:

1. Встроенная функция append() используется для добавления элементов в срез. Если размера базового среза недостаточно, то автоматически создается новый срез и в него копируется содержимое старого среза.

2. Функция len() возвращает количество элементов, присутствующих в срезе.

3. Функция cap() возвращает емкость базового среза.

4. Функция copy(), содержимое исходного слайса копируется в срез назначения.

**в) Структура (Structure)**

Golang поддерживает коллекции данных в виде структур, состоящих из набора множественных типов данных (полей), представляемых как единая сущность. В Golang структура реализуется c помощью типа данных Struct:

package main

import (

"fmt"

)

type Employee struct {

firstName string

lastName string

age int

salary int

}

func main() {

//создание структуры с указанием полей

emp1 := Employee{

firstName: "Петр", age: 35, salary: 20000, lastName: "Семенов",

}

// создание структуры без указания имен полей

emp2 := Employee{"Павел", "Кудинов", 49, 35000, }

fmt.Println("Employee 1", emp1)

fmt.Println("Employee 2", emp2)

}

Результат.

Employee 1 {Петр Семенов 35 20000}

Employee 2 {Павел Кудинов 49 35000}

Замечание. При выводе на монитор (emp1) порядок следования полей необязательно должен совпадать с порядком при создании структуры. Во втором случае создания структуры (emp2) порядок должен совпадать.

Golang позволяет создавать указатели на структуру.

package main

import (

"fmt"

)

type Employee struct {

firstName string

lastName string

age int

salary int

}

func main() {

emp3 := &Employee{

firstName: "Николай",

lastName: "Степин",

age: 55,

salary: 22000,

}

fmt.Println("First Name:", (\*emp3).firstName)

fmt.Println("Age:", (\*emp3).age)

}

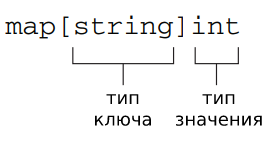
Результат.

First Name: Sam

Age: 55

**г). Карта (Map)**

Карты представляют собой коллекции, в которых хранятся неупорядоченные пары *ключ-значение*, где ключи – это уникальные идентификаторы, связные с каждым значением в карте. Карты особенно эффективны в алгоритмах поиска данных. Ключи карты могут быть практически любого типа, в отличие от массивов и срезов, где для ключей используется последовательность [целых чисел](https://golangs.org/integer). Тип для ключей и значений в Go нужно уточнять. Для объявления карты с ключами типа [string](https://golangs.org/string)  и значениями типа int используется синтаксис map[string]int:



|  |  |
| --- | --- |
| Синтаксис  (mapName -имя карты) | Смысл |
| var mapName map [KeyType] ValueType | объявить карту |
| var mapName = map [KeyType] ValueType {} | объявить и назначить пустую карту |
| var mapName = map [KeyType] ValueType {key1: val1, key2: val2} | объявить и назначить карту |
| mapName: = make (map [KeyType] ValueType) | объявлять и инициализировать карту размера по умолчанию |
| mapName: = make (map [KeyType] ValueType, length) | объявить и инициализировать карту размера *длины* |

Различные операции при работе с картами показаны на конкретном примере. Создадим карту, содержащую данные о сотрудниках и их возрасте EmployeeAge:

employeeAge := map[string]int{}

Инициализация этой карты:

employeeAge = map[string]int{

  "П. Петров": 45,

    "В. Маркин": 47,

Добавление новой записи:

employeeAge["К. Маркин"] = 34

Вывод на монитор:

map[А. Костин:34 В. Маркин:47 П. Петров:45]

Карта может быть создана с помощью функции make:

employeeAge := make(map[string]int)

employeeAge["П. Петров"] = 45

Другие операции с картой.

Обновление ключа:

Получение значения, соответствующего ключу:

age := employeeAge["К. Маркин"]

Удаление пары "ключ-значение"

delete(employeeSalary, "Tom")

Проверка существования ключа:

val, ok := mapName[key]

Если ключ существует, переменная val будет значением ключа в map, а переменная ok будет true. Если ключ не существует, переменная val получит нулевое значение по умолчанию типа value, а переменная okбудет false.

* + 1. **Линейные структуры последовательного доступа**

Последовательный доступ к данным означает, что за каждый конкретный момент времени можно обратиться лишь к одному элементу структуры, причем доступ к элементам происходит в определённом порядке. Классическими примерами структуры последовательного доступа являются *односвязный список, стек и очередь.*

**а) Cвязный список (Linked List)**

Связный список представляет собой динамическую структуру данных, элементами которых называют узлы, состоящие из двух частей: *содержательной* и *ссылочной*. Содержательная часть, предназначенная для хранения значения данных, может быть представлена одним из базовых типов данных, такими как целое число, число с плавающей запятой, строка или какого-либо другого типа данных. Ссылочная часть представляет собой ссылку, которая используется для хранения адреса следующего элемента в списке.

Связные списки находят свое применение при решении многих вычислительных задач от организации операционных систем до создания плейлистов. В частности, они полезны при обработке файловой системы: иногда трудно найти дисковое пространство для размещения всего файла, поэтому он может разбиваться на разрозненные фрагменты. Для организации работы с этими фрагментами формируется связный список участков, в которых фрагменты файла хранятся на диске. Этим они отличаются от массивов или срезов, где все элементы расположены смежно по отношению друг к другу.

Существуют различные типы связных списков. Прежде всего, они различаются по количеству полей (односвязные и двусвязные списки) и по способу связи элементов (линейные и циклические). В простейшем случае односвязного списка каждый узел (кроме последнего узла) содержит ссылку (указатель) на следующий узел того же списка. Именно в ссылочной части содержится адрес следующего узла. Ссылочная часть последнего узла содержит значение *nil* (рис.4.3).

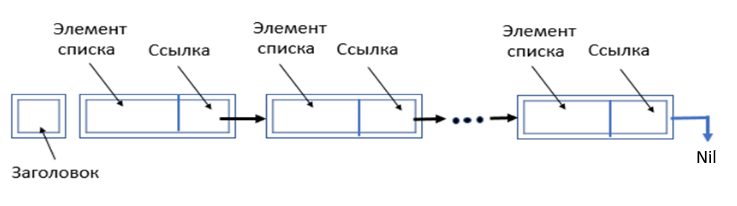


Рис. 4.3. Структура односвязного списка

Структуру отдельного узла связного списка можно описать типом данных *struct* в таком виде:

type *Node* struct {

    data string

    nextNode \*Node

}

Структура связного списка содержит длину списка, головной узел и хвостовой узел:

type LinkedList struct {

    len      int

    headNode \*Node

}

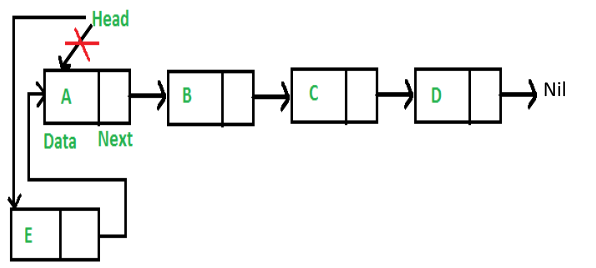
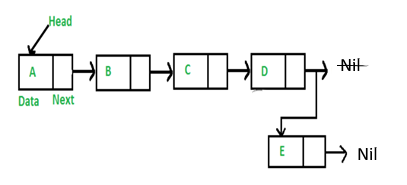
Поле «len» в структуре связного списка содержит его длину. В поле «headNode» хранится адрес памяти заголовка (первого узла связного списка). Инициализация (создание экземпляра) структуры типа LinkedList выполняется так:

var ll LinkedList = LinkedList{}

Язык Go предоставляет возможность реализовывать различные операции со связными списками, среди которых основными являются:

* + вставка элемента в список;
  + удаление элемента из списка;
  + поиск элемента в списке;

Рассмотрим операцию вставки элемента в связный список (рис.4.4):

а) б)

Рис. 4.4. Вставка нового узла в начало (а) и в конец (б) связного списка

Для вставки новых узлов используются три метода: в начало списка – PushFront(val), в конец списка – PushBack(val) и в указанную позицию PushVal(nodeVal,val), где параметр val – значение элемента списка того или иного типа (в данном случае – string), nodeVal - значение элемента списка, после которого вставляется новый узел. Рассмотрим последовательно алгоритмы всех трех методов. Удаление узлов реализует метод RemoveVal(val)

1. Метод PushFront – вставка нового узла в начало списка

Этот метод реализует процесс вставки нового узла в начало списка. Алгоритм метода представлен в виде Дракон-диаграммы c последующей автоматической генерацией программного кода в редакторе DRAKON WEB Editor и его выполнения в редакторе Visual Studio Code (рис.4.5).

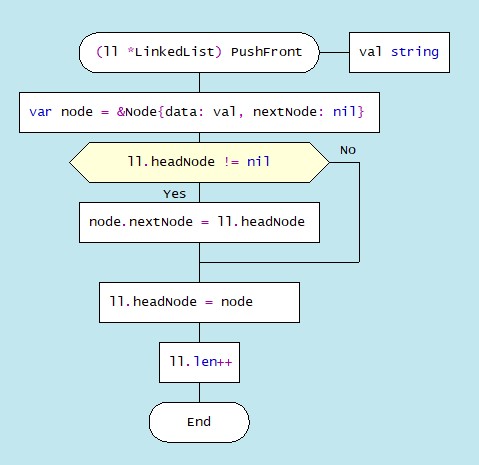


Рис.4.5. Дракон-диаграмма метода PushFront(val)

Программный код метода вставки в начало списка PushFront:

func (ll \*LinkedList) PushFront(val string) {

    // ll - экземпляр данного типа LinkedList

    var uzel = &Uzel{data: val, next: nil}

// объявляем переменную типа структура Uzel

    if ll.headUzel != nil {

        uzel.next = ll.headUzel

    }

    ll.headUzel = uzel // включение нового узла в список

 // новому начальному узлу присваивается адрес следующего узла

    ll.len++ // прирост длины списка

}

**Метод PushBack – вставка нового узла в конец списка**

Вставка нового узла в конец односвязного списка реализуется методом PushBack(val). Алгоритм вставки нового узла в конец списка заключается в следующих шагах. Создается новый узел (newNode), которому передается параметр (val). Если список пуст, то новый узел будет являться одновременно и первым, и последним в списке.  Если список не пуст, то перебираются все узлы до конца списка и новый узел добавляется в конец списка. Дракон-диаграмма метода PushBack(val) (рис. 4.5).

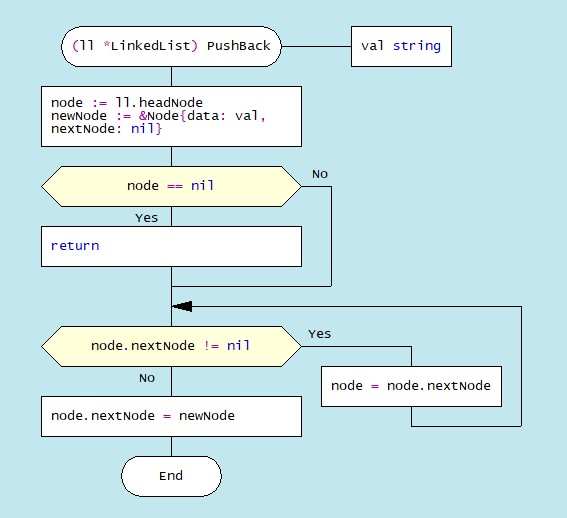


Рис.3.5. Дракон-диаграмма алгоритма вставки в конец списка PushBack(val)

func (ll \*LinkedList) PushBack(val string) {

    uzel := ll.headUzel

    newUzel := &Uzel{data: val, next: nil}

    if uzel == nil {

        return

    }

    for uzel.next != nil { // если вставка после единственного узла, то ==nil

        uzel = uzel.next

    }

    uzel.next = newUzel

    fmt.Println("PushBack >  uzel.next = newUzel >>", newUzel, uzel)

}

**Метод PushVal(nodedata,val**) - **вставка после заданного узла**

Третий метод PushVal(nodeVal,val) реализует процесс вставки нового узла с параметром val после узла со значением nodeVal. Алгоритм метода вначале обращается к модулю NodeWithNode(nodeVal), определяющего узел, после которого нужно вставить новый узел. В нашем случае вставляется слово «наверное» в созданный выше список после узла с содержимым «Это». Дракон-диаграмма алгоритма такой вставки и соответствующий программный код показаны на рис. 4.7.

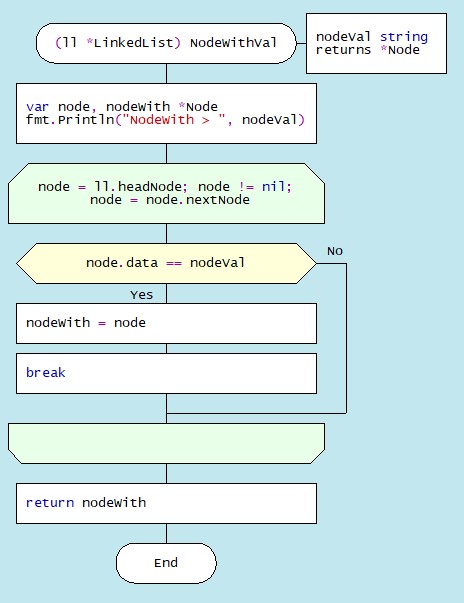
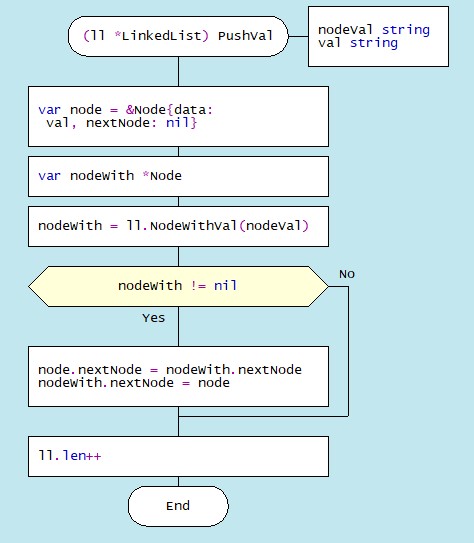
 

Рис. 4.7. Дракон-диаграммы алгоритмов вставки узла NodeWithVal PushVal

К основным методам работы с узлами связного списка относится методы удаления одного или несколько узлов либо удаление узлов по условию. Рассмотрим алгоритм удаления узла по заданному значению.

**Метод RemoveKey (val) – удаление заданного узла**

Удаление узла из связного списка после заданного значения (k) выполняется с помощью метода RemoveVal(val), параметром которого является ключ val (поле data структуры узла списка Node). Для удаления узла списка с ключом val необходимо вначале найти этот узел. В этом модуле вначале проверяется первый узел (ll.head.data == key) и если ключ совпадает со значением поля ll.head.data, то адрес первого узла заменяется адресом следующего узла, который становится главным (l.head.next). Далее в цикле происходит поиск узла с искомым ключом и сдвиг узлов (рис.4.8).

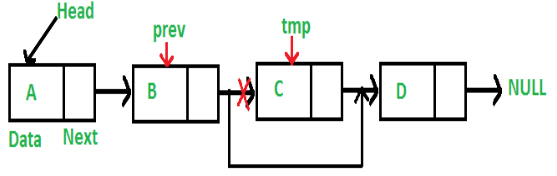


Рис. 4.8. Иллюстрация удаления узла из списка

Дракон-диаграмма алгоритма удаления узла по значению показана на рис. 4.9.

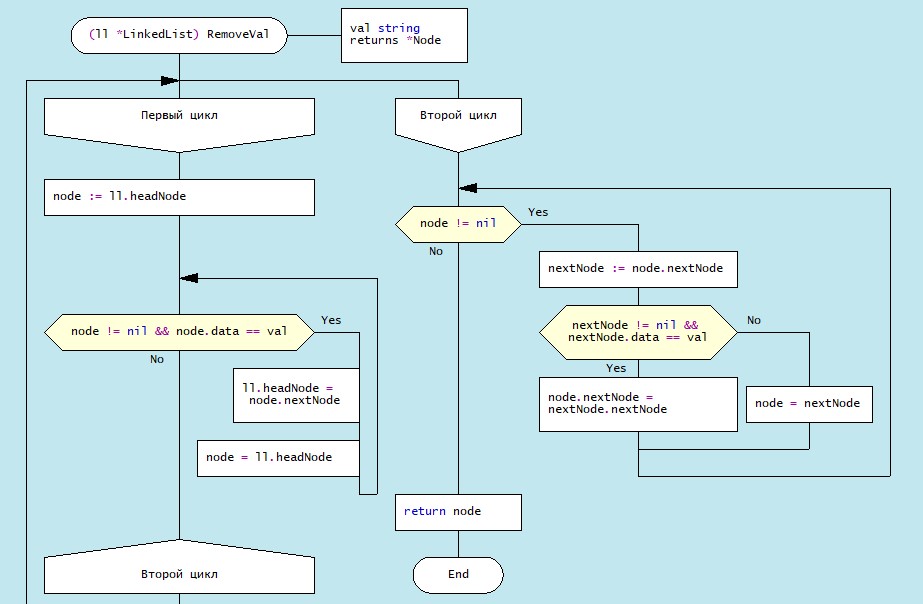


Рис. 4.9. Дракон-диаграмма удаления узла из списка RemoveVal

В качестве примера создадим список, состоящий из трех узлов, содержащих три значения: “Это”, “мой”, “список”, Для формирования списка необходимо создать тип Node{data,nextNode), где data – значение, а nextNode – адрес следующего узла и тип LinkedList(len, headNode), где len – длина списка, а headNode - заголовок списка с типом \*Node:

Далее в функции main()нужно инициализировать и создать экземпляр списка ll и вставить первый узел (заголовок), вызвав метод PushFront(“Это”). Далее вставляются новые узлы в конец списка с помощью метода PushBack(“мой”) и “список”::

func main() {

    var ll LinkedList = LinkedList{}

    ll.PushFront("Это")

    ll.PushBack("мой")

    ll.PushBack("список")

ll.PushBack("группы")

Далее нужно вставить новый узел со значением «наверное» после первого узла “Это”, после чего удаляем узел с содержанием “мой”:

ll.PushKey("Это", "наверное")

ll.RemoveVal("мой")

Результаты обработки узлов списка выводятся на монитор с помощью метода IterateList() (рис. 4.10.):

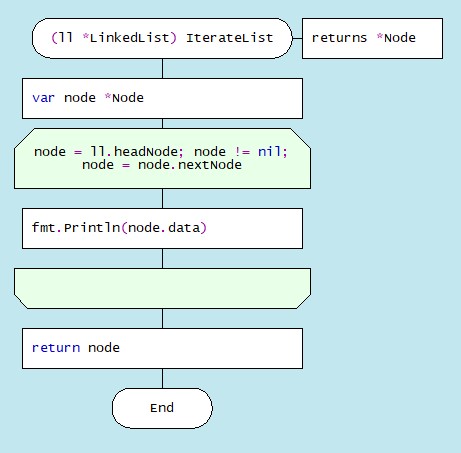


Рис. 4.10. Дракон-диаграмма метода IterateList()

Реализация этих модулей выполняется в функции main():

func  main()  {

    var ll LinkedList = LinkedList{}

    ll.PushFront("Это")

    ll.PushBack("мой")

    ll.PushBack("список")

    ll.PushBack("группы")

    ll.PushVal("Это", "наверное")

    ll.IterateList()

    ll.RemoveVal("группы")

    ll.IterateList()

}

В представленном программном коде элементами списка являются данные типа string. Для того чтобы программа работала со списками, элементами которых являются числа (целые или вещественные), необходимо вместо встроенного типа string использовать сигнатуру interface{}[Manuel]

Результат:

Длина связного списка = 5

Заполненный список

20 -> 30 -> 40 -> 50 -> 70 ->

Длина связного списка = 4

Связный список после удаления узла со значением 40

20 -> 30 -> 50 -> 70 ->

**б). Стек (Stack)**

Стек — это абстрактный тип данных, содержащий элементы с двумя основными операциями: Push, который добавляет элемент в коллекцию, и Pop, который удаляет последний добавленный элемент. В набор носителей этого типа включается набор всех стеков, содержащих элементы типа Т, включая пустой стек, стек с одним элементом типа Т, стек с двумя элементами типа Т и так далее. С технологической точки зрения стек - это память, в которой значения данные загружаются и извлекаются согласно стратегии «последний пришел – первый ушел» (LIFO – Last-In-First-Out). Данные поступают в стек лишь с одной стороны, называющейся **вершиной**стека (рис. 4.11.):

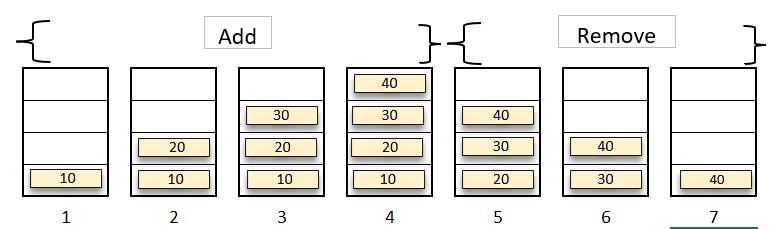


Рис.4.11. Работа стека «последний пришел – первый ушел»

Обычной иллюстрацией стека является стопка тарелок: в любой момент времени доступна только одна тарелка — та, которая поставлена на стопку последней, т. е. верхняя. Для того, чтобы получить доступ к произвольной тарелке внутри стопки, необходимо сначала последовательно снять все тарелки, находящиеся над ней. Наиболее частым применением этой структуры является реализация операции “Отмена последнего действия” в различных редакторах (Ctrl+Z). Другой пример применения организации данных в стеке – рекурсивные обращения функций. Каждый адрес обращения к функции запоминается в стеке и извлекается в обратном порядке.

Согласно предназначению стека основными операциями с его элементами являются функция (метод) добавления Push(item) и функция (метод) извлечения Pop():

func (s \*Stack) Push(str string) {

    \*s = append(\*s, str) // append встроенная функция добавления

}

func (s \*Stack) Pop() (string, bool) {

    if s.IsEmpty() {

        return "", false

    } else {

        index := len(\*s) - 1   // Получить адрес верхнего элемента.

        element := (\*s)[index] // Указать на срез и получить элемент

        \*s = (\*s)[:index]      // срез от 0 до index.

        return element, true

    }

}

Здесь \*s – указатель на адрес стека; при каждом обращении определяется длина стека, вычисляется параметр index для формирования нового среза от 0 до index. В результате извлечений всех элементов стек оказывается пустым.

В процессе вычислений используется дополнительная функция IsEmpty(), которая определяет состояние заполненности стека: пуст стек или заполнен. Возвращает 1, если стек пуст или возвращает 0:

func (s \*Stack) IsEmpty() bool {

    return len(\*s) == 0

}

Прежде чем выполнять какие-либо операции со стеком рассмотрим способы программного создания стеков в языке программирования Golang. Существует два основных подхода к созданию стеков: первый - на основе среза (slice), второй – на основе односвязного списка (Linked List). Согласно первому подходу в функции main() вводится новый тип Stack в виде среза:

s.data = s.data[:len(s.data)-1]:

Это ключевая строка, которая удаляет элемент из стека. Она работает, изменяя срез s.data, чтобы исключить последний элемент. В Go срезы являются динамическими, и вы можете изменить их размер, используя синтаксис s.data[:n], который возвращает первые n элементов среза. Здесь n равно len(s.data)-1, что означает “все элементы, кроме последнего”. Таким образом, последний элемент удаляется из стека.

**в). Очередь (Queue)**

Очередь (Queue**)** — это линейная структура данных, отличающаяся от стека порядком удаления элементов: в стеке удаляется последний добавленный элемент; в очереди - наоборот, удаляется элемент, добавленный первым (рис. 4.12).

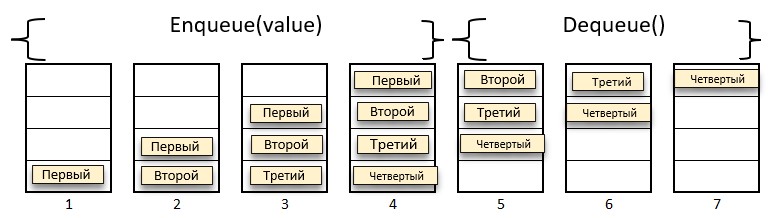


Рис. 4.12. Работа структуры «очередь»

Структура данных в виде очереди находит свое применение в многозадачных системах в системах связи (сетях с промежуточным хранением), в сетях массового обслуживания. Очереди играют важную роль в вычислениях, когда ресурсы предоставляютcя в порядке очереди, например, задания, отправленные на принтер, или процессы, ожидающие процессора в операционной системе.

Абстрактный тип данных Queue определяется как класс, объекты которого реализуют принцип FIFO или First-In-First-Out для добавляемых и удаляемых элементов. С точки зрения концепции АТД очередь — это контейнер, содержащий значения определенного типа. Набор носителей этой структуры данных — это набор всех очередей, содержащих элементы типа Т, включая пустую очередь. Набор основных операций, поддерживаемых очередью, включает в себя:

Add(s) – добавление одного элемента в конец очереди.

Remove() - удаление одного элемента из начала очереди.

Создание очереди реализуется в Golang либо с помощью среза (Slice) либо на основе связного списка (Linked List). В первом случае для создания очереди вводится тип в виде структуры :

type SliceQueue struct {

    queue []string

}

Здесь поле queue []string - целочисленный срез, в котором будут храниться данные.

Ниже представлены основные методы, связные с обработкой элементов очереди - добавление в конец очереди нового элемента Add(s) и удаление элемента Remove():

func (q \*SliceQueue) Add(s string) {

    q.queue = append(q.queue, s)

}

func (q \*SliceQueue) Remove() string {

    temp := q.queue[0]

    q.queue = q.queue[1:]

    return temp

}

Дополнительными методами по отношению к структуре Очередь являются методы IsEmpty(), возвращающий true в случае пустой очереди, Size(), возвращающий длину очереди и некоторые другие.

Работа с очередью в функции main() начинается с создания экземпляра очереди типа SliceQueue с нулевой длиной:

queueSlice := &SliceQueue{make([]string, 0)}

Далее реализуем иллюстрацию работы двух экземпляров очереди, первый – для числовых данных, второй – для строковых.

Результат работы программного кода:

Очередь можно создавать на основе связного списка. Каждый новый узел создается методом newQueue(), имеющим тип \*Queue:

func newQueue() \*Queue {

    return &Queue{

        nil,

        nil,

        0,

    }

}

Ввод каждого нового узла реализуется методом enqueue(val):

func (LL \*Queue) enqueue(value int) {

    var node \*Uzel = getUzel(value) // Создать новый узел

    if LL.headUzel == nil {

        LL.headUzel = node // Добавить узел в начало очереди

    } else {

        LL.tailUzel.next = node // Добавить узел в конец очереди

    }

    LL.count++ // увеличить длину очереди

    LL.tailUzel = node

}