Editorial CIMTED

ISBN: 978-958-53396-4-4

Programación estructurada en GO-LANG

Autor:
© Luis Eduardo Muñoz Guerrero

Primera Edición
Editado en Colombia

2021

Página Legal

Título de la obra: PROGRAMACIÓN ESTRUCTURADA EN GO-LANG
ISBN:978-958-53396-4-4
Materia: Sistemas
Tipo de contenido: Computación y sistemas
Clasificación THEMA: UMX - Lenguajes de programación y extensión / "scripting": generalidades
Público objetivo: Enseñanza universitaria o superior
Editado por: Centro Internacional de Marketing Territorial para la Educación y. El desarrollo CIMTED
Nit: 811043395-0
editorialcimted@gmail.com
Cuidado de edición: Juliana Escobar Gómez
Calle 41 no 80b 120 código postal 055017
Medellín - Colombia
www.cimted.org
www.editorialcimted.com
www.memoriascimted.com
1ª Edición, Pereira-Risaralda. Agosto de 2021

Autor
Profesor Titular
Universidad Tecnológica de Pereira

Miguel Ángel López Fernández
Editor literario, compilador y corrector
Estudiante de Ingeniería de Sistemas y Computación
Universidad Tecnológica de Pereira

Ficha de catalogación en la fuente
Nota legal.

Agradecimientos

*A mi amada esposa **Nancy Portilla**, con su sinceridad y lealtad absoluta, me permite seguir el camino de la honestidad y el amor verdadero.*

Introducción

Este documento presenta los conceptos fundamentales y temas adicionales de la programación imperativa desarrollada con el lenguaje de Go-Lang. Se observarán desde los tipos de datos primitivos, uso de funciones y estructuras de control, hasta el manejo de estructuras de datos como *arreglos*, *datos definidos por el usuario (structs), listas*, *pilas*, *colas y deques*.

Se propone como una lectura para personas que quieran iniciarse en los temas relacionados a la programación imperativa, siendo una base conceptual para experimentar a futuro con temas más avanzados proporcionados por este lenguaje. Por otra parte, cada sección que compone al documento está dividida en teoría y conceptos, definición de sintaxis y ejemplos de prueba, y, por último, ejercicios que recopilen la información analizada mediante su desarrollo y análisis. Ciertos apartados como las funciones y las estructuras de control disponen al final de un conjunto de ejercicios de práctica que permiten afianzar lo desarrollado en dichas secciones.

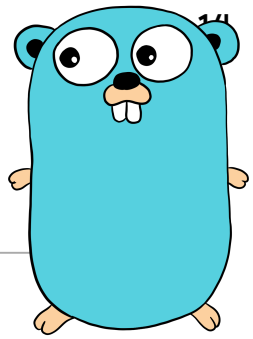
Al final se proporciona la bibliografía que ha sido referencia para digitar toda la información mostrada en este libro. De lo anterior, es de recordar que todo es con base a la documentación oficial que soporta el lenguaje de Go, y que por ende se señala su importancia en cuanto a la comprensión de la sintaxis y funcionalidad del lenguaje.

Tabla de contenido

Lista de ilustraciones

Lista de tablas

ELEMENTOS BÁSICOS DE GO-LANG



Se explicará conceptualmente el lenguaje, aspectos técnicos al momento de su instalación, así como sus elementos y estructuras principales que permiten su debido manejo. Entre estos últimos se encuentran los tipos de datos primitivos como *enteros*, *flotantes*, *booleanos*, *strings*, como los formatos de entrada y salida de información.

Introducción al lenguaje GO-LANG

1.1 Concepto fundamental de GO-LANG

Go (también conocido como Go-lang) es un lenguaje de programación creado por Robert Griesemer, Rob Pike y Ken Thompson, en colaboración con la compañía de Google. Es un lenguaje concurrente, realiza varios cálculos de distintas maneras sin afectar el resultado, y compilado, ya que el código fuente debe pasar por fases de traducción a código de máquina para ser ejecutado.

Otros lenguajes como Pascal, Modula y Oberon han sido influencias importantes, aunque su funcionalidad es similar a la sintaxis de C. Go utiliza características de otros lenguajes y las implementa de manera tal que se facilite la comprensión del código escrito. Su estructura permite el desarrollo del paradigma funcional, imperativa y orientada a objetos.

Comprende algunas otras características como:

-Desarrollo eficiente a través de un recolector de basura (*garbage collector*), permitiendo manipular de una manera más eficiente la memoria usada para cada operación.
-Capacidad para el manejo de subrutinas (a través de go-routines).
-Especialmente hecho para construir estructuras de servidores y herramientas para programadores, como por ejemplo las API's de trabajo.
-Es un proyecto *opensource*.
-Usa tipado estático. Su sintaxis a comparación de muchos otros lenguajes es de inferencia implícita, lo que significa que se puede ahorrar la precisión con la que se crea cada declaración.

1.2 Descarga, instalación e interfaz de trabajo

Para la descarga de GO-LANG se debe de seguir los siguientes pasos:

•Entrar en la página oficial de Go.
•Escoger la versión a descargar (para usuarios ya sea de Windows, Apple o Linux)
•Abrir el instalador y seguir los pasos correspondientes.
•Escoger un editor de texto. Para el desarrollo de los temas se utilizará Visual Studio code. Para este se debe de instalar la extensión de Go a través del buscador de extensiones

•Luego de tener la extensión, cree una carpeta donde pueda almacenar los futuros trabajos realizados. Posterior a ello cree un documento "hello.go", este servirá como verificación de los pasos anteriores.
•Se debe abrir el documento *".go"* .Se mostrará un anuncio de advertencia para instalar el resto de los paquetes necesarios, como los siguientes:

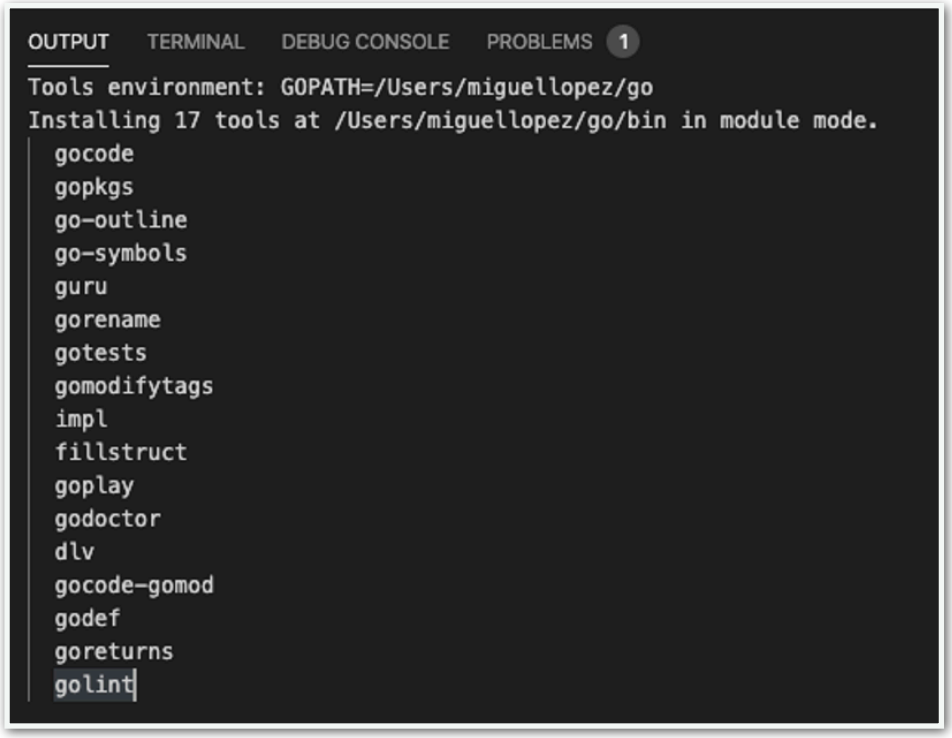


Ilustración 1. Instalación de programas

Al final deberá aparecer un mensaje notificando que la instalación ha concluido.
Con todo lo anterior se deja preparado el ambiente de trabajo para la sintaxis de GO-LANG.

1-3 Primeras interacciones con el entorno

1.3.1 Hello, World!

Para comenzar se dará un primer contacto con el lenguaje Go a través del famoso "Hello, ¡world!", un pequeño programa que apareció por primera vez en 1978 con la obra "The C programming language".
Se debe crear un nuevo documento llamado *hello.go*, abrir en el editor de texto elegido y en este se debe copiar las siguientes líneas:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  fmt.Println(¡”Hello, World!”)
8  }
```

Para ejecutar el código se debe escribir en el terminal del ordenador (o si el editor lo proporciona) *go run hello.go* se debe de visualizar:

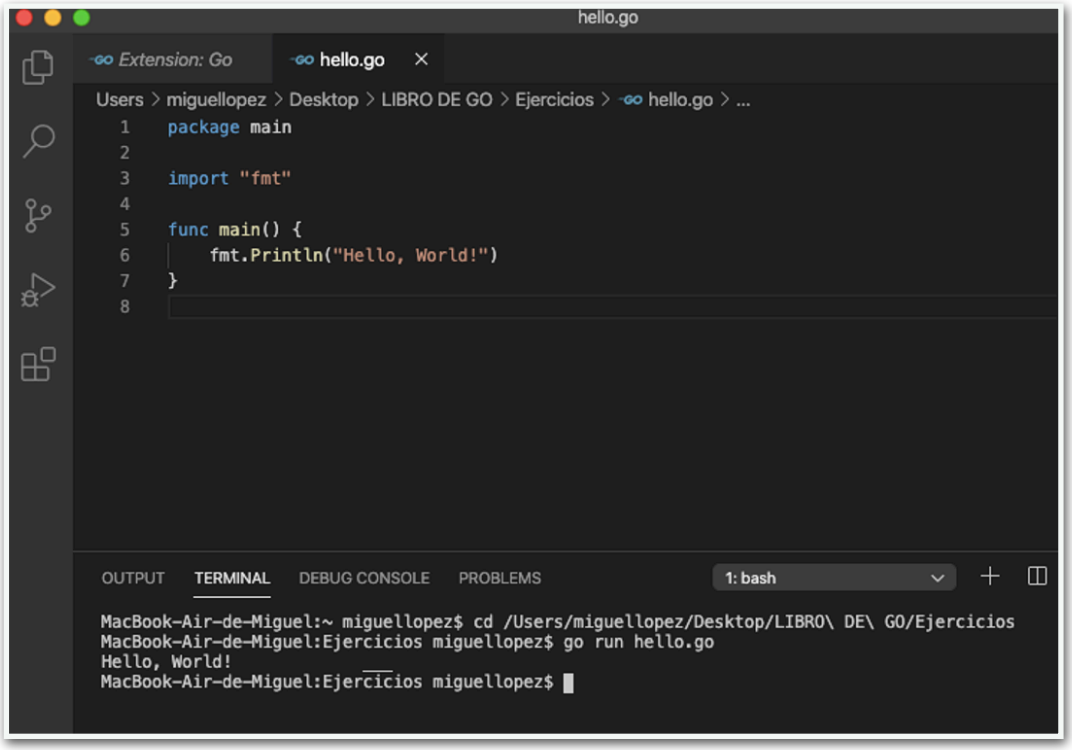


Ilustración 2. Ejecución de Hello.go

Para generar un archivo ejecutable del programa anterior (facilita la posterior ejecución de este) se debe escribir en la terminal el comando build de la siguiente forma:

```
$ go build hello.go
```

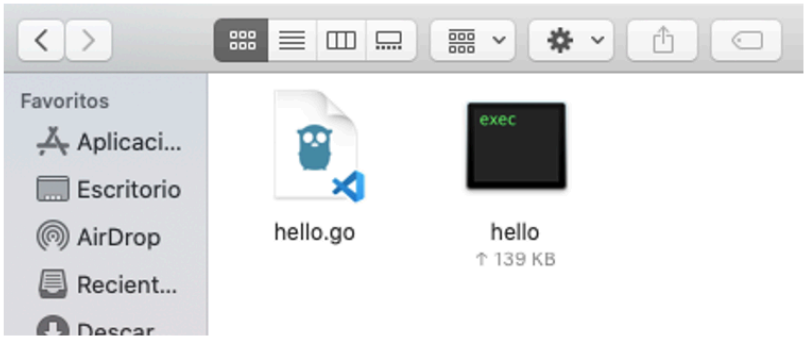


Ilustración 3. Hello world ejecutable

1.3.2 Aspectos técnicos del programa

El lenguaje de Go funciona a través de paquetes, lo que en otros ambientes como C/C++ o Java serian librerías o módulos. Estos paquetes son un conjunto de archivos con extensión .go dentro de un mismo archivo los cuales definen un grupo de acciones que se ejecutan cuando son requeridas. Package main es un indicativo para mencionar la librería a utilizar y en este caso el módulo main es uno de los más comunes. Cabe notar que pueden existir paquetes dentro de otros, lo que facilita el uso de múltiples funciones a partir de una sola dirección de archivos.
Import <nombre de paquete> le indica al compilador qué paquetes utilizar, esto es debido a la restricción propia de Go al traducir un código fuente, donde se debe de ser lo más eficiente con las funciones a utilizar. Fmt es una de las más de 100 librerías pertenecientes a Go, esta provee funciones como fmt.Println o fmt.Scanf que sirven para imprimir y escanear información.

Estructuras de un programa en GO-LANG

2.1 Sintaxis básica

Como cualquier otro lenguaje de programación, Go puede construir programas a partir de un set de instrucciones, puede formar variables que representen un dato a información, organizar estructura de control de flujo usando condicionales *if* o iteradores como el *for*. Permite desenvolver un gran volumen de información y hacerlo comprensible a nivel de código.

En esta sección se mencionarán las estructuras básicas usadas en Go, sintaxis básica, tipos de datos y su uso a partir de ejemplos que representen casos de aplicación.

2.1.1 Operadores aritméticos

Los operadores aritméticos son aquellos que pueden manipular ciertas acciones entre números reales, enteros etc., dichas acciones se conocen como la suma, resta, multiplicación y división.

Nombre	Suma	Resta	Multiplicación	División
Símbolo	+	-	*	/

Tabla 1.Operadores aritméticos

Estos operadores conformaran las operaciones básicas más comunes de la programación, a través de ella se representan múltiples procesos. Para su uso, Go integra una notación infija, es decir, se escriben los números y luego los operados, por ejemplo:

```
3 + 5
65 - 9
2 * 3
8 / 2
```

Cada uno de los operadores tiene un nivel de jerarquía, dependiendo del símbolo, se mantendrá un orden de lectura distinto. Con ello aparecen los paréntesis, signos de control que especifican las partes más importantes de una operación aritmética, por ejemplo:

```
2 / ((3 + 6) * 9)
(5 * 7) / (34 + 8)
```

Al utilizar paréntesis se está indicando que las operaciones que se sitúen dentro de estos se deberán de resolver primero, los paréntesis se eliminan, y se analiza lo que se encuentre después.
Para el caso de no usar otros literales, se debe de tener en cuenta los niveles de jerarquía anteriormente dichos:

Orden
Multiplicación (*)
División (/)
Suma (+)
Resta (-)

Tabla 2.Simbología de operando

2.1.2 Operadores Logísticos

Al igual que los operadores de la suma o de la resta existen expresiones literarias que determinan las comparaciones usadas en la lógica de proposiciones como el *Y* (AND) o el *O* (OR). Este grupo indica el cumplimiento de una condición u operación lógica, muy comunes en estructuras de control como condicionales e iteradores.

Nombre	Símbolo	Función
NOT	~	Niega un dato booleano (dato verdadero o falso)
AND	&	Evalúa dos elementos a la vez. Los dos deben de tener el mismo valor (verdadero o falso), caso contrario devuelve falso.
OR	|	Escoge el valor entre dos datos, ya sea verdadero o falso.
Izquierda	<<	Asigna un valor a la izquierda
Derecha	>>	Asigna un valor hacia la derecha

Tabla 3.Operadores lógicos

Nota: La representación de los resultados de una condición se pueden dar como TRUE o FALSE, la convención de 0 (falso) y 1 (verdadero), o cualquier otra expresión dada para un dato de tipo booleano.

2.1.3 Operadores de comparación

Encargados de comparar dos o más elementos. Se tienen los siguientes:

Nombre	Símbolo	Función
Igual que	==	Determina si dos elementos son iguales
No igual o distinto de	=!	Determina si dos elementos son diferentes
Menor que	<	Si un elemento a es menor que un elemento b
Menor o igual que	<=	Determina si es un elemento a es menor igual que un elemento b
Mayor que	>	Si un elemento a es mayor que un elemento b
Mayor o igual que	>=	Determina si es un elemento a es mayor o igual que un elemento b

Tabla 4. Operadores de comparación

2.1.4 Nivel de procedencia de operadores

El nivel de jerarquía que se debe de seguir es el siguiente:

Nota: El uso de paréntesis puede alterar el orden de lectura de las expresiones.

Niveles de procedencia
* / % << >> & &^
+ |
^
== |= < <= > >=
&&
||

Tabla 5. Simbología de operadores lógicos y de comparación

Los tres tipos de operadores son usados para el análisis detallado de operaciones con diferentes tipos de datos ya sean enteros, flotantes, cadenas de texto, booleano, entre otros.

2.2 Tipos de datos

2.2.1 Enteros

En lenguaje de código se les conoce como "*int*", son un tipo de dato que representa una cantidad numérica entera positiva o negativa. También existe una versión de enteros sin signo o "*unsigned int*". Para su creación solo se debe escribir *int* o *uint*,

estos representaran a manera general la funcionalidad de los datos numéricos enteros.

Cada tipo de dato existente tiene una representación en cuanto al tamaño de memoria que se requiere para su creación. GO permite especificar este tamaño a través de expresiones como ***int8***, ***int16***, ***int32*** e ***int 64***, aunque también los casos sin signo como ***uint8***, ***uint16***, ***uint32*** e ***uint 64***. Dichos números hacen referencia al ancho de memoria soportado por el procesador para la creación y representación de un cierto tipo de número.

2.2.2 Flotantes

Representan los números con decimales. Se destacan por precisar la magnitud de un número, si es grande o pequeño en comparación con otros. Al igual que los enteros, se dividen en dos categorías en cuanto al manejo de memoria ***float32*** y ***float64***, en donde la primera provee una precisión de 6 decimales, mientras que la segunda 15 decimales.

2.2.3 Booleanos

Son datos con dos posibles valores, verdadero o falso. En programación suelen representarse como 0's o 1's, al igual que utilizando las convenciones de ***true*** o ***false***. Los operadores lógicos y de comparación producen resultados de este tipo de dato, usados mayormente en estructuras de control como el *for, while* o *if*.

2.2.4 Strings

Representan secuencias de texto como frases u oraciones. En términos de computación, se conocen como secuencias de bytes inmutables interpretadas a través del formato de codificación de caracteres UTF-8. Más adelante se profundizará sobre el funcionamiento interno y las funciones guiadas al manejo de cadenas.

2.2.5 Representación de datos en pantalla

Gracias al paquete *fmt* Go puede asimilar el funcionamiento de entrada y salida de datos con el sistema de C, usando formatos de *print* y *scanf*. Algunos ejemplos son:

package main

Para la presentación de información en pantalla se usa *fmt.print*:

```
 1  import "fmt"
 2
 3  func main() {
 4
 5  c := "cadena de texto"
 6  i := 8
 7  f := 4.56789
 8
 9  fmt.Println("Esto es una línea de texto")
10  fmt.Printf("Esto es una %s\n", c)
11  fmt.Printf("Hay %d manzanas\n", i)
12  fmt.Printf("%f\n", f)
13  }
```

Nota: La asignación ":-" se analizará en capítulos posteriores.

Como resultado:

```
>   Esto es una línea de texto
    Esto es una cadena de texto
    Hay 8 manzanas
    4.567890
```

Para la lectura de datos se utiliza ***fmt.scanf*** :

```
1  package main
2
3  import "fmt"
4
5  func main() {
6  var i int
7  fmt.Println("¿Cuál es tu edad? ")
8  fmt.Scanf("%d", &i)
9  }
```

Como resultado:

```
>   ¿Cuál es tu edad?
    17
```

2.3 Tipos de lecturas e impresiones

Se tienen las siguientes formas:

IMPRIMIR DATOS	fmt.Print(...)	Muestra en pantalla los datos sin formato.
	fmt.Println(...)	Muestra en pantalla los datos sin formato y con un salto línea después de leer el último carácter
	fmt.Printf(...)	Muestra los datos en pantalla usando los formatos de cada dato.
LEER DATOS	fmt.Scanf("%...", &...)	Lee datos ingresados por teclado. El formato de lectura para cada dato es distinto.

Tabla 6. Impresión de datos en pantalla

Formatos de impresión y lectura	
Símbolo	**Descripción**
Generales	
%v	Valor por defecto.
%T	Obtener el tipo de dato manejado.
Enteros	
Valor por defecto %d	Enteros con base 10
%o	Enteros con base 8
Flotantes	
%f	Punto decimal sin exponentes
%e, %E	Notación científica
%x, %X	Notación hexadecimal
Strings y bytes slice's	
Valor por defecto %s	Cadena de caracteres en bits
%p	Dirección de la posición 0 con base 16
Punteros	
Valor por defecto %p	Dirección. Pueden usarse otros componentes de datos anteriores como %b, %d, %o,
Valor por defecto %p	Dirección. Pueden usarse otros componentes de datos anteriores como %b, %d, %o,

Tabla 7. Formatos de impresión

Nota: No se mencionan todas las posibilidades de formato para cada tipo de dato por lo que se recomienda consultar la documentación de GO.

2.3.1 Ejemplos

Enteros y flotantes

```
1  package main
2  import "fmt"
3
4  func main () {
5  //Enteros
6       var n int
7       fmt.Printf("%v, %T\n", n, n)
8
9  //Flotantes
10      var x = 4.5e78 // Se determina como valor base float64
11      fmt.Printf("%v, %T\n", x, x)
12 }
```

Var es la estructura que designada para la creación de variables. En este caso se utiliza la siguiente forma:

var <nombre de la variable> <tipo de dato>

Al ejecutar el código se obtiene:

```
>   0 int
        4.5e+78 float64
```

En el caso de tener una variable no inicializada Go asigna un cero por defecto al valor de esta. Esto se presenta únicamente para los datos de tipo de entero.

En la siguiente línea se presenta *var x* inicializada de forma flotante exponencial. Para este caso los flotantes de manera automática se toman como *float64*, sin embargo, esto se puede precisar al momento de su creación.

Strings y bytes

```
1  Package main
2
3  Import “fmt”
4
5  Func main{
6       s := "soy una cadena"
7       t := "estoy junto a una cadena"
8       fmt.Printf("%v\n", s+t)
9
10      b := []byte(t)
11      fmt.Printf("%v, %T", b, b)
12 }
```

Se obtiene lo siguiente:

```
>  soy una cadenaestoy junto a una cadena
   [101 115 116 111 121 32 106 117 110 116 111 32 97 32 117 110 97
   32 99 97 100 101 110 97]
```

Dentro del *main*, los primeros renglones definen las variables *s* y *t*. Note que esta vez se usa un formato de asignación distinto (“ := ”). Esta nueva forma evita precisar el tipo de dato utilizado y de manera automática Go lo reconoce para futuros cambios.

Algunas operaciones entre cadenas se volverán mucho más usual en su construcción, como se puede observar en *printf* donde se suman las dos variables *s* y *t* para formar una nueva cadena.

La variable *b* como su definición indica, se asignarán valores de tipo byte provenientes de la cadena *s*. Se leerá la cadena por completo calculando el bit representativo en su definición ASCII, como por ejemplo la letra ‘*a*’ que según el estándar mundial se define con el número 97. Este formato será utilizado más adelante en los *bytes slice’s* para comunicación entre redes

Operadores binarios y asignaciones

```
1   //Operadores lógicos y de comparación
…   …
4       n := 2 == 3
5       m := 5 == 5
6
7       fmt.Printf("%v, %T ", n, n)
8       fmt.Printf("%v, %T\n", m, m)
9
10      //Representación de operadores binarios, lógicos y aritméticos
11
12      a := 5 //0101
13      b := 3 //0011
14      // 0 is false, 1 is true
15      fmt.Println(a & b)  // Se unifican los dos valores con 'AND', 0001
16      fmt.Println(a | b)  // Se unifican los dos valores con 'OR', 0111
17      fmt.Println(a  ^  b)    //  Se  unifican  los  dos  valores  con  'XOR'
(bicondicional), 0110
18      fmt.Println(a &^ b) // Se unifican los dos valores con 'and/or' , 0100 ,
entender mejor
19      y := 5
20      fmt.Println(y << 4) //en representación de exponenciales (binarios), se
añade 4 al exponente
21      fmt.Println(y >> 2) //en representación de exponenciales (binarios), se
resta 2 al exponente
```

El código anterior resulta en lo siguiente:

```
>   false, bool true, bool
    1
    7
    6
    4
    80
    1
```

Las primeras líneas ejecutan instrucciones lógicas de comparación, en donde se evalúa en las variables *n* y *m* la condición sobre si dos números son iguales. Como resultado se generan valores *true* y *false*.

En concordancia de operadores, todos se relacionan en el ámbito de la lógica de Bool o lógica de puertos. Cómo se indicó anteriormente los operadores lógicos (de la mano con los comparativos) funcionan para el uso de condiciones, por ello para comprender el análisis que realiza la maquina en el momento de ejecución se debe tener claro las situaciones presentes para cada operando, dadas en lo que se conoce como una tabla de verdad, como la que se muestra a continuación:

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabla 8. Tablas de verdad de operadores lógicos

Al realizar cada ejecución, dependiendo del operador y las variables dadas, se determina una verdad o falsedad. Para el caso de números que no representen una condición fija como lo es el código presente, lo que se hace es asumir la representación binaria de cada dato y hacer operaciones entre 0's y 1's de la siguiente manera:

Los números son 5 y 3, con representaciones binarias de 0101 y 0011 respectivamente. Teniendo en cuenta la tabla de verdad anterior, y dependiendo el operador en cuestión, se realiza:

(Para el caso 1)

$$\begin{array}{rr} & 0101 \\ \& & \underline{0011} \\ & 0001 \end{array}$$

La representación en binario del resultado es el número 1.

2.4 Ejemplos de estudio

Se proponen los siguientes ejemplos:

1.Crear un programa que permita calcular el volumen de un cilindro, una esfera y un cono. Las ecuaciones de las respectivas figuras son:

Cilindro = $\boldsymbol{\pi * r^{2} *}$ h Esfera = $\boldsymbol{4/3 * \pi * r^{3}}$ Cono= $\boldsymbol{1/3 * \pi * r^{2} *}$

2.Calcular el número más grande de 5 números dados.

3.Realizar las operaciones básicas con números de tipo entero y números de tipo flotante.

4.En un teatro cada cliente paga $10.000 por entrada y cada función le cuesta al teatro $300.000 por la atención prestada. Por cada cliente que entre el teatro debe pagar un costo de $2.000 por aseo. Desarrollar un programa que reciba el número de clientes de una función y devuelva el valor de las ganancias obtenidas.

5.En un supermercado se ofrecen descuentos por el total del valor en cada uno de los siguientes productos: carnes con 10%, frutas con el 5%, aseo con el 7%, dulces con el 9%.

6.Una persona desea llevar $10.550 en dulces, $50.000 en carne y $35.000 en productos de aseo. Se necesita calcular el descuento total por cada tipo de producto, posteriormente entregar el valor total a pagar con descuento y sin descuento.

7.Verificar si dos números ingresados son iguales.

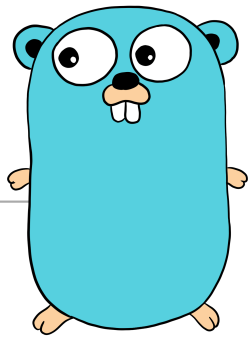
8.Verificar si un número es menor a 10.

9.Hacer un programa que determine verdadero o falso si un número es mayor o igual que 10 y menor que 20 o mayor que 30.

10. Dado un número de 3 cifras, descomponerlo en sus unidades fundamentales y decir la cantidad correspondiente, ejemplo: el número 631 tiene una unidad, 3 decenas y 6 centésimas.

11. Del ejercicio anterior, descomponer un numero dado e invertirlo, por ejemplo, 582 pasaría a ser 285.

FUNCIONES Y ESTRUCTURAS DE CONTROL



En esta sección se profundizará en el concepto de funciones y estructuras de control como los condicionales e iteradores para el uso del lenguaje a un nivel más complejo.

Funciones

En el mundo de la programación, es muy común el término "Funciones" al momento de escribir código. Hay situaciones en las que se requiere escribir programas que realicen demasiados pasos para distintas acciones, una tras otra, y de manera repetida. Para estos casos se recomienda dividir el problema en pequeñas partes, seccionar las actividades que se deben realizar para luego, paso por paso desarrollar cada uno hasta obtener una solución.

Como ejemplo se tiene la siguiente situación:

"Como programadores, dados dos puntos (X,Y) pertenecientes al plano cartesiano en dos dimensiones, calcular la distancia existente entre ellos"

Nota: Siempre, antes de iniciar a resolver un problema se debe identificar lo que se pide, modelar el problema antes de su implementación.

Se pide calcular una distancia entre dos puntos dados, dichos puntos serán de a dos números cada uno. Dicha distancia se puede calcular a través de la ecuación:

$$\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2)}$$

Observando el problema desde una perspectiva más amplia, el conjunto de operaciones a realizar es más grande. Usando el concepto anterior, se puede dividir el problema de la siguiente forma:

- Declaración de variable a utilizar
- Recibir y almacenar información
- Creación de formula
    - Resolución de operaciones internas a la raíz
        - Restas entre coordenadas
        - Cálculo de cuadrados
        - Suma de resultados internos de la raíz
    - Calcular la raíz.
- Retornar la respuesta final

Cada una de esas pequeñas partes se conoce como función (o subrutinas). Estas son encargadas de realizar una acción específica dentro de un algoritmo más amplio. Se componen de un conjunto de sentencias (órdenes) las cuales indican el proceso que se ejecuta cada vez que estas estructuras son utilizadas. Son una parte fundamental de la programación, ya que permiten una escritura modularizada, mantener una eficiencia tiempo-espacio, calidad de estética, entre muchos otros componentes.

3.1 Estructura de funciones en Go

Una función se compone principalmente de dos elementos, un *nombre* único en el ambiente de código y una *lista de parámetros*.

func name (<lista de parámetros>)(<tipo de resultado a entregar>)
{
**Cuerpo de instrucciones*
}

El campo *name* indica el nombre único para una función.

La lista de parámetros describe el tipo de datos que recibe la función (vistos en *Estructura de un programa en Go*), y con los cuales se trabajará en el cuerpo de instrucciones. A un lado de los argumentos se especifica el tipo de dato que la función retornara, ya sean datos enteros, flotantes, vectores, entre otros.

El cuerpo de instrucciones, como su nombre lo indica, resguarda el algoritmo realizado por la función.

3.1.1 Uso de la función

Para poder utilizar una función se es necesario "llamarla" en el momento requerido, es decir, identificar mediante su *nombre* la función en el momento necesario de un programa. Los llamados son de la siguiente manera:

func name (<lista de parámetros>)(<tipo de resultado a entregar>){
**Cuerpo de instrucciones*
}

...

func main(){
...
Name(<lista de argumentos>)
...
}

Se observa un nuevo concepto, *argumento*, el cual se sitúa de forma similar que *parámetros* unas líneas más arriba. Son términos similares, pero su diferencia radica en el momento de uso. Sus definiciones son:

•**Argumento(función):** Son los valores con los cuales se desarrollan las instrucciones almacenadas en la función. Estos valores son asignados en el llamado de una función.

•**Parámetro(función):** Indican el tipo de datos que se pueden usar en el algoritmo que desarrolla la función en su cuerpo de instrucciones. Se construyen en el momento en el cual la función se crea y almacenan valores de cierto tipo cuando esta última es "llamada".

Hasta este punto, el concepto de función se ha venido manejando implícitamente con la construcción de *main()*, una función que indica al lenguaje de Go que es la parte importante de todo un programa, la que contiene toda la información necesaria y es por la que se debe comenzar a compilar.

El "llamado" a las funciones no se restringe, puede hacerse en cualquier lugar dentro de una o varias funciones (también pueden ser distintas de un *main()*) y las veces que sea necesario, siguiendo la misma estructura de uso.

3.2 Variables

Se identifican como tipos de dato que almacenan información que puede estar en constante cambio. Su estructura es:

var <nombre> <tipo de dato> = <inicialización>

Las variables se consideran un tipo de dato declarativo, es decir, un dato que define o nombra algún elemento particular. Pueden ser llamadas de múltiples formas, aunque se debe evitar crearlas con nombres alusivos a funciones prediseñadas de Go. Se componen gracias a los tipos de datos explicados anteriormente, existiendo gran cantidad de opciones como variables para datos enteros, complejos, cadenas, arreglos, etc.

El termino inicialización designa la acción de asignar un valor inicial a la variable creada. Este valor inicial puede ser el que se busca implementar, o simplemente un valor alternativo que mantenga la variable preparada hasta el momento de reasignación.

Dentro de un esquema de código se pueden manejar dos tipos de variables:

- **Globales:** Variables que pueden ser usadas en todo momento y lugar.
- **Locales:** Variables que pueden ser únicamente utilizadas en su ambiente de declaración como por ejemplo las funciones, estás utilizan variables locales para definir su funcionamiento interno.

3.2.1 Uso de *make()* y *new()* para la declaración de variables con nueva inicialización y asignación de memoria.

Las funciones que se presentan a continuación permiten crear variables con valores de inicialización base y nueva memoria proveniente de almacenamiento dinámico (o heap).

•**Make():** Función que únicamente crea y retorna *slices*, *mapas* y *canales* que serán almacenados dentro de una variable. Esta función no retorna información inicializada con ceros, mas, sin embargo, go automáticamente inicializa las nuevas declaraciones de variables con cero

•**New():** Función que asigna nueva memoria a todo tipo de datos. Recibe un tipo de dato y retorna la dirección del tipo de dato creado. Esta función, en comparación con la anterior si regresa la información inicializada con cero.

Un ejemplo de las dos funciones sería el siguiente:

```
1  package main
2
3  import (
4        "fmt"
5        "reflect"
6  )
7  type object struct {
8        property1 int
9        property2 string
10 }
11
12 func main() {
13
14 z := make([]int, 10)
15       v := new(object)
16
17       fmt.Printf("La estructura de datos almacenada en z ->  %v || El tipo de
   dato-> %v\n", z, reflect.TypeOf(z))
18       fmt.Printf("El puntero almacenado en v ->   %v || Lo que almacena la
   dirección en v -> %v || El tipo de dato-> %v", v, *v, reflect.TypeOf(v))
19 }
```

En el código anterior se definen dos variables, *z* y *v*, donde la primera corresponde a la creación de un *slice* por medio de la función *make()*, lo que indica que *z* puede comportarse como un *slice*; por otro lado, *v* crea una nueva instancia de la estructura *object* con nueva memoria, y que a partir de la cual se podrá manejar las características que este contiene.

El resultado del código al ejecutarlo es el siguiente:

```
La estructura de datos almacenada en  z ->  [0 0 0 0 0 0 0 0 0 0] || El tipo de dato-> []int
El puntero almacenado en v ->  &{0 } || Lo que almacena la dirección en v -> {0 } || El tipo de dato-> *main.object
```

A lo largo del desarrollo del libro se mostrarán en ciertos apartados el uso de *make()* o *new()* para la declaración de nueva información.

Nota: Algunos de los términos mencionados en esta sección se profundizan en secciones posteriores.

3.3 Constantes

Al igual que las variables, son datos declarativos que crean valores con información constante, que no puede ser cambiada en ningún momento. En un programa pueden definirse matemáticamente valores constantes como el valor **π** o el número de *e*.

También se dividen en dos grupos, constantes globales y constantes locales, que al igual que en las variables, la primera será de uso general para todo momento y espacio, mientras que la segunda solo podrá utilizarse en su ambiente declarativo.

Estas pueden definirse en go con ayuda de la estructura:

```
Const <nombre de la constante> = <valor>
```

En el campo *valor*, se pueden hacer declaraciones con datos booleanos, numéricos y cadenas.

La declaración de las contantes no está definida con la sintaxis ": =".

3.4 Declaración y uso de variables

Las funciones en Go pueden ejemplificarse de la siguiente manera:

```
1  package main
2
3  import "fmt"
4
5  func suma(x int, y int) int {
6       var sum int
7       sum = x + y
8       return sum
9  }
10
11 func main() {
12
13      fmt.Printf("Resultado de la suma %v, y el dato es de tipo
   %T\n", suma(9, 13), suma(9, 13))
14 }
```

Se crea una función *suma* que recibe dos parámetros enteros *x* y *y*. Se especifica el tipo de dato retornar como entero *int*. Se crea una variable local llamada *sum* la cual será la operación de suma entre *x* y *y* como se muestra en la línea siguiente. Se debe recordar que Go implícitamente inicializa las variables a cero, evitando dejar valores aleatorios. Al final se llama al elemento *return* que devuelve el resultado almacenado en la variable *sum*.

EL uso de la función se hace dentro de un campo *main* donde esta se llama con la expresión *sum(parámetros)* los cuales se mostrarán en pantalla con el formato de impresión *fmt.printf*.
El resultado obtenido es:

```
>  Resultado de la suma 22 y el dato es de tipo int
```

Se debe tener en cuenta que todas las funciones creadas deben de tener un elemento de retorno, a excepción de que no se pueda encontrar el final de la función por una llamada a *pánico* o *recursiones infinitas*.

Nota: Funciones como pánico y otros elementos de control y eficiencia se verán en otras secciones.

Las funciones permiten maneras distintas de organizar la entrada de parámetros. Se tienen las siguientes formas:

```
1  package main
2
3  import "fmt"
4
5  func add1(x int, y int) int { return x + y }
6  func add2(x, y int) (z int) {
7       z = x + y
8       return
9  }
10
11 func main() {
12
13      fmt.Printf("%v, %T\n", add1(1, 2), add1(1, 2))
14      fmt.Printf("%v, %T\n", add2(1, 2), add2(1, 2))
15 }
```

Cada una de las formas expresa un manejo distinto, pero retornan el mismo resultado, por un lado, la función *add1* especifica el tipo de argumento que se recibe, mientras que la función *add2* resume el camino escribiendo todas las funciones que serán de un mismo tipo.

Como se verá a continuación las dos retornan el mismo valor:

```
>  3, int
   3, int
```

Cabe resaltar que las variables se insertan en los argumentos de una función a través de paso por copia, lo que significa que los valores enviados son una copia del valor real de la variable que se está analizando. Con ello no se modificará el valor inicial, o de otra manera, la variable local que se está dando como argumento permanecerá igual.

Existen otros métodos de paso de información, como el uso de punteros o referencias, mapas o canales. Estas nuevas maneras permiten trabajar con los valores originales que toma un argumento, por lo que la variable inicial se puede ver afectada a cambios emergentes.

3.5 Retorno de múltiples valores

Las funciones en Go permiten, en algunos casos permiten retornar múltiples valores. Muchos de estos se componen de un resultado y un dato de verificación de ejecución, por ejemplo:

```
1  package main
2
3  import "fmt"
4
5  func twoValues() (int, int, string) {
6       return 45, 36, "valor"
7  }
8  func main() {
9
10      a, b, c := twoValues()
11
12      fmt.Printf("%v, %T, %v, %T, %v, %T\n", a, a, b, b, c, c)
13 }
```

Como resultado se obtiene:

```
>  45, int, 36, int, valor, string
```

La función *twoValues* no contiene argumentos, pero integra tres resultados en el campo de retorno. Se devolverán dos datos de tipo entero y uno de tipo string. Se utiliza la misma estructura con *return* indicando cada uno de los resultados a enviar.

Dentro de una función *main* se almacenan los valores obtenidos por la función. Cabe notar que no se especifica el tipo de valor de las variables usadas, además que estas se están asignando de manera resumida con el carácter ':=', esto debido a que el tipo de dato se especifica al momento de unificar los valores, es decir, en el instante en que se obtienen los nuevos datos, partiendo de la información que contengan, se configuran los datos de almacenaje.

3.6 Funciones variádicas

Las funciones variádicas son estructuras que permiten el uso de un número indefinido de argumentos. Se puede tomar como ejemplo *printf*, función que recibe

determinado número de datos y los ajusta dependiendo del tipo de formato a imprimir.

La estructura de estas funciones consiste en utilizar como regla de argumentos el tipo de dato a manejar antecedido por puntos suspensivos, de la siguiente manera:

```
func name (<…<tipo de dato>>)(<tipo de resultado a entregar>)
{
*Cuerpo de instrucciones
}
```

De una manera más detallada se tiene la siguiente función:

```
1  package main
2
3  import "fmt"
4
5  func suma(números ...int) int {
6       total := 0
7
8       for _, números := range números {
9            total = total + números
10      }
11      return total
12 }
13
14 func main() {
15
16      fmt.Println(suma(2, 2, 2))
17      fmt.Println(suma(6))
18      fmt.Println(suma(2, 3, 4, 5, 6, 7, 8))
19
20 }
```

Como resultado se obtiene:

>	6
	6
	35

Se puede observar que la función *suma* recibe un argumento *números* precedido por la estructura de argumentos variadicos *"...<tipo de dato>"*. Cada llamado a la función permite el envió de múltiples valores, por lo que se recurre a la utilización de un iterador *for* cada vez que se envían más valores. Al final se retorna el *total*, el cual estará almacenando la suma realizada entre cada valor pasado a la función. Dentro de un "*main*" se imprimen en pantalla los posibles valores que admiten las líneas anteriores.

Un detalle importante con el uso de este tipo de argumentos es su posición dentro de una función. Cuando se quiere incorporar a funciones con más de un tipo de argumento, el argumento variadico a incorporar debe de ir al final del total de parámetros a ingresar ya que la estructura de Go no reconoce el final de un argumento con tamaño indefinido.

Nota: Se recomienda utilizar los ejercicios del capítulo anterior y volverlos funciones. Algunos de ellos permiten usar funciones varíadicas.

3.7 Ejemplos de estudio

A partir de la información descrita en este capítulo, se propone desarrollar los siguientes ejercicios:

1.Del capítulo "*2. Estructuras de un programa en GO*", desarrollar los ejercicios propuestos en modo de funciones.

2.Crear un programa que determine el mayor de 3, 4 y 5 números.

3.Se necesita crear un programa que reciba la fecha de un año e indique si este es o no bisiesto.

4.Crear un programa que, dado un intervalo de números entero, verifique si la suma de los elementos comprendidos en el intervalo da como resultado un numero primo.

5.Construir un programa que reciba un numero de n dígitos y devuelva (según el número de dígitos) las unidades utilizadas con su cantidad. Por ejemplo:

Se recibe 356, se debe retornar como respuesta:
6 unidades

- 5 decenas
- 3 centenas

6. Construir una función que reciba un número entero y cuente el número de veces que se repite un dígito (si existe).

7. Crear una función que permita llenar un vector de n digitas (por teclado) y calcule cual es el mayor de todos los elementos.

8. Crear una función que permita ordenar un vector de mayor a menor y menor a mayor.

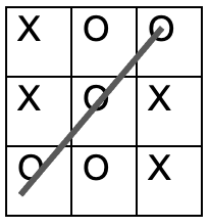
9. Construir un programa que, dados dos vectores, devuelva dos nuevos vectores que junten los números pares e impares de cada uno.

10. Crear un programa capaz de recibir una lista de datos y calcular la media, mediana, moda y desviación estándar del conjunto de datos.

11. Construir una función que, dadas dos matrices, calcule la suma y multiplicación de sus elementos. (Tener en cuenta propiedades de las matrices)

12. Construir una función en donde dada una matriz, calcule su matriz transformada. (Tener en cuenta propiedades de las matrices)

13. Construir un programa que pueda recrear el juego *Tic-tac-toe*. Serán dos jugadores posibles y gana quien alcance a completar 3 casillas en línea o diagonal, por ejemplo:



Estructuras de control y flujo

En programación se conocen como estructuras de control aquellos modelos que permiten modificar el flujo de la información. Basada en la programación estructurada, el lenguaje de Go permite secuencias de tipo iterativas y condicionales que permiten analizar la información a partir de ciertas características:

- De acuerdo con una condición
- De acuerdo con un valor
- De acuerdo con un rango de procesos que lleguen a cierto valor

Cada estructura es similar para la mayor parte de lenguajes de programación, con la diferencia de sintaxis y posiblemente comandos especiales.
En Go se manejan las siguientes estructuras:

4.1 Condicionales

También llamadas de selección, son estructuras que proporcionan un conjunto de operaciones siempre y cuando se cumpla una condición. Algunos modelos son:

4.1.1 Condicional *if*

Se toma la siguiente forma:

```
If( <condiciones> ){
    <Bloque de instrucciones #1>
}
else {
    <Bloque de instrucciones #2>
}
```

Si el campo *condiciones* es verdadero se ejecuta el *bloque de instrucciones #1*. Si es falso, se ejecuta el *bloque de instrucciones #2*.

Este modelo permite el anidamiento de estructuras, así como la verificación para cada operación ya sea verdadera o falsa.

4.1.2 Condicional switch

Al igual que el *if*, es un condicional que ejecuta un conjunto de operaciones dependiendo de un valor resultante. Se compone de la siguiente manera:

```
switch (<variable>){

    case <Valor variable 1> {<bloque de instrucciones #1>}
    case <Valor variable 2> {<bloque de instrucciones #2>}
    .
    .
    .
    case <Valor variable n> {<bloque de instrucciones n>}
    case else {<bloque de instrucciones>}
    end switch
}
```

•Se ingresa un valor el cual se almacena en *variable*
•Se recorren todos los *Case*, buscando aquel que sea igual al valor de *variable*
•Si este se encuentra se ejecuta el bloque de instrucciones
•Caso contrario se ejecuta el *Case else* (si se propone) con su bloque de instrucciones.
•Se termina la ejecución de switch.

4.1.3 Iteraciones

Componen un conjunto de operaciones que se ejecutan un determinado número de veces, esto hasta que se cumpla o se mantenga una condición. Uno de los modelos más utilizados es el *for* el cual aparece mayormente con la siguiente estructura:

```
for ( <variable = valor de inicio>, [variable (<,<=),(>,>=) o (=) tope], paso) {
                                <bloque de operaciones>
}
```

1.Se crea e inicializa una variable
2.Se forma una condición que indicara el número de repeticiones. Esta zona se determinará a partir del rango de números que llegara a tomar la *variable*, por ejemplo, decir [*variable < 8*] indica que el trabajo del iterador será hasta que se obtenga el valor de ocho en *variable*.
3.Se define el tipo de paso o recorrido que tendrán las asignaciones de valores para la *variable*. Estos pueden ser de tipo +1, +2, . . ., +n.
4.Dependiendo del paso definido se llegará más o menos rápido a la condición inicial, momento en el cual se detienen las iteraciones y se termina la acción del *for*.

Nota: Modelos como el while o do-while no hacen parte de la sintaxis de Go.

4.2 Estructuras de condición

Una de las estructuras de condición o secuencia manejadas en Go es el *if*. Por ejemplo:

```
1  package main
2
3  func main() {
4
5       if 5 > 6 {
6
7       }
8  }
```

A partir de estas estructuras se pueden tener múltiples funcionalidades, ya sea el anidamiento de estructuras, manejo de múltiples situaciones, etc.

Nota: Los condicionales *if* no necesitan del uso de paréntesis que rodean la condición, más si se es necesario el encerrar el bloque de instrucciones con {}.

4.3 Ejemplos

A partir del siguiente código se desglosa las estructuras que componen a un condicional *if* dentro del lenguaje Go.

Problema: Se propone crear un programa que simule un juego de adivinanza de números, donde el usuario puede ingresar por teclado un valor y se debe retornar si la información suministrada es el número para adivinar, caso contrario donde el numero sea mayor o menor, retornar un indicativo.

4.3.1 Conceptualización y categorización

La situación planteada emplea una condición para poder satisfacer la necesidad. Se quiere introducir un número y verificar que este dato sea el correcto. Se pueden utilizar los temas vistos en capítulos anteriores conjunto a una sentencia de condición para desarrollar el código.

4.3.2 Desarrollo

Se debe plantear las variables a manejar, conjunto a la popularización del programa, por lo que se propone lo siguiente:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6       //Programa para adivinar un numero almacenado
7       var numero int
8
9       fmt.Print("Digite un numero: ")
10      fmt.Scanf("%d", &numero)
11 }
```

Luego de crear las entradas de información, se procede a crear la parte lógica de lo pedido por el ejercicio.

4.3.3 Operadores de comparación

Al evaluar condiciones, directamente se incorporan expresiones booleanas como el AND, OR y NOT (&, |, ~), así como elementos de comparación (<, >, =, <=, >=). Estas expresiones ayudan a evaluar condiciones que requieran de un o más parámetros. El ejercicio en cuestión propone adivinar un número, previniendo situaciones como de si el numero ingresado es menor o mayor que el buscado. Para esto se puede hacer lo siguiente:

```
1  var adivinar int = 30
2
3       if numero == adivinar {
4            fmt.Println("HAS ADIVINADO")
5       }
6       if numero < adivinar {
7            fmt.Println("Muy por debajo")
8       }
9       if numero > adivinar {
10           fmt.Println("Muy por encima")
11      }
```

El numero para adivinar será 30; Las posibles situaciones (igual, menor y mayor) se describen usando expresiones de comparación.

Para obtener resultados por cada condición planteada se debe de ejecutar el programa cada vez que se quiera ingresar un nuevo número. Con esto, se puede obtener la siguiente comprobación:

```
>  Digite un numero: 5
   Muy por debajo
   Digite un numero: 100
   Muy por encima
   Digite un numero: 30
   HAS ADIVINADO
```

4.3.4 Operadores lógicos

Para dificultar un poco más la resolución del problema, se restringirá la entrada de datos a un intervalo de datos. Para ello se puede realizar otro segmento *if* el cual determinaría la evaluación de las situaciones anteriores, por ejemplo:

```
1   if numero > 1 && numero < 100 {
2           if numero == adivinar {
3                   fmt.Println("HAS ADIVINADO")
4           }
5           if numero < adivinar {
6                   fmt.Println("Muy por debajo")
7           }
8           if numero > adivinar {
9                   fmt.Println("Muy por encima")
10          }
11      }
```

El ingreso de datos solo podrá ser válido para valores que se encuentren en un rango de 1 a 100, descrito a partir de expresiones booleanas AND. Nótese que el operador lógico es utilizado con dos expresiones, indicando una acción de comparación lógica.

4.3.5 Estructura else

La estructura *if* puede manejar múltiples valores, ya sea segmentos para un resultado verdadero, como segmentos para un resultado falso, esto repitiéndolo fuera o dentro

de la misma estructura. Para el problema en desarrollo, podríamos ejemplificar un caso contrario al momento de no cumplirse la condición inicial:

```
1  if numero > 1 && numero < 100 {
2            …
3      } else {
4            fmt.Println("Ingrese un número valido")
5      }
```

Al ejecutarlo, se muestra como resultado lo siguiente:

```
>  Digite un numero: 100
   Ingrese un número valido
   Digite un numero: 5
   Muy por debajo
```

Cada vez que se ingresa un dato que no cumpla con la condición inicial, se ejecutan las líneas almacenadas en el campo *else*, las cuales funcionan como segunda opción. Se debe ser cuidadoso en la formación de un *else* ya que este se maneja únicamente en la terminación de un condicional, después de las llaves cerradas.

4.3.6 Condiciones anidadas y estructura else-if

El código desarrollad hasta el momento puede adquirir algunas modificaciones, como las que se muestran a continuación:

```
 1  package main
 2
 3  import (
 4        “fmt"
 5  )
 6
 7  func distancia(numero, adivinar int) bool {
 8       var A bool
 9
10       if numero-adivinar >= -10 && numero-adivinar <= 10 {
11            A = true
12        } else {
13            A = false
14        }
15
```

```
16        return A
17  }
18  func main() {
19        //Programa para adivinar un numero almacenado
20        var numero int
21
22        fmt.Print("Digite un numero: ")
23        fmt.Scanf("%d", &numero)
24
25        var adivinar int = 30
26
27        if numero < 1 {
28              fmt.Println("El número debe ser mayor a uno")
29        }else if numero > 100 {
30              fmt.Println("El número debe ser menor a cien")
31        } else {
32               if numero == adivinar {
33                     fmt.Println("HAS ADIVINADO")
34               }
35               if numero < adivinar {
36                     if distancia(numero, adivinar) {
37                           fmt.Println("Por debajo, estas muy
    cerca de adivinar el numero")
38                     } else {
39                     fmt.Println("Muy por debajo, estas muy
    lejos de adivinar el numero")
40                     }
41               }
42               if numero > adivinar {
43                    if distancia(numero, adivinar) {
44                         fmt.Println("Por encima, estas muy
    cerca de adivinar el numero")
45               } else {
46                    fmt.Println("Muy por encima, estas muy lejos
    de adivinar el numero")
47               }
48         }
49     }
50  }
```

Como se puede observar, se ve algo más complejo la estructura condicional, integrando el llamado de la función *distancia (...)*, anidaciones entre sentencias y condicionales de segunda opción.

El objetivo del cambio es volver al programa más dinámico cubriendo pequeños aspectos en cuanto a la interacción con el usuario se refiere. Tomando como base el

código anterior, se cubren situaciones para números entre un rango de uno a cien; Cabe resaltar que el rango definido se compone a partir de un modelo *if-elseif*-else, un derivado del condicional que permite evaluar situaciones al momento de ejecutar un campo *else*. Continuando, se determinan valores que sean mayores, menores o iguales al dato que se quiere adivinar, estos como segunda opción cuando se cumple la condición del intervalo propuesto anteriormente; Al cumplirse alguna, se debe verificar el resultado devuelto por la función *distancia(...)*, sabiendo de que su intención es la de determinar si el numero ingresado por el usuario está cerca o no del dato para adivinarla función es de tipo *bool* por lo que para valores *true* ejecuta lo almacenado el campo if, mientras que para casos *false* ejecuta lo almacenado en el campo *else*.

Nota: La función distancia() maneja números enteros, por ende, el rango de estos será de desde expresiones con signo negativo (-) hasta expresiones con signo positivo (+), por esta razón se formula el cálculo de la función para valores entre -10 y 10, indicados en el condicional de la construcción de distancia().

Al realizar algunas pruebas se obtienen los siguientes resultados:

Para números menores:

```
>  Digite un numero: 20
   Por debajo, estas muy cerca de adivinar el numero
```

Para números mayores:

```
Digite un numero: 50
Muy por encima, estas muy lejos de adivinar el numero
```

Cuando el número es igual al dato almacenado:

```
Digite un numero: 30
HAS ADIVINADO
```

Para datos fuera del intervalo (1,100):

```
Digite un numero: 0
El número debe ser mayor a uno
Digite un numero: 101
El número debe ser menor
```

Con todo lo anterior se da por terminado el problema propuesto al principio de esta sección.

Como se pudo observar, es muy amplio el funcionamiento de las estructuras condicionales permitiendo abstraer situaciones de la vida real al lenguaje de código de una manera más sencilla.

4.4 Condicional Switch

Como ya se mencionó en *Tipos de estructuras*, la estructura *switch* permite comparar múltiples o situaciones o valores que una variable pueda tomar. Su representación en Go es la siguiente:

```
…   …
4   var numero int
5
6       fmt.Print("Digite un numero: ")
7       fmt.Scanf("%d", &numero)
8
9       switch numero {
10      case 1:
11          fmt.Println("Uno")
12      case 2:
13          fmt.Println("Dos")
14      default:
15          fmt.Println("es otro número")
16      }
```

Se pregunta por la entrada de un número, en donde dependiendo de su valor (siendo posibles 1 y dos) se muestra su nombre en pantalla.

El desempeño de switch es mucho más simple que otros lenguajes de programación. Se compone de casos posibles, y un caso base, oportuno para cuando ninguna de las posibilidades definidas es ejecutada. No maneja estrictamente llaves y puntos de quiebre *break*, ya que usa como delimitadores de campos el inicio de un nuevo *case*.

El uso de esta estructura puede asimilarse con los condicionales *if* de muchas posibilidades (if, if-elseif). En comparación, se permite una estética visual y lógica mucho más definida y organizada.

4.4.1 Otros aspectos de switch

Se tiene el siguiente código:

```
…   …
4   var numero int
5
6       fmt.Print("Digite un numero: ")
7       fmt.Scanf("%d", &numero)
8
9   switch numero {
10      case 1, 3, 5, 7, 9:
11          fmt.Println("Es un numero impar")
12      case 2, 4, 6, 8, 10:
13          fmt.Println("Es un numero par")
14      default:
15          fmt.Println("Fuera de rango ")
16      }
```

Se determina si un numero ingresado es para o impar. Obsérvese que solo hay dos *case*, cada uno con los casos posibles, separados por comas. Con switch se permiten múltiples valores dentro de un mismo *case*, acción que puede proponer un funcionamiento mucho más eficiente ahorrando líneas de código que describan lo mismo.

Junto con esto, otra "otra acción rápida" (por llamarlo de alguna manera) es la posibilidad de inicializar valores en la condición de la estructura, por ejemplo:

```
…   …
4   switch i := numero; i {
5       case 1, 3, 5, 7, 9:
6           fmt.Println("Es un número impar")
7       case 2, 4, 6, 8, 10:
8           fmt.Println("Es un numero par")
9       default:
10          fmt.Println("Fuera de rango ")
11      }
```

Los valores ingresados por *número* se almacenan en *i*, que posteriormente es usada por switch para verificar cada caso posible.

Nota: *Los condicionales switch, así como la creación de cada case no necesitan del uso de paréntesis, más si se es necesario el encerrar el bloque de instrucciones con {} tanto al iniciar el switch como al iniciar un case.*

4.4.2 Fall through

Cada vez que se ejecuta la opción correcta, la acción del switch termina, por lo que ningún otro caso será ejecutado, sin embargo, existen algunos comandos que pueden priorizar las respuestas, como lo es *fallthrough* y break.

Para la primera expresión se tiene el siguiente código:

```
…   …
4   i := numero
5
6       switch {
7       case i >= 5:
8           fmt.Println("Mayor o igual a 5")
9       case i <= 10:
10          fmt.Println("Mayor o igual a 10")
11      default:
12          fmt.Println("Fuera de rango ")
13      }
```

Se verifica si un número es mayor o igual a cinco, mayor o igual diez o, por último, fuera de rango.

Nótese que el switch está por sí solo si una variable expresada, esto quiere decir que el resultado de la estructura dependerá de las condiciones descritas en cada *case*, como se muestra en las líneas siguientes.

Al momento de ejecutar el programa con el número seis se obtiene lo siguiente:

```
>   Digite un numero: 6
    Mayor o igual a 5
```

Se concluye la respuesta entrando en el primer *case*, sin embargo, el siguiente caso también es posible. Es aquí donde el comando *fallthrough* tiene su papel, indicando que tome la siguiente posibilidad sin importar si se cumple o no. Por ejemplo:

```
…   …
4   i := numero
5
6         switch {
7         case i >= 5:
8               fmt.Println("Mayor o igual a 5")
9               fallthrough
10        case i <= 10:
11              fmt.Println("Mayor o igual a 10")
12        case i < 50:
13              fmt.Println("Muy grande")
14        default:
15              fmt.Println("Fuera de rango ")
16        }
```

Al ejecutarlo se obtiene:

```
>   Digite un numero: 6
    Mayor o igual a 5
    Mayor o igual a 10
```

Como se puede observar se utiliza el comando para utilizar la condición siguiente a la actual, mostrando en pantalla dos resultados posibles aun así la condición evalué falso; Nótese también que de las tres posibilidades solo se pueden abarcar dos que se encuentren continuas, por lo que para mantener un funcionamiento de desplazamiento para todos los casos es necesario llamar al comando *falltrough* por cada *case* propuesto.

Esta herramienta puede mejorar los resultados analizados por un programa, sin embargo, es responsabilidad de quien la ejecute el analizar los casos que la necesiten, ya que de esto dependerá el flujo y la interpretación de datos.

4.4.3 Break

Como caso contrario, se puede utilizar el comando *break* para romper el ciclo de condición en un punto dado. Siguiendo con el ejemplo anterior:

```
…   …
4   switch {
5         case i <= 5:
6               fmt.Println("Mayor o igual a 5")
7         case i >= 10:
8               fmt.Println("Mayor o igual a 10")
9               break
10              fmt.Println("Muy grande")
11        default:
12              fmt.Println("Fuera de rango ")
13        }
```

Al ejecutar el código se obtiene:

```
>   Digite un numero: 11
    Mayor o igual a 10
```

Existen dos mensajes que deberían de aparecer en pantalla, pero a causa del break que se sitúa entre estos dos, el mensaje mostrado será siempre el código anterior a este.

Como ya se mencionó, Go delimita cada posibilidad al inicio de un nuevo *case* por lo que el uso del break es muy poco, además, es usado propiamente para estructuras iterativas las cuales se verán más adelante.

4.5 Ejercicios de repaso (estructuras de condición)

1.Un cliente ordena una cierta cantidad de brochas de cerda, rodillos y sellador; las brochas de cerda tienen un 20% de descuento y los rodillos un 15% de descuento. Los datos que se tienen por cada tipo de articulo son: La cantidad pedida y el precio unitario. Además, si se paga de contado todo tiene un descuento del 7%. Elaborar un programa que permita visualizar el precio total de una cierta orden de n cantidad productos, tanto para pago de contado como pago a crédito.

2.Realice un programa que calcule el sueldo que le corresponde al trabajador de una empresa que paga 60.000 dólares anuales. El programa debe realizar los cálculos en función de los siguientes criterios

a.Si lleva más de 10 años en la empresa se le aplica un aumento del 10%.
b.Si lleva menos de 10 años, pero más que 5 se le aplica un aumento del 7%.
c.Si lleva menos de 5 años, pero más que 3 se le aplica un aumento del 5%.
d.Si lleva menos de 3 años se le aplica un aumento del 3%.

3.En la tienda MercaFacil, el impuesto pagado por la compra de articulos se calcula de la siguiente manera:

-Los primeros $30.000 pesos no cobran impuesto
-Los siguientes $30.000 pesos tienen un 30% más de impuesto y el resto un 40% más de impuesto.
-Si el costo de la venta total es mayor a $85.000 pesos, entonces solo se cobra el 50% sobre el total de la venta.

Generar un programa que, ingresando el valor total de la compra, permita calcular el resultado total de la venta luego de aplicar impuestos.

4.El jefe del departamento de construcción de la constructora FERIESPACIOS desea desarrollar un programa para sus empleados que calcule el sueldo total para cada uno según el número de horas trabajadas. Para ello se tienen en cuenta los siguientes criterios:

-Sí el número de horas trabajadas es mayor a 40, el excedente de las 40 horas se paga al doble del valor de la hora. En caso contrario se paga la hora al valor normal.
-Si las horas exceden las 50 horas trabajadas el excedente se paga al triple del valor de la hora y se les descontará un impuesto del 12% del total del sueldo ganado.

Realizar un programa que, pidiendo el nombre del trabajador y el número de horas trabajadas muestre en pantalla el nombre del trabajador y el cálculo del sueldo total ganado. Parta del hecho que la hora trabajada tiene un valor de $8.500 pesos.

5.En un curso de programación II se hacen cuatro pruebas de las cuales el profesor obtiene 4 notas, cada una de las cuales está entre 0 y 5. En vista de las muy buenas notas que obtuvieron los estudiantes del grupo 35, el profesor decide que la nota final no será el promedio aritmético de las cuatro notas, sino que hará lo siguiente con las notas de cada uno de sus estudiantes:

-Eliminará la menor de las 4 notas
-La mayor nota tendrá un porcentaje del 50%
-Cada una de las dos notas restantes tendrá un porcentaje del 25%

De esta forma, la nota final de un estudiante que obtuvo las notas 3.0, 2.0, 4,5 y 3,2, será 3.8. Haga un programa que lea las 4 notas de un estudiante y calcule y escriba su nota definitiva.

6.Una compañía inmobiliaria requiere calcular fácilmente el precio de unos lotes que ha ofrecido para la venta. Para ello la inmobiliaria ofrece lotes desde 70 metros cuadrados hasta lotes de 900 metros cuadrados. El plan de descuentos es el siguiente: Si el lote es mayor a 400 metros cuadrados el descuento será del 25%, si tiene entre 400 y 700 metros cuadrados el descuento será del 17 % y para los lotes con área entre 700 y 900 metros el descuento es del 10 %. Para efectos del cálculo del área, se conoce el largo, el ancho de cada lote y el precio por metro cuadrado.

7.Cree un programa que permita calcular el numero de la secuencia Fibonacci correspondiente a un número que indique su posición, por ejemplo: si se ingresa el número 6 , el resultado deberá ser 8 ya que:

Posición	0	1	2	3	4	5	6
Sucesión de Fibonacci	0	1	1	2	3	5	8

8.Se necesita escribir un algoritmo para convertir un valor de temperatura de la escala Celsius a otras escalas de temperatura (Farenheit y Kelvin). El algoritmo debe solicitar al usuario un valor de temperatura, y la escala a la cual se quiere convertir dicho valor y debe reportar como resultado el valor de temperatura en la nueva escala.

4.6 Iteradores

El uso de los iteradores en Go es similar a C con la diferencia de que existe unificación entre estructuras como el *for* y el *while*. Debido a ello, se presentan tres tipos de modelos para la creación de ciclos *for* en Go:

```
//Como un ciclo for en C
for <variable (init)>; <condición>; <post> {Operaciones}

//Como un while en C
for condición{<operaciones>}

//Como un ciclo for infinito en C for(;;)
for{Operaciones}
```

Cabe notar que el primero, al ser el más general denota las tres fases de un ciclo for:

-Init: Cuando se declara la variable que funcionará como iterador solamente hasta cumplir la condición definida en la siguiente fase. Esta siempre es ejecutada al momento de entrar en una estructura *for*.

-Condition: Aquí se define la condición para la cual el ciclo *for* funcionará. Este será el punto final que indica el momento en el cual los ciclos o iteraciones dadas por el *for* se detienen (únicamente cuando la condición definida es falsa). Esta fase se ejecuta en cada ciclo desarrollado.

-Post: Esta fase determina la forma en como la variable definida en init incrementa, decrementa o avanza con respecto a la condición dada anteriormente. Esta fase se ejecuta en cada ciclo desarrollado.

A continuación, se muestran ejemplos de algunas de las formas de definir un ciclo *for*.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       for i, j := 0, 0; i < 5; i, j = i+1, j+1 {
8            fmt.Println(i, j)
9       }
10
11 }
```

Se crea un iterador *for* que utilizara dos variables *i* y *j*. Nótese que la creación de más variables se hace de la misma manera que por fuera de este, a través de comas y especificando por grupos el tipo de dato a manejar (de igual manera sucede con la estructura ':='). La condición puede contener múltiples parámetros que también incluyan múltiples variables, acompañados de expresiones aritméticas, lógicas y de comparación. Por último, el aumento de los valores de las variables definidas se debe de hacer por separado, indicando cómo será el cambio tanto para *i* como *j*.

Se tienen otras maneras de representar lo anterior como las siguientes:

```
…  …
4   i := 0
5
6       for ; i < 5; i++ {
7            fmt.Println(i)
8       }
9
10      fmt.Print(i)
```

Esta forma permite crear y modificar una variable para todo el ambiente local, es decir, *i* podrá ser usada en todo el entorno fuera del *for*, además de que su valor inicial cambiará ya que se verá afectado por el incremento del iterador.

Para mayor comprensión, obsérvese el resultado de la ejecución del código anterior:

```
> 0
  1
  2
  3
  4
  5
```

Nota: En el ejemplo inicial se puede intentar imprimir en pantalla el valor de la variable *i* por fuera del iterador, para luego compararlo con el ejemplo anterior. Encontrará un error al momento de la ejecución.

Como se indicó al inicio de esta sección, las otras formas de escritura para el *for* hacen referencia a los ya mencionados *while*, como a ciclos infinitos. En código trabajarían de la siguiente manera:

4.6.1 While

```
…  …
4  for i < 5 {
5          fmt.Println(i)
6          i++
7      }
```

El único cambio significativo es la simplificación de los tres campos en solo uno, que es la condición que determinara el número de ejecuciones o ciclos a realizar, y la variable que recorre iteración por iteración estando situada dentro del conjunto de *operaciones*.

4.6.2 For infinito

Para el caso de manejar ciclos sin fin se maneja lo siguiente:

```
for {
        <bloque de instrucciones>
    }
```

Cuando se maneja este tipo de iteración, lo más común es mantener una condición que permita limitar el número de ciclos generados evitando desbordar el programa. Esta condición puede ir definida dentro del conjunto de *operaciones*, permitiendo determinar en cada ciclo alcanzado si se debe continuar o no.

4.6.3 Break y continúe

Mencionado ya en la sección anterior, el *break* es un elemento mayormente usado por estructuras de repetición la cual ayuda a finalizar una ejecución en un momento dado. De igual forma se establece el *continue*, siendo el contrapuesto al anterior. En este caso esta expresión permite pasar a la iteración siguiente sin ejecutar el conjunto de operaciones.

A manera de ejemplo se tiene lo siguiente:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       i := 0
8
9       for ; i < 5; i++ {
10           fmt.Println(i)
11           if i == 2 {
12                break
13           }
14      }
15       fmt.Println()
16      i = 0
17
18      for ; i < 5; i++ {
19           if i == 2 {
20                continue
21           }
22           fmt.Println(i)
23      }
24 }
```

Se proponen dos iteradores *for*, el primero manejando la expresión *break* y el segundo con la expresión *continue*.

Si se analiza cada iteración, se observa que cada que el número de la variable *i* no sea igual a dos, este se mostrará en pantalla y no se ejecutará las operaciones del condicional. Caso contrario cuando la variable toma el valor anterior, entrando a ejecutar el elemento break.

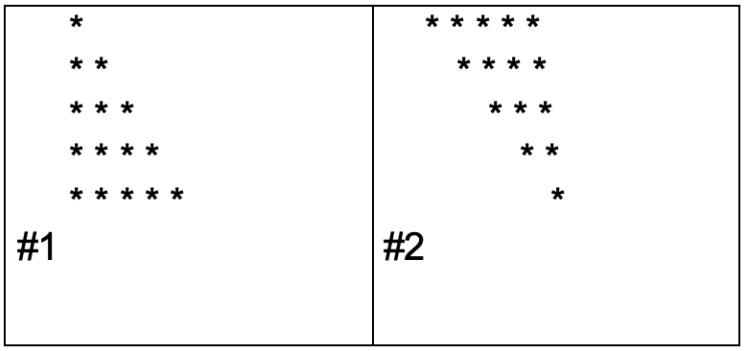
A comparación de la siguiente estructura, se evalúa primero si el valor *i* es igual a dos, en donde un caso verdadero genera la acción de pasar a la siguiente a iteración del *for*. Caso falso se mostrarán todos los elementos en pantalla.

Lo anterior se ejemplifica a partir del resultado de la ejecución:

```
> 0
  1
  2

  0
  1
  3
  4
```

4.7 Ejemplos

Teniendo en cuenta las estructuras vistas anteriormente, se propone formar las siguientes figuras:



4.7.1 FIgura #1

•Conceptualización y categorización

La situación que se pide analizar propone la formación de figuras como un triángulo y un triángulo invertido usando asteriscos y saltos de línea.

Nótese que en la figura #1 las filas se completan de forma ascendente y consecutiva, es decir, la primera fila va con un asterisco, la segunda con dos, la tercera con tres y así repetidamente hasta que se alcance un tope proporcionado. Con ello se puede definir su desarrollo a partir de estructuras de iteración.

De lo anterior se propone el siguiente análisis:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*				
	2	*	*			
	3	*	*	*		
	4	*	*	*	*	
	5	*	*	*	*	*

Para crear la figura usando una estructura iterativa se debe conocer las variables a relacionar y saber cómo será el comportamiento general del programa. Al iniciar se debe proponer un número que será el límite al cual llegará la iteración (para este caso la figura será de cinco niveles). Se pueden manejar dos variables *i* y *j*, las cuales representan las filas y columnas respectivamente. Estas aumentaran desde 0 (siendo la posición inicial) hasta que las dos tomen el valor límite.

Como se mencionó anteriormente, el llenado de las filas se hacía consecutivamente, además, el número de asteriscos corresponden con el numero tanto de la fila como de la columna.

•Desarrollo

Se puede usar un ciclo de iteración *for* para el recorrido de filas y otro para el de columnas, aunque la diferencia es que al momento de entrar en una fila se debe recorrer instantáneamente las columnas. Ello podría realizarse a través de ciclos anidados, como el siguiente:

```
for i := 0; i <= h; i++ {
   for j := 0; j < i; j++ {
     fmt.Print("*")
   }
  fmt.Println()
}
```

Nota: Se debe tener en cuenta la importancia de identificar el valor de inicialización para cada variable dentro del for, a la vez que la cantidad aumentada por ciclo.
En el ejemplo se inicia desde cero hasta el numero limite, sin embargo, puede intentarse cambiar los valores y condiciones para obtener una figura completamente distinta.

Obsérvese que las columnas (variable *j*) crean sus ciclos tomando como limite el valor de la fila (variable *i*). Esto permite que el número de asteriscos sea semejante al número de la fila y columna, como se mencionó anteriormente.

Cada vez que se termina la ejecución del *for* interno, se crea un salto de línea con *fmt.println()* para describir los distintos niveles de la figura, los cuales son descritos por la variable *h* y a la vez son el límite de iteración para el *for* más externo, encargado de recorrer las filas.

Como último detalle, para volver más agradable el uso de este código, se puede promover el ingreso de datos desde el teclado, usando la variable *h* como almacén de información.

El código final y su ejecución serían los siguientes:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       h := 0
8       fmt.Print("Altura: ")
9       fmt.Scanf("%d", &h)
10
11 for i := 0; i <= h; i++ {
12      for j := 0; j < i; j++ {
13           fmt.Print("*")
14      }
15      fmt.Println()
16  }
17 }
```

```
>  *
   **
   ***
   ****
   *****
```

4.7.2 FIgura #2

- Conceptualización y categorización

Siguiendo el esquema del ejemplo anterior, se debe crear un triángulo con sentido inverso. Se observa que el número de asteriscos va disminuyendo conforme el número de filas y columnas aumenta, además, se agregan espacios por cada paso de línea.

Se puede utilizar la construcción anterior, cambiar ciertos parámetros como el valor de inicialización para *i* y *j*, poner nuevos límites y relacionar el aumento de espacios con respecto al descenso de asteriscos.

- Desarrollo

Con el análisis anterior, se puede partir de lo siguiente:

```
…  …
4  for i := h; i >= 0; i-- {
5       for j := 0; j < i; j++ {
6            fmt.Print("*")
7       }
8       fmt.Println()
9   }
```

Al ejecutarlo se genera la siguiente figura:

```
>  Altura: 5
   *****
   ****
   ***
   **
   *
```

En lugar de que las filas iniciaran en cero, estas se inicializan en el valor ingresado como límite (o niveles). Luego de que la iteración interna termina, el valor de *i* disminuye generando cada vez un asterisco menos por cada fila lo que resulta en un triángulo en forme descendente.

De lo anterior se puede encontrar más fácilmente una forma de generar los espacios para retornar la figura solicitada. Para ello se sitúa el siguiente análisis:

- Cuando el número de espacios es cero, situándose en la primera línea:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*	*	*	*	*
	2	*	*	*	*	
	3	*	*	*		
	4	*	*			
	5	*				

- Cuando el número de espacios es uno y se ejecuta un cambio de línea:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*	*	*	*	*
	2		*	*	*	*
	3	*	*	*		
	4	*	*			
	5	*				

- Cuando el número de espacios es dos y se ejecuta nuevamente un cambio de línea:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*	*	*	*	*
	2		*	*	*	*
	3			*	*	*
	4	*	*			
	5	*				

•Cuando el número de espacios es tres y se hace un cambio de línea:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*	*	*	*	*
	2		*	*	*	*
	3			*	*	*
	4				*	*
	5	*				

•Cuando el número de espacios es cuatro y se hace un cambio de línea:

	COLUMNAS					
FILAS	0	1	2	3	4	5
	1	*	*	*	*	*
	2		*	*	*	*
	3			*	*	*
	4				*	*
	5					*

Los espacios pueden formarse como una variable que va cambiando cíclicamente. Dichos espacios son generados momento justo después de ejecutar un cambio de línea. Por otro lado, el valor de los espacios es inversamente proporcional al número de

asteriscos, lo que puede servir como parámetro dentro de cada iteración, evaluando el número de asteriscos a incorporar.

De lo anterior, se puede obtener el siguiente código:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  h := 0
8       fmt.Print("Altura: ")
9       fmt.Scanf("%d", &h)
10
11 e := 0
12
13 for i := 0; i <= h; i++ {
14      for j := 0; j <= h; j++ {
15           if e > j {
16           fmt.Print(" ")
17           } else {
18           fmt.Print("*")
19           }
20       }
21       e++
22       fmt.Println()
23 }
24 }
```

Como se observó en el análisis, los espacios generados serán manejados en una variable que inicialmente toma el valor de cero, el cual ira cambiando cada nuevo ciclo proveniente del *for* externo (o cada nuevo salto de línea).

Las variables *i* y *j* tomaran como límite el valor ingresado por teclado, por ende aumentaran cada iteración. El condicional determina la relación indicada anteriormente, donde cada vez que el número de espacios sean mayores que el valor tomado por *j*, se imprima un espacio, mientras que los casos en donde el parámetro inicial sea falso (*j* sea mayor a *e*) se imprima un asterisco.

Como resultado se obtiene la figura #2:

```
> Altura: 4

****
 ***
  **
   *
```

4.8 Ejercicios de repaso (estructuras de repetición)

1.Crear un programa que reciba una serie de números y calcule la sumatorio de todos estos.

2.Escribir una función que calcule cuantos números pares hay comprendidos entre dos números límites (sin incluirlos).

3.Crear un programa que calcule si un número es primo o no.

4.Crear un programa que simule el comportamiento de un reloj digital, escribiendo en formato de horas, minutos y segundos, iniciando desde las 00:00:00 horas hasta las 23:59:59 horas

5.Hacer un programa que pida un número y muestre en pantalla su tabla de multiplicar.

6.Generar los primeros diez números perfectos.

Nota: Un Número perfecto es aquel que es igual a la suma de sus divisores, excluyéndose el propio número. Por ejemplo, el número 28 presenta 5 divisores menores y distintos de 28, que son: 1, 2, 4, 7 y 14. Al sumarlos da como resultado 28

7.Según historias cuentan que el inventor del juego de ajedrez, el rey que le solicitó hacerlo le preguntó cómo quería que este le pagase, a lo cual el inventor le contesto que ubicara un grano de trigo en el primer cuadro del tablero, en el segundo ubicara el doble del primero, en el tercero el doble del segundo y así sucesivamente hasta completar los 64 cuadrados.

Se debe crear un programa que muestre un tablero (Puede ser sin líneas) en donde se indique el número de granos de trigo debió colocar el rey por cada cuadro.

Ejemplo:

1	2	3	4	5	6
12	11	10	9	8	7
13	14	15	16	17	18
24	23	22	21	20	19
25	26	27	28	29	30
36	35	34	33	32	31
37	38	39	40	41	42

Nota: El ejemplo es ilustrativo, se debe basar en el tablero de ajedrez.

8.Realizar las siguientes figuras para *n* niveles:

<table>
<tr>
<td>

```
    *
   **
  ***
 ****
*****
```

#2

</td>
<td>

```
*
**
***
****
*****
```

#3

</td>
</tr>
<tr>
<td>

```
*****
****
***
**
*
```

#4

</td>
<td>

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

#5

</td>
</tr>
<tr>
<td>

```
*
*  *
*  *  *
*  *  *  *
*  *  *
*  *
*
```

#6

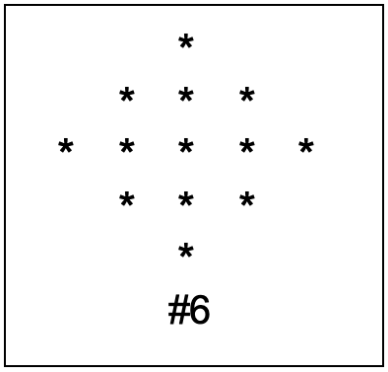
</td>
<td>

```
    *
   * *
  * * *
 * * * *
  * * *
   * *
    *
```

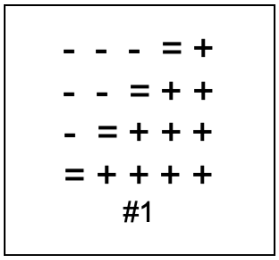
#7

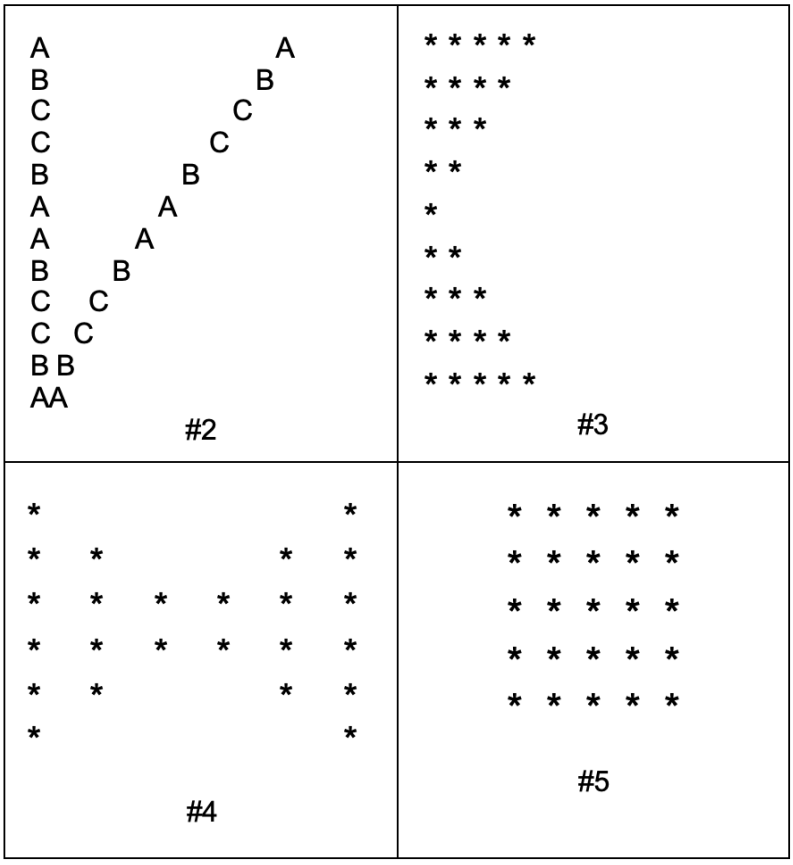
</td>
</tr>
</table>

9.Crear el siguiente cuadrado:

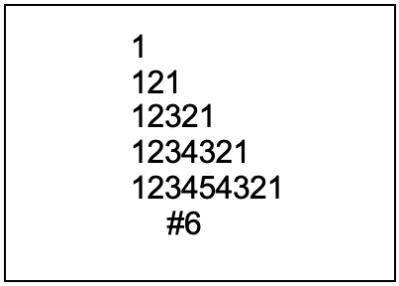


10.Crear el siguiente conjunto de figuras:





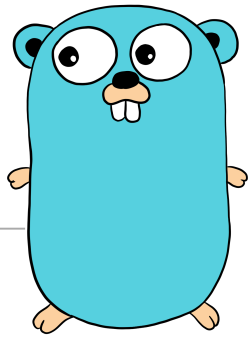
11. Leer número e imprimir una pirámide de dígitos:



12. Leer un número positivo e imprimir las sumas de números enteros positivos consecutivos que den el número introducido. Por ejemplo:

50 = 8 9 + 10 + II + 12
50 = II + 12 + 13 + 14

ESTRUCTURAS DE DATOS



En esta sección se explicarán algunas estructuras de datos, algunas muy conocidas en la programación en general, así como otras propias del lenguaje Go. Entre ellas se encuentran los *arreglos, mapas, slices, structs, listas, pilas, colas y deques.*

Arrays and slices

5.1 Arreglos

Los *arrays* (traducidos al español como *arreglos*) son estructuras de datos que almacenan información del mismo tipo.

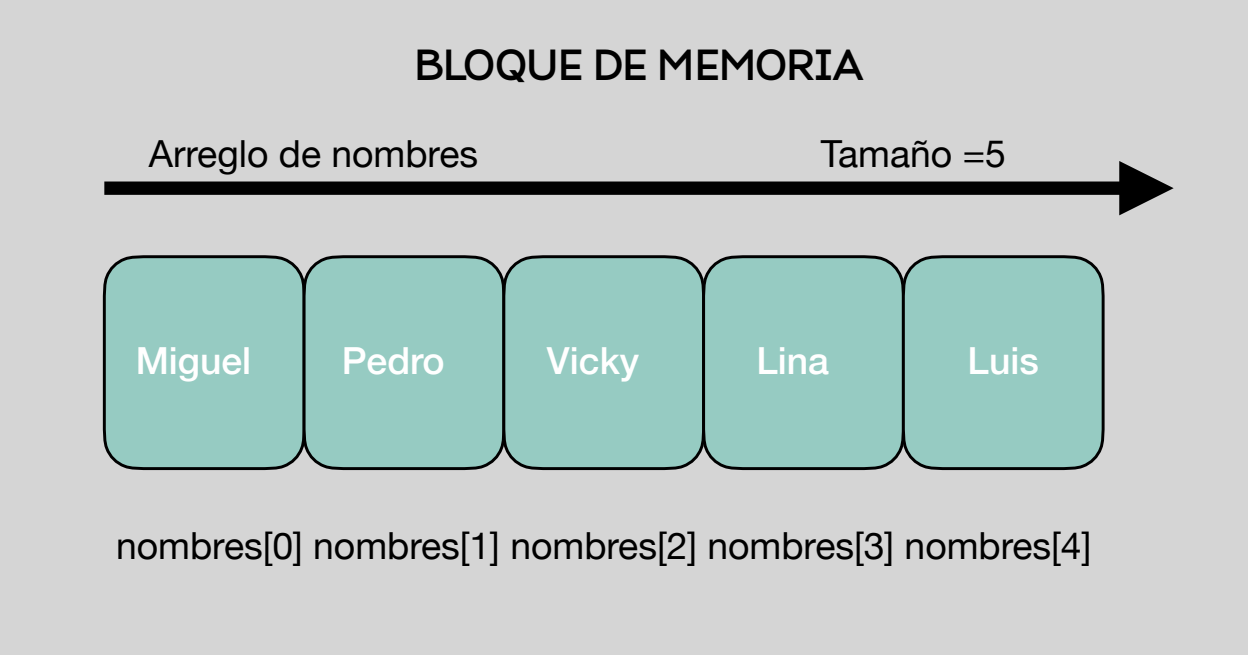


Ilustración 4. Representación gráfica de un arreglo

Se observa en la imagen un arreglo de tamaño cinco, que almacena valores en forma de cadena de texto (o strings). Observe que cada elemento es contiguo con el siguiente.

Los arreglos son creados en la memoria de un computador de forma consecutiva (un solo bloque de memoria subdividido en más bloques), lo que genera que cada dato se almacene como parte de una secuencia.

Las expresiones de *nombres[n]* indican la posición o índice en el cual se encuentra un elemento en específico. De esta manera se puede manipular un arreglo y la información que este almacene.

Nota: Los arreglos pueden entenderse como variables que almacenan valores de un mimo tipo.

5.2 Características de los arreglos

A comparación de otras estructuras, los arreglos disponen de las siguientes particularidades:

•Son estructuras que almacenan información, por ende, necesitaran de un tamaño especifico.
•Al usar la memoria de un computador, este tipo de estructuras indican la cantidad necesaria a usar para almacenar la información, es por ello por lo que se catalogan como estructuras estáticas.
•En Go, a comparación de otros lenguajes los arreglos funcionan como valores (no como punteros que almacenan la dirección del primer elemento almacenado en memoria, como en el lenguaje de C), por ende, en el momento de querer cambiar la información de forma indirecta, esta se tendrá que hacer con lo que se conoce en punteros como paso por referencia.

Nota: Cabe resaltar que, para esta última descripción, el tamaño no es re-asignable para un mismo arreglo.

5.3 Declaración e inicialización de arreglos

Para crear arreglos en la sintaxis de Go, se debe seguir la siguiente estructura:

var <nombre del arreglo> [<tamaño>]<tipo de dato>

Usando las declaraciones cortas quedaría:

<nombre del arreglo> := [<tamaño>]<tipo de dato>{<elementos>}

Puede usarse cualquier nombre como identificador. Su tamaño deberá ser un entero mayor o igual que cero, y su tipo de dato podrá ser cualquiera de los vistos en secciones anteriores.

La inicialización de arreglos puede hacerse de varias maneras:

- <nombre del arreglo> := [<tamaño>]<tipo de dato>{n_1, n_2, n_3, . . .,$n_{tamaño}$}
- <nombre del arreglo>[posición] = <valor>

Al igual que sucede con la asignación de valores por posición, pasa con el acceso a la información, que basta con solo especificar la posición en donde se encuentra el elemento buscado. Estos elementos, desde el punto de vista de posición son conocidos como índices. Comienzan desde el índice cero (siendo el primer elemento) hasta el

índice que es igual al tamaño del arreglo menos una posición, esto puesto que siempre al tener elementos se cuenta iniciando desde uno, pero como ya se mencionó, los índices comienzan desde cero.

<nombre del arreglo>[posición]

Nota: El tamaño de un arreglo es una parte propia de este tipo estructura por lo que en sus definiciones siempre se debe de indicar el tamaño. Existen otro tipo de estructuras que permiten dinamizar el uso de este tamaño, pero estas se verán más adelante.

Por otra parte, el tema de índices y su conteo desde una posición cero puede llevar a un problema de índices por fuera del límite si no se comprende la explicación anterior. Esto último es dar una posición no existente dentro de un arreglo.

Observe el siguiente ejemplo:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       var x [10]int
8       u := [...]float32{}
9       z := [5]string{"a", "b", "c", "d", "e"}
10      h := [...]float32{1.9, 2.0, 3.6, 8.0889759105910092804}
11
12      fmt.Println(x)
13      fmt.Println(u)
14      fmt.Println(z)
15      fmt.Println(h)
16 }
```

Se presentan varías de las formas de cómo se inicializa un arreglo. De estas últimas surge una sintaxis un poco más descriptiva, los puntos suspensivos "…" que sirven para indicar que el tamaño este guiado por el número de elementos asignados al vector entre las llaves {}.

Como resultado se obtiene:

```
>  [0 0 0 0 0 0 0 0 0 0]
   []
   [a b c d e]
   [1.9 2 3.6 8.088976]
```

Nota: Tanto los arreglos como los slices son usados por la función *len()* que calcula el tamaño de los mismos, es decir, el número de elementos asignables.

5.4 Manipulación de arreglos

Para ejemplificar la manipulación de arreglos, se propone el siguiente ejemplo:

•Realizar el algoritmo de ordenamiento de burbuja para vectores.

De lo anterior se obtiene lo siguiente:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  var arreglo [7]int
8
9       for i, j := 0, 7; i < 7; i++ {
10           arreglo[i] = j
11           j--
12      }
13      fmt.Printf("Arreglo desordenado %v\n",arreglo)
14
15      aux := 0
16
17      for i := 0; i < 6; i++ {
18           for j := i + 1; j < 7; j++ {
19                if array[i] > array[j] {
20                     aux = vector[i]
21                     arreglo [i] = arreglo[j]
22                     arreglo [j] = aux
23                }
24           }
25      }
26
27      fmt.Printf("Arreglo ordenado %v\n",arreglo)
28 }
```

Para cumplir con un ordenamiento (en este caso un arreglo en forma ascendente) se debe de tener un arreglo desordenado , y para este caso se optó por tomar una estructura que almacena números enteros de mayor a menor (esto para luego devolverlo en forma ascendente).

Observe que se puede utilizar una estructura iterativa para llenar cada posición con un dato especifico. Si se ejecutase lo creado hasta esa línea se obtiene lo siguiente:

```
> Arreglo desordenado [7 6 5 4 3 2 1]
```

Analizando más a fondo la inicialización del *arreglo[]*, mediante el uso de posiciones (tomando en cuenta que como límite se tienen siete), se accede a cada partición o segmento de este ubicado en la memoria, y luego se iguala al valor que va tomando la variable *j*. Un dato importante es el comportamiento de los valores tomados por *i*, la cual inicia desde 0 y recorre hasta la posición final menos uno del arreglo; aquí se destaca una característica primordial de los arreglos no mencionada antes, las indexaciones o posiciones inician desde cero; contar desde cero hasta seis es igual que contar desde uno hasta siete, se sigue cumpliendo con el criterio de siete espacios.

Posterior a la creación, se resuelve el problema utilizando el algoritmo prescrito de ordenamiento de burbuja. Este consiste en recorrer un arreglo mediante dos índices *i* y *j*. Cada uno tomara posiciones distintas tal que uno siempre vaya delante del otro, con esto se permite la comparación entre los elementos albergados en el arreglo, y de la cual se genera el ordenamiento.

5.5 Arreglos multidimensionales

Pueden describirse como *arreglos de arreglos*. De una mejor manera, pueden asimilarse con las dimensiones que componen al termino “realidad”. Cada dimensión compone un arreglo dentro de la expresión (o variable representativa).

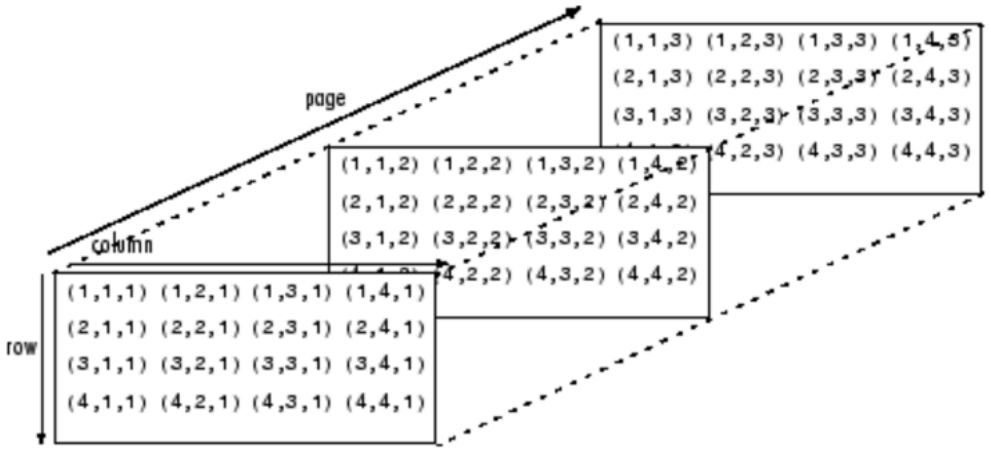


Ilustración 5.Arreglo multidimensional
Fuente: MathWorks

Los arreglos multidimensionales pueden tener n cantidad de arreglos representados. Comúnmente se manejan máximo hasta dos dimensiones (o arreglos bidimensionales).

Los arreglos bidimensionales son también conocidos como matrices. Pueden ser presentados como una tabla, relacionando aspectos como filas y columnas.

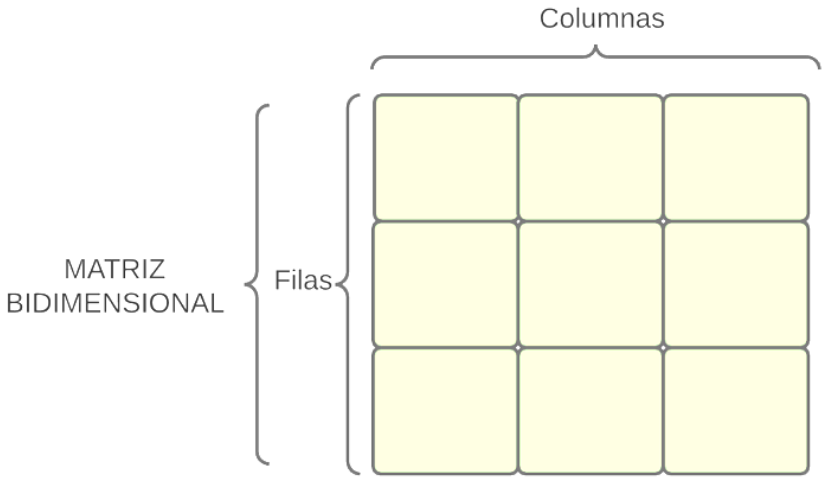


Ilustración 6. Arreglos bidimensionales

5.5.1 Creación e inicialización de arreglos bidimensionales

Para crear una matriz, se sigue la misma estructura que en arreglos unidimensionales con la diferencia que se agrega un nuevo campo [], indicando el nuevo arreglo a incorporar. Por ejemplo:

VAR <nombre del arreglo> [<tamaño Filas>] [<tamaño columnas>] <tipo de dato>

De igual forma sucede para la inicialización (o almacenamiento de datos):

<nombre del arreglo> := [][]<tipo de dato>{{n_1, n_2, n_3, . . .,n},{ n_1, n_2, n_3, . . .,n }}

5.5.2 Manipulación de matrices

Partiendo del siguiente código, se describe el funcionamiento de presentación de las matrices para Go:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6  var matriz [3][3]int
7       cont := 0
8       fmt.Println()
9       fmt.Printf("Matriz no inicializada -> %v\n", matriz)
10      for i := 0; i < 3; i++ {
11           for j := 0; j < 3; j++ {
12                matriz[i][j] = cont
13                cont++
14           }
15      }
16 }
```

Al ejecutarlo se produce:

```
>  Matriz no inicializada -> [[0 0 0] [0 0 0] [0 0 0]]
   Matriz inicializada -> [[0 1 2] [3 4 5] [6 7 8]]
```

Las matrices se manejan de igual forma que sucede con los arreglos unidimensionales, con la diferencia que ahora cada posición estará identificada por dos variables, *i* y *j*.

Note que luego de declarar la *matriz[][]*, al momento de mostrarla en pantalla, esta se carga con ceros en cada posición, al igual que sucedió con las variables no inicializadas.

Cada posición de la matriz se cambió utilizando un *for* anidado, donde uno recorre las filas (con los valores tomados por *i*) mientras que el otro recorre las columnas (valores tomados por *j*).

5.6 Slices

Go proporciona otras herramientas alternativas para el trabajo con arreglos, como lo son los *Slices* (traducido al español como pedazo o *rebanada*).

Los *slices* son estructuras de datos que *"describen la sección de un array"*. Almacenan referencias hacia los índices, más específicamente a un único elemento, lo que indica

que al tener este acceso se podrá conocer el resto de los elementos en el *array*. Es por esta razón que los slices **no almacenan información.**

Estas estructuras comparten otras características que hacen de estas herramientas más flexibles y eficientes en el manejo de datos, volviéndose también mucho más comunes en código Go-lang.

5.6.1 Características de los slices

A partir de su funcionamiento como un *puntero* que almacena la dirección de los datos (de un solo dato accede al resto de elementos en adelante) de un arreglo y que a su vez tiene la capacidad de modificar los datos de este último, se permite tener cierto nivel de flexibilidad en el volumen de datos que se manejan, además de procesos como la inserción y copia, tornando su uso mucho más eficiente.

Lo anterior puede representarse en lo siguiente:

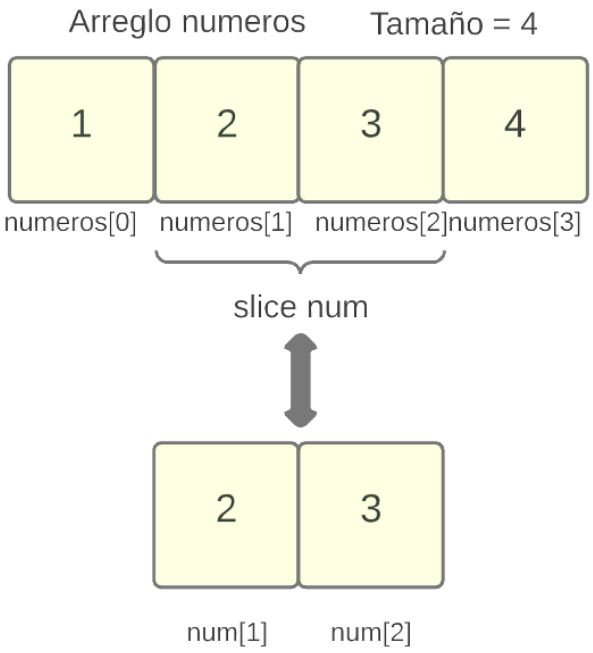
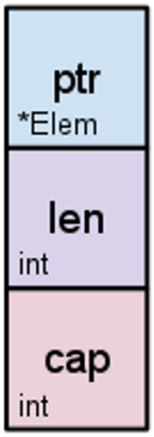


Ilustración 7. Slice referenciando un arreglo

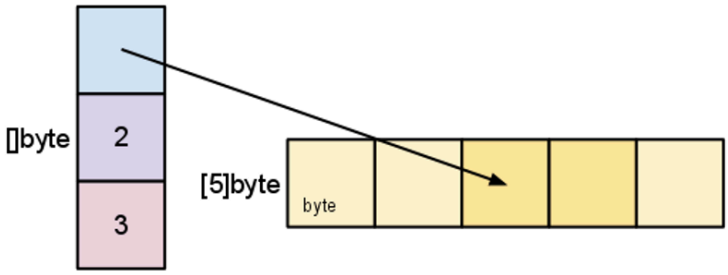
Nota: Los punteros se hablarán en capítulos posteriores.

Como se puede observar en la imagen, partiendo de un *arreglo* de cuatro elementos se crea un *slice* que referencia los datos que van desde la posición uno hasta la posición dos.

La siguiente imagen representa de manera más detallada la composición de un *slice*:



La



forma de usar esta función es:

Len(<nombre del slice o array>)

*Solo se aceptan variable del tipo slice o array

Ilustración 8. Composición interna de un slice
Fuente: Go Slices: usage and internals - The Go Blog (golang.org)

Como se puede observar en la imagen anterior, el *slice* se compone de tres campos, *ptr* es un dato de tipo puntero que almacena la referencia (o dirección) del primer elemento almacenado en un *array*; *len* y *cap* son variables de tipo entero que determinan lo siguiente:

-Len(): Esta función permite conocer el tamaño de un slice (y también de un array), siendo este tamaño igual al número de elementos almacenados en la estructura, que para un *slice* sería el número de elementos almacenados en el array.

Cap(): Esta función muestra la capacidad (cantidad de datos que puede ser almacenada) tanto para un slice como un para un *array* de guardar nuevos elementos. Para los *slices*, la capacidad total será igual a la capacidad del arreglo referenciado (más específicamente al elemento almacenado en la dirección que se apunta), y se debe resaltar que esta característica comenzará a contar el tamaño disponible dentro de dicho *array* desde el primer elemento, ósea, el elemento almacenado en la dirección que guarda el *slice*.

La forma de usar esta función es:

Cap(<nombre del slice o array>)

*Solo se aceptan variable del tipo *slice* o *array*

En la ilustración 8 se observa un cambio de coloración para identificar los elementos que serán indexados por el elemento referenciado en el *slice*, es decir, los elementos que compondrán al *slice*, y los elementos que no harán parte de este último.

Otro detalle a tener en cuenta es el tres que determina la capacidad del *slice*, que, recordando el concepto anterior, al estar determinado por el *array* referenciado y el hecho de que inicia a contar su capacidad desde el primer elemento referenciado por el *slice*, ese tres es generado por que hay un espacio restante que no se ha referenciado en el arreglo (no hace parte del slice) y además que el *slice* referencia en total a dos elementos. Este apartado se podrá ver con más facilidad en *5.6.4 Función append y visualización del tamaño y capacidad de un slice*.

5.6.2 Creación de slices

Para crear "rebanadas" se usan las siguientes estructuras:

•Cuando se toma como referencia un arreglo:

<nombre del slice> := <nombre del arreglo>[<posición inicial> : <posición final>]

Los campos de *posición inicial* y *posición final* indican el intervalo a representar por medio del slice.

Nota: Observe que en la definición de slice no se necesita indicar el tamaño ya que como se mencionó al inicio, estos datos permiten tamaños dinámicos.

•Cuando se crea un slice desde cero:

<nombre del slice> := <tipo de dato a referenciar> { n_1, n_2, n_3, . . .,$n_{tamaño}$}

•Usando la función predefinida *make*:

<nombre del slice> := make([]<tipo de datos>, <size>,<capacity>)

5.6.3 Ejemplos

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       z := [5]string{"a", "b", "c", "d", "e"}
8
9       z1 := z[:]
10      z2 := z[2:]
11      z3 := z[:3]
12      z4 := z[0 : 4]
13
14      fmt.Println(z1)
15      fmt.Println(z2)
16      fmt.Println(z3)
17      fmt.Println(z4)
18 }
```

En el ejemplo anterior se declara un arreglo de tamaño cinco con la variable *z*. A su vez este se inicializa con valores de tipo *string*. En las siguientes líneas se declaran con una sintaxis resumida *slices* que almacenan distintas secciones del arreglo.

-*z1* define un *slice* que contiene los elementos del arreglo *z* que van desde el inicio hasta el inidice final.
-*z2* define *slice* que contiene a los elementos del arreglo *z* que van desde el índice dos hasta el índice final
-*z3* define un *slice* que contiene a los elementos que van desde el índice inicial (índice cero, en arreglos y slices la numeración es con base a un índice cero) hasta el índice tres.
-z4 define un slice que contiene a los elementos que van desde el índice inicial hasta el último elemento.

Los resultados obtenidos son:

```
>  [a b c d e]
   [c d e]
   [a b c]
   [a b c d]
```

Ahora bien, observe que los *slices* que definen un índice final siempre toman todos los elementos antes de este sin incluirlo al final, como por ejemplo el *slice z4* que va desde el índice inicial (cero) hasta el índice final (donde se ubica el último elemento que, en este caso es en la posición cuatro), se muestra en pantalla solo hasta el índice tres, o el *slice z3* que va desde el índice inicial hasta el índice tres, más en pantalla se muestran todos los elementos que llegan hasta el índice dos. Esto es debido a que en la definición de los *slices,* el intervalo que se toma es abierto para el último elemento, es decir que se toma la penúltima posición. Sin embargo, esto se descarta en un único caso que es cuando el intervalo no se detalla en el inicio y en el fin, o alguno de estos, tomando todo por completo como sucede con *z1* o *z2*.

5.6.4 Función append, y visualización del tamaño y capacidad de un slice

La función *append()*, traducida al español como adjuntar, es una función predefinida por Go-lang que indica lo siguiente:

-*Append()*: Esta función es propia en el uso de *slices* ya que estas estructuras permiten modificar su tamaño en tiempo de ejecución. Permite la adición de elementos cumpliendo con un modelo de llamada a la función definido como:

Append(<nombre del slice>, <elemento a insertar en el slice>)

<elemento a insertar en el slice> -- [elemento 1, elemento 2, …] o [[]<tipo de datos del slice>{elemento 1, elemento 2, …}…] o [nombre del slice]…

NOTA: Los puntos suspensivos se deben de tener en cuenta para la definición de cada campo en la función *append()*.

*En el campo de *<elemento a insertar en el slice>* solo se acepta una cantidad de elementos aleatoria. Esto es porque la función *append()* maneja como segundo argumento un parámetro variadico (ver sección 3.6 Funciones variadicas). Por otra parte, Go también permite anexar slices de dos formas distintas, La primera se puede representar en la definición de la segunda opción de la estructura en este campo. Observe que se propone un *slice* definiendo los elementos que este contiene, y que al final se describen tres puntos suspensivos "…" que, al igual que en las funciones variadicas, indican una gran cantidad de argumentos los cuales serán separados uno por uno.
La segunda se puede representar con la tercera opción de la estructura de este campo. Con esta solo se utiliza el nombre del *slice* que se desea unir, en conjunto de los tres puntos suspensivos "…".

Al utilizar la función lo que sucede es que el *slice* abarca un nuevo elemento, aumentando su tamaño y disminuyendo su capacidad. Cuando se tiene que insertar información, pero la capacidad (tamaño disponible del arreglo al cual se está apuntando) no es suficiente, entonces se debe crear un nuevo espacio para todos los elementos (incluyendo al nuevo) con suficiente memoria para su almacenamiento, por lo que se recurre a copiar y transferir los elementos actuales más la adición a un nuevo *slice* con el tamaño suficiente. Esto último es, que, a partir de la creación de un nuevo arreglo en memoria con memoria suficiente, se produce un slice que almacena la dirección de un elemento en dicho arreglo, y que este último contiene tanto los elementos que ya estaban en el *slice* como los añadidos.

Debido a la creación de nuevas estructuras cada vez que no hay espacio disponible, estas operaciones pueden no traer afecciones cuando el volumen de datos es pequeño. Caso contrario ocurre cuando son volúmenes de información extensos, donde el coste computacional tanto de memoria como de ejecución se puede ver afectado

Nota: El aumento de tamaño de cada nuevo *slice* creado al usar la función *append()* se observa que es aproximadamente con base dos, es decir, un crecimiento donde n es la capacidad del *slice* (del array) antes de adjuntar un nuevo elemento.

Ejemplos

```
1  package main
2
3  import (
4       "fmt"
5  )
6
7  func main() {
8
9       z := [...]int{1, 2, 3, 4, 5}
10      z1 := z[:]
11      z2 := z[2:]
12      z3 := z[:3]
13      z4 := z[0:4]
14      z5 := z[1:4]
15
16      fmt.Println("Antes de usar append()")
17      fmt.Printf("slice z1 -> %v -> len %v || cap -> %v \n", z1, len(z1), cap(z1))
18      fmt.Printf("slice z2 -> %v -> len %v || cap -> %v \n", z2, len(z2), cap(z2))
19      fmt.Printf("slice z3 -> %v -> len %v || cap -> %v \n", z3, len(z3), cap(z3))
20      fmt.Printf("slice z4 -> %v -> len %v || cap -> %v \n", z4, len(z4), cap(z4))
21      fmt.Printf("slice z5 -> %v -> len %v || cap -> %v \n", z5, len(z5), cap(z5))
22      fmt.Printf("Array z -> %v  -> len %v || cap -> %v \n", z, len(z), cap(z))
23
24      z1 = append(z1, 6, 7, 8, 9, 10)
25      z2 = append(z2, 6, 7, 8, 9, 10, 11, 12, 13)
26      z3 = append(z3, []int{3, 4, 5, 6, 7, 8, 9}...)
27      z4 = append(z4, []int{5, 6, 7}...)
28      z5 = append(z5, z2...)
29
30      fmt.Println("Después de usar append()")
31      fmt.Printf("slice z1 -> %v -> len %v || cap -> %v \n", z1, len(z1), cap(z1))
32      fmt.Printf("slice z2 -> %v -> len %v || cap -> %v \n", z2, len(z2), cap(z2))
33      fmt.Printf("slice z3 -> %v -> len %v || cap -> %v \n", z3, len(z3), cap(z3))
34      fmt.Printf("slice z4 -> %v -> len %v || cap -> %v \n", z4, len(z4), cap(z4))
35      fmt.Printf("slice z5 -> %v -> len %v || cap -> %v \n", z5, len(z5), cap(z5))
36      fmt.Printf("Array z -> %v  -> len %v || cap -> %v \n", z, len(z), cap(z))
37
38 }
```

Para el código anterior se utilizan nuevas definiciones de los *slices z1, z2, z3, z4*, y añadiendo z5, todos dados a partir de un arreglo de números enteros *z*. Para este ejercicio, cada *slice* hará uso de las funciones de *append()* y *len()* y *cap()*.

Se observa una transición de valores antes y después de usar *append()*. En el momento en que se llama a esta función, cada *slice* inserta una cierta cantidad de números enteros determinada en el llamado a la función (y en donde se manejan múltiples formas).

Al ejecutar el código se observa lo siguiente:

```
>  Antes de usar append()
   slice z1 -> [1 2 3 4 5]          -> len 5 || cap -> 5
   slice z2 -> [3 4 5] -> len 3 || cap -> 3
   slice z3 -> [1 2 3] -> len 3 || cap -> 5
   slice z4 -> [1 2 3 4] -> len 4 || cap -> 5
   slice z5 -> [2 3 4] -> len 3 || cap -> 4
   Array z -> [1 2 3 4 5] -> len 5 || cap -> 5
   Despues de usar append()
   slice z1 -> [1 2 3 4 5 6 7 8 9 10] -> len 10 || cap -> 10
   slice z2 -> [3 4 5 6 7 8 9 10 11 12 13] -> len 11 || cap -> 12
   slice z3 -> [1 2 3 3 4 5 6 7 8 9] -> len 10 || cap -> 10
   slice z4 -> [1 2 3 4 5 6 7]  -> len 7 || cap -> 10
   slice z5 -> [2 3 4 3 4 5 6 7 8 9 10 11 12 13]  -> len 14 || cap -> 14
   Array z -> [1 2 3 4 5] -> len 5 || cap -> 5
```

Del funcionamiento se observa los cambios que sufre la capacidad de cada *slice* antes y después de usar *append()*. Como ya se mencionó anteriormente, la capacidad de un *slice* se determina por la capacidad que posea el arreglo referenciado, esto contando desde el primer elemento que se referencie en el *slice*. Esto último viéndolo desde el análisis de una parte del código anterior es:

•Tomando el *slice z2*, el cual coge los elementos almacenados en z que van desde el índice dos hasta el índice final, se obtiene un *slice* que referencia en total a tres elementos, y que, si se observa en el arreglo, se tomaron todos los elementos desde una posición arbitraria hasta el final, dejando sin capacidad de nuevos elementos por referenciar al slice. Es por lo anterior que la capacidad de *z2* es tres. Un caso contrario sucede con *z3* que, al estar definido desde el inicio hasta el índice tres, siendo esto realmente hasta una posición menos que la indicada, es decir, índice dos, el *slice* referencia en total a tres elementos, sin embargo en el arreglo todavía hay elementos disponibles (dos elementos) que pueden ser referenciados por *z3*, por lo que la capacidad de este último será tanto los elementos que ya hacen parte de este, como los disponibles en el array, generando así el número cinco tras sumar los tres elementos referenciados más los dos disponibles en *z*.

De otra forma, el funcionamiento de *append()* y el crecimiento de un *slice* puede observase en la siguiente imagen:

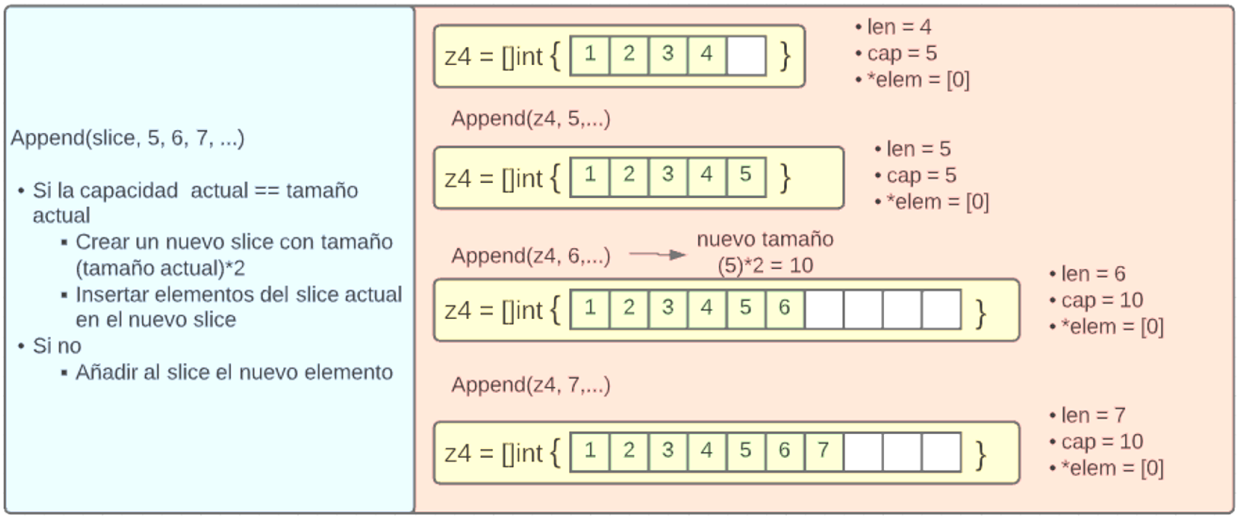


Ilustración 9. Representación del crecimiento de un slice al usar la función append()

En la ilustración se observa que hay un cambio de tamaño cada vez que se agrega un nuevo elemento al *slice*, y un cambio en la capacidad, y de forma *2n* cuando el tamaño del *slice* es igual a la capacidad. Mediante los cuadritos de color blanco se indica el espacio restante por ocupar, mientras que los coloreados y que contienen un número son el espacio ya ocupado del total de la capacidad.

Maps y struct

6.1 Maps

Los mapas son una estructura de datos integrada que proporciona el lenguaje de Go y que se basa en asociar dos tipos de elementos, unos conocidos como conocidos como llaves y otros conocidos como valores.

Los datos *maps* tienen una gran capacidad de almacenamiento en la que solo se necesitan expresiones a las cuales asignarles un valor como por ejemplo los registros de información con el campo nombres o un restaurante con la dirección de su domicilio.

6.2 Representación en memoria y funcionamiento interno de maps

Los maps generalmente se componen de datos conocidos como tablas hash (*hash tables* o tablas asociativas). Al igual que el concepto de mapas, las tablas hash crean relaciones entre ciertos elementos (denominados llaves) con otros (denominados valores), mediante una función dispersora conocida como función *hash* y un arreglo de tamaño arbitrario.

Cada tipo de dato puede ser representado de la misma manera:

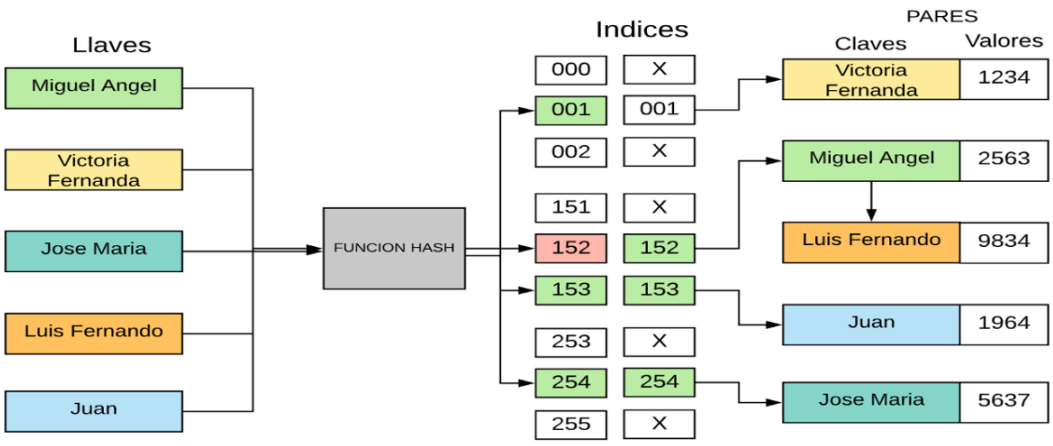


Ilustración 10. Funcionamiento de datos maps y tablas hash

Las funciones Hash (también conocidas como funciones resumen) son funciones que, utilizando un algoritmo matemático transforman un conjunto de datos en un código alfanumérico con una longitud fija. Esto último sin importar el tamaño del conjunto de datos siempre se entregará un código de igual longitud.

Gracias a la función hash se puede destacar la eficiencia como factor sobresaliente de los *maps* en comparación con otros mecanismos como los *arrays*. Lo anterior es dado a que la función calcula el índice adecuado para cada llave, por lo que al momento de requerir una información específica basta con solo identificar la llave necesaria, en lugar de recorrer y buscar por la información solicitada.

Nota: Se recomienda consultar más sobre las tablas hash como estructuras de datos independientes. Además, se recomienda también la conceptualización de los algoritmos usados por la función hash presentada en este apartado.

6.3 Creación de maps

Para la creación de *maps* se tiene las siguientes estructuras:

Opción 1

<nombre del dato> := map[<tipo de dato que define las llaves>] <tipo de dato que define las claves>] {
<llave #1> : <clave #1> ,
. . .
<llave #n> : <clave #1>,
}

Elementos como las comas o los puntos de asignación ':' son necesarios para la compilación de esta estructura. Estos funcionan como indicativos de un elemento *llave: valor*, así como el valor asignado a una llave respectivamente.

El campo donde se declaran las llaves, estas pueden ser de cualquier tipo de dato que permita la asignación de valores como lo son elementos enteros, punto flotante, complejos, cadenas, punteros, interfaces, arreglos y datos *struct*. Tipos de datos como los slices no se encuentran definidos en la comparación del operador de igualdad

Opción 2

Esta forma utiliza la función make(. . .). (ver apartado 3.2.1. Uso de make() y new() para la declaración de variables con nueva inicialización y asignación de memoria)

<Variable> := make(map[<tipo de dato que define el valor de la llave>] <tipo de dato que define el valor de la clave>]

6.4 Representación de un mapa

Los mapas pueden verse de la siguiente manera:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7       m := map[string]int{
8            "Maria":   123,
9            "Miguel":  456,
10           "Sara":    765,
11           "Jose":    835,
12           "Luis":    601,
13           "Ernesto": 419,
14      }
15
16      fmt.Printf("%v", m)
17 }
```

Se presenta dentro de una función main() una variable m, la cual se definirá como un mapa que almacena cadenas y datos de tipo entero. En su definición se puede encontrar una lista de nombres en donde cada uno tiene asignado un valor de tres cifras.

Al momento de imprimir el dato en pantalla se obtiene lo siguiente:

```
>    map[Ernesto:419 Jose:835 Luis:601 Maria:123 Miguel:456 Sara:765]
```

De esta manera se conocen los datos almacenados por un elemento creado a partir de maps. Otras funcionalidades que permite este tipo de datos serán analizadas más adelante.

6.5 Manipulación de maps en Go

Los mapas manejan operaciones de creación, inserción, eliminación, modificación y recorrido. Por ejemplo:

6.5.1 Ejercicio

Desarrollar una aplicación que nos permita crear un diccionario ingles/castellano (utilizar un dato tipo maps). La clave es la palabra en inglés y el valor es la palabra en castellano.

Crear las siguientes funciones:
1. Cargar el diccionario.
2. Listado completo del diccionario.
3. Ingresar por teclado una palabra en inglés y si existe en el diccionario mostrar su traducción.
4. ingresar por teclado una palabra en inglés o español y si existe en el diccionario, eliminar ya sea la traducción, la palabra en inglés, o la palabra en español.
5. ingresar por teclado una palabra en inglés, si existe en el diccionario modificarla

Desarrollo

```
1   package main
2
3   import "fmt"
4
5   func imprimir(diccionario map[string]string) {
6       fmt.Println(diccionario)
7   }
8
9   func cargar(diccionario map[string]string) {
10      var ingles string
11      var espanol string
12      var opcion string
13      for {
14          fmt.Print("Ingrese la palabra en Ingles:")
15          fmt.Scan(&ingles)
16          fmt.Print("Ingrese la palabra en Español:")
17          fmt.Scan(&espanol)
18          diccionario[ingles] = espanol
19          fmt.Print("Desea cargar otro producto[s/n]:")
```

```
20              fmt.Scan(&opcion)
21              if opcion == "n" {
22                   break
23              }
24         }
25   }
26
27   func borrar(diccionario map[string]string) {
28        var PalabraIngles string
29        fmt.Print("Ingrese la palabra (en ingles) a eliminar:")
30        fmt.Scan(&PalabraIngles)
31        _, existe := diccionario[PalabraIngles]
32        if existe {
33             delete(diccionario, PalabraIngles)
34             fmt.Println("Se eliminó el producto")
35             imprimir(diccionario)
36        } else {
37             fmt.Println("No existe")
38        }
39   }
40
41   func modificar(diccionario map[string]string) {
42
43        var PalabraIngles string
44        var TraduccionEspanol string
45        var PreTraduccionEspanol string
46        var aux string
47
48        fmt.Print("Ingrese la palabra (en ingles) a
     modificar:")
49        fmt.Scan(&PalabraIngles)
50        _, existe := diccionario[PalabraIngles]
```

```
51
52      if existe {
53              var op int
54              fmt.Println("Que desea modificar?")
55              fmt.Println("1. Palabra en Ingles")
56              fmt.Println("2. Traducción del Español")
57              fmt.Println("3. Modificar por una nueva
palabra")
58              fmt.Println("4. Salir")
59              fmt.Scan(&op)
60              switch op {
61              case 1:
62                      PreTraduccionEspanol =
diccionario[PalabraIngles]
63                      aux = PalabraIngles
64                      delete(diccionario, PalabraIngles)
65                      fmt.Println("Ingrese la nueva palabra en
Ingles")
66                      fmt.Scan(&PalabraIngles)
67                      diccionario[PalabraIngles] =
PreTraduccionEspanol
68                      fmt.Printf("La palabra %v ha sido cambiada
por %v \n", aux, PalabraIngles)
69                      imprimir(diccionario)
70               case 2:
71                        fmt.Println("Ingrese la nueva traducción
en Español")
72                        fmt.Scan(&TraduccionEspanol)
73                        aux = diccionario[PalabraIngles]
74                        diccionario[PalabraIngles] =
TraduccionEspanol
75                        fmt.Printf("La palabra %v ha sido
cambiada por %v\n", aux, PalabraIngles)
76                        imprimir(diccionario)
77
78               case 3:
79                      delete(diccionario, PalabraIngles)
```

```
80                  fmt.Println("Ingrese la nueva palabra en
    Ingles")
81                  fmt.Scan(&PalabraIngles)
82                  fmt.Println("Ingrese la nueva traducción
    en Español")
83                  fmt.Scan(&TraduccionEspanol)
84                  diccionario[PalabraIngles] =
    TraduccionEspanol
85                  fmt.Println("su nuevo diccionario es: ")
86                  imprimir(diccionario)
87              case 4:
88                  return
89
90              }
91      } else {
92              fmt.Println("No existe")
93      }
94  }
95
96  //Usar el for..range para recorrer un mapa
97
98  func recorrer(diccionario map[string]string) {
99       for clave, valor := range diccionario {
100           fmt.Println("Clave:", clave, " Valor:", valor)
101      }
102 }
103
104 func main() {
105
106 diccionario := make(map[string]string)
107
108      cargar(diccionario)
109      imprimir(diccionario)
110     borrar(diccionario)
```

```
111       modificar(diccionario)
1120      recorrer(diccionario)
113   }
```

Nota: Este ejemplo es influenciado de otros similares encontrados en la web

El programa divide su ejecución en varias funciones:

1-) Cargar el diccionario

```
 9  func cargar (diccionario map[string]string) {
10        var ingles string
11        var espanol string
12        var opcion string
13        for {
14              fmt.Print("Ingrese la palabra en Ingles:")
15              fmt.Scan(&ingles)
16              fmt.Print("Ingrese la palabra en Español:")
17              fmt.Scan(&espanol)
18              diccionario[ingles] = espanol
19              fmt.Print("Desea cargar otro producto[s/n]:")
20              fmt.Scan(&opcion)
21              if opcion == "n" {
22                    break
23              }
24         }
25  }
```

La función cargar es la encargada de almacenar cada palabra en inglés y su traducción al español en diccionario, que es el mapa que se está pasando como argumento. Nótese que el dato map se define con llaves valores del mismo tipo. Dentro del cuerpo de la función se definen variables de tipo string (ingles, español y opción) las cuales servirán de datos auxiliares en el ingreso de datos por teclado.

Dentro de un ciclo for infinito, el cual, recordando paginas atrás hace referencia a una estructura iterativa while, se producen los mensajes de texto para ingresar las palabras tanto en inglés como su traducción en español. Dicha información se almacenará en las variables string mencionadas anteriormente. Luego, se realiza la asignación de la información ingresada dentro del diccionario con diccionario[ingles] = español. Al final se pregunta si por cargar un nuevo producto, de lo cual si se responde s se itera nuevamente y si se responden, el ciclo for para.

2-) Listado completo del diccionario

-Imprimiendo el diccionario

```
5  func imprimir(diccionario map[string]string) {
6      fmt.Println(diccionario)
7  }
```

La función imprimir, como su nombre lo indica, muestra en pantalla toda la información almacenada en el diccionario.

-Iterando sobre el diccionario

```
98  func recorrer(diccionario map[string]string) {
99      for clave, valor := range diccionario {
100         fmt.Println("Clave:", clave, " Valor:", valor)
101     }
102 }
```

La función recorrer, al igual su antecesora recibe un diccionario de tipo string. Dentro del cuerpo de esta se utiliza un ciclo iterativo for en donde las variables que serán iteradas con múltiples elementos, clave y valor son asignadas con la información proporcionada por range(), un elemento de Go que permite reconocer el rango total de elementos dentro de un conjunto dado, y en este caso el rango calculado se indica por el número de elementos existentes en diccionario. Como instrucciones del ciclo se tiene el despliegue en pantalla de los mensajes clave y valor respectivamente.

Cabe resaltar que las variables a ser iteradas están tomando el orden de asignación de valores como llave-valor, lo que significa que clave tomará el rango de todas las llaves existentes en diccionario, mientras que valor, de manera similar tomará el rango de valores existentes en diccionario.

Como resultado de este proceso se tiene lo siguiente:

```
>  Ingrese la palabra en Ingles: jump
   Ingrese la palabra en Español: saltar
   Desea cargar otro producto[s/n]: s
   Ingrese la palabra en Ingles: run
   Ingrese la palabra en Español: correr
   Desea cargar otro producto[s/n]: n
   Clave: run Valor: correr
   Clave: jump  Valor: saltar
```

3-) Proceso de eliminación

```
27  func borrar(diccionario map[string]string) {
28        var PalabraIngles string
29        fmt.Print("Ingrese la palabra (en ingles) a eliminar:")
30        fmt.Scan(&PalabraIngles)
31        _, existe := diccionario[PalabraIngles]
32        if existe {
33              delete(diccionario, PalabraIngles)
34              fmt.Println("Se eliminó el producto")
35              imprimir(diccionario)
36        } else {
37              fmt.Println("No existe")
38        }
39  }
```

La función borrar se encarga de eliminar una palabra con su traducción, si esta se encuentra dentro del diccionario. Dentro del cuerpo de la función se crea una variable PalabraIngles de tipo string, la cual servirá para almacenar y posteriormente buscar la palabra a eliminar.
Verificación 'comma ok'

Luego de ingresar la información solicitada se crea un tipo de dato evaluador con existe el cual almacena la información guardada en la llave que proporciona PalabraIngles. Este tipo de funcionamiento en Go se conoce como notación "comma ok" y se encarga de verificar si algún dato map contiene dentro de sí una pareja llave-valor especifica. Por ejemplo:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  e := map[int]float32{
8            1: 10.0,
9            3: 11.5,
10           5: 12.7,
11           7: 18.3,
12       }
13
14       KeyVer, ValueVer := e[4]
15       //Una posicón que no existe
16       fmt.Println(KeyVer)
17       fmt.Println(ValueVer)
18
19       KeyVer1, ValueVer1 := e[3]
20       //Una posicón que si existe
21       fmt.Println(KeyVer1)
22       fmt.Println(ValueVer1)
23 }
```

El programa muestra como resultado:

```
>  0
   false
   11.5
   true
```

La estructura del verificador 'KeyVer, ValueVer' toma valores en el mismo orden que una pareja llave-valor, es decir, al momento de hacer la asignación de valores del mapa e[] con una llave aleatoria con dos variables y KeyVer y ValueVer, lo que se hace es asignar correspondiente al orden de las variables a usar, la llave analizada y un resultado booleano.
En los resultados del programa se observa que no existe una llave que tenga como elemento un 4 dentro del mapa analizado, por lo que termina mostrando como resultado un 0 (keyValue) y un false (ValuVer). Por otra parte, se verifica con nuevas

variables KeyVer1 y si en el mapa existe una llave con elemento 3, dando como resultado la existencia de una llave 3 con un valor booleano true.

Se puede notar que la variable decisiva sobre la existencia de información dentro de un mapa es aquella que almacene la información existente en el valor de una llave específica, por ello, el uso de esta verificación se hace comúnmente de formas más rápidas a nivel de memoria si se evita el uso de datos para almacenar una llave en estudio, por ejemplo

```
…   …
4   _, ok := e[4]
5       //Una posicón que no existe
6       fmt.Println(ok)
7
8       _, ok1 := e[3]
9       //Una posicón que si existe
10      fmt.Println(ok1)
…   …
```

Al igual que el programa anterior, se comprueba la existencia de una llave y su valor dentro del mapa e[].El guion bajo sirve como "variable fantasma" ya que le indica al programa al momento de compilación que dicha variable no será utilizada o tomada en cuenta.

Retomando el ejemplo

Luego de analizar el proceso que conlleva la verificación de una llave y su valor, se ejecuta un condicional en donde dependiendo del resultado arrojado por los verificadores previamente analizados, si se es verdadero, la palabra ingresa al inicio de la función será eliminando con la función prediseñada delete(), mientras que si es falso se dispone en pantalla el mensaje "No existe", lo que indica que la palabra ingresada no se encuentra en el diccionario.

4-) Proceso de modificación (palabra, traducción, pareja llave-valor)

```
41  func modificar(diccionario map[string]string) {
…   …}
```

La función modificar se encarga de cambiar la información presente en el diccionario, ya sea la palabra en ingles, su traducción al español, o las dos por completo.

Dentro del cuerpo de la función se usarán cuatro variables de tipo string, las cuales ayudarán a almacenar y reasignar la nueva información introducida. Luego, se introduce la palabra a la cual se le quiere hacer una modificación, y nuevamente se hace uso de la verificación "comma ok". Si la palabra se encuentra dentro del diccionario entonces se despliega en pantalla un menú de opciones y del cual se harán funciones distintas dependiendo del número a escoger.

La opción para escoger es almacenada en la variable op, que mediante una estructura switch evalúa los casos posibles a tomar (siendo 1, 2, 3 o 4). El funcionamiento para cada uno es el siguiente:

- **Case 1:** Este caso se encarga de modificar una palabra en inglés. Se almacena la traducción que se desea modificar en una variable PreTraduccionEspañol y una variable aux almacena la palabra en ingles que será modificada. Posteriormente se elimina la palabra en ingles seleccionada, para luego ingresar nuevos valores usando PalabraIngles y se hacen las nuevas asignaciones dentro del diccionario. Finalmente se muestra en pantalla el cambio realizado.

- **Case 2:** Este caso se encarga de modificar la traducción en español de una palabra proporcionada. Al igual que el anterior, el funcionamiento es similar ya que se utiliza una variable aux que almacena la palabra en inglés, mientras que se usa TraduccionEspanol para almacenar la nueva traducción. Finalmente, después de hacer las nuevas asignaciones se muestra un mensaje en pantalla indicando el cambio realizado.

- **Case 3:** Este caso se encarga de eliminar tanto la palabra en inglés como la traducción en español. Se utiliza la función prediseñada delete(), luego se ingresa la nueva información y se hace la nueva asignación de valores al mapa diccionario. Al final se hace una llamada a la función imprimir, La cual mostrará en pantalla los cambios realizados.

- **Case 4:** Este caso es el encargado de dar salida cuando ninguna de las opciones presentadas es la adecuada. Se utiliza un *return* para indicar la terminación de la función modificar.

6.6 Structs

Las "estructuras "son un tipo de dato que puede ser definido por cualquier persona, por ejemplo, un registro que indique los trabajadores. Estos tipos *integran/agrupan* una colección de elementos de uno o distintos tipos, y que representarán de forma general un solo concepto.

En otras palabras, las estructuras permiten representar algún elemento que cuente con ciertas características. También pueden ser asimiladas con el paradigma de

programación orientado a objetos en donde la representación de cualquier elemento se hace mediante su abstracción a los conceptos de clases, objetos y métodos.

Otro ejemplo trabajado desde las estructuras podría ser un *celular*. Cuenta con características como nombre (o referencia), capacidad de batería, almacenamiento interno, capacidad de RAM, megapíxeles de la cámara y definición de la pantalla, precio, entre otras.

6.7 Composición de estructuras

6.7.1 Declarando una estructura

Haciendo uso del concepto de estructuras, se mostrará a continuación la agrupación de las características que componen a un celular según el párrafo anterior.

Se debe conocer el formato de creación de estos datos, por ende, la manera de declarar todo tipo de struct es:

```
type <nombre de la estructura> struct {

    <elementos que componen la estructura
    (características del elemento a representar)>
    ...
}
```

Los *<elementos que componen la estructura>* pueden ser cualquiera de los tipos de datos permitidos en Go tales como enteros, flotantes, bytes, strings, booleanos, arreglos, slices. Por otra parte, se permiten anidar estructuras, así como funciones, lo cual hace referencia a funcionalidades que se verán más adelante en el capítulo.

En este campo, las formas de declaración de nuevas variables son las siguientes:

Nombre_del_campo <tipo de dato>

Por ejemplo:

```
nombre string
```

La declaración de la estructura del ejemplo propuesto se vería de la siguiente manera:

```
1   package main
2
3   import "fmt"
4
5   type Phone struct{
6         reference string
7         battery int
8         memory_capacity_ [2]int //[storage,RAM]
9         camera_definition string
10        price float32
11  }
12
13  func main(){
14  …
15  }
```

En esta definición se ha utilizado tipos de datos como *string* (cadenas de texto), *int* (números enteros), *float32* (números flotantes con alcance de 32 bits), y arreglos de números enteros ([] int) para la definición de características de un teléfono (*phone)* como referencia (*reference)*, batería (*battery)*, capacidad de memoria (*memory_capacity)*, definición de cámara (*camera_definition).* y precio (*price).*

6.7.2 Creando instancias de una estructura

Hasta este momento solo se ha creado el tipo de dato *Phone*, que de otra forma de verlo sería como tener una plantilla para la creación de nuevos datos de este tipo. A este proceso se le puede conocer como instanciamiento (similar al proceso con el paradigma de programación orientado a objetos), y se realiza de las siguientes maneras:

1. *<var> <Nombre de la instancia (variable)> <nombre de la estructura>*
2. *<Nombre de la instancia> := <nombre de la estructura> <{}>*

Partiendo del ejemplo de los celulares, las declaraciones en sus dos formas serían:

Creación de instancias #1

```
…    …
13   func main() {
```

```
14        var my_phone Phone
15  }
```

Creación de instancias #2

```
…   …
13  func main() {
14        my_phone := Phone{}
15  }
```

Las dos indican el mismo procedimiento, a diferencia que una es más explícita que otra en cuanto a sintaxis.

Nota: en el desarrollo del ejemplo se usará la forma #2

6.7.3 Inicializando valores

Para su inicialización, primero se debe tener en cuenta que, al igual que sucede con las instancias una vez creadas estas automáticamente son inicializadas con cero, pasa de igual manera con las estructuras, por ejemplo:

```
13  func main() {
14        var my_phone Phone
15        fmt.Print(my_phone)
16
17  }
```

```
13  func main() {
14        my_phone := Phone{}
15        fmt.Print(my_phone)
16
17  }
```

Los dos códigos anteriores dan como resultado:

```
>  { 0 [0 0]  0}
```

Ahora bien, la modificación de estos valores puede hacerse también de dos formas distintas:

Inicialización forma #1

```
13  func main() {
14
15      phone_capacity := [2]int{1, 2}
16
17      my_phone := Phone{
18          reference:         "iphone",
19          battery:           3500,
20          memory_capacity:   phone_capacity,
21          camera_definition: "18 Mpx",
22          price:             1500.5,
23      }
```

Esta representación utiliza los campos definidos en la estructura para la inicialización con nuevos valores en la instancia *my_phone*. Esta forma permite tener claro como están compuestos los campos de una estructura, siendo esto una característica muy útil al momento de manejar grandes volúmenes de información o en la realización de múltiples operaciones dentro de un mismo programa.

Nota: En el campo de memory_capacity se asigna una variable que almacena un arreglo de enteros, esto es únicamente por almacenar el valor de dicho campo en forma de variable independiente. Puede simplemente ser asignado como valor a dicho campo en la forma de arreglo []int{1,2}, sin la necesidad de usar una variable, como se verá más adelante.

Inicialización forma #2

```
>  my_phone := Phone{"iphone", 3500, phone_capacity , "18px",
   1500.5}
```

La forma anterior es más simplificada puesto que no es necesario nombrar los campos al que pertenece, únicamente basta con asignar el valor en la posición adecuada. Esta representación puede presentar problemas si no se conoce correctamente la disposición de los campos definidos en la estructura, como también si se quisiera agregar nuevos campos en la misma, por ejemplo:

```
…  …
4  my_phone := Phone{"18px", 3500, phone_capacity, "iphone", 1500.5}
5      fmt.Print(my_phone.reference)
```

```
>  18px
```

Como se puede observar, al llamar la referencia de la instancia *my_phone* se obtiene el nombre que anteriormente se definió como la cantidad de píxeles que posee la cámara del celular representada. También se tiene que en la inicialización de los campos de *my_phone*, al primer elemento se le asignan los valores que deben estar asociados en *camera_definition*, haciendo que el campo *reference* tome dicho valor. Este tipo de errores también pueden ocurrir al momento de querer agregar nuevos campos a la estructura, así como cambiar su orden de aparición, por ello, se recomienda utilizar la *forma #1* para la gran parte de los casos.

6.7.4 Acceso a los valores de una instancia estructurada

Habiendo ya inicializado una instancia de una estructura, se puede acceder a los valores de dicha instancia mediante la notación de punto y los campos de la misma. Por ejemplo:

```
1  package main
2
3  import "q"
4
5  type Phone struct {
6       reference         string
7       battery           int
8       memory_capacity   []int //[storage,RAM]
9       camera_definition string
10      price             float32
11 }
12
13 func main() {
14
15      my_phone := Phone{
16           reference:         "iphone",
17           battery:           3500,
18           memory_capacity:   []int{1, 2},
19           camera_definition: "18 Mpx",
20           price:             1500.5,
21      }
22
23      fmt.Println("Las características de los celulares ",
        my_phone.reference, " son:")
24      fmt.Println("Batería: ", my_phone.battery)
25      fmt.Println("Almacenamiento interno : ", my_phone.memory_capacity[0],
        "\n", "RAM: ", my_phone.memory_capacity[1])
26      fmt.Println("Definición de la cámara: ", my_phone.camera_definition)
27      fmt.Println("Precio: ", my_phone.price, "$")
28 }
```

La ejecución del código anterior es lo siguiente:

```
> Las características de los celulares iphone son:
  Batería:  3500
  Almacenamiento interno :  1
   RAM:  2
  Definición de la cámara:  18 Mpx
  Precio:  1500.5 $
```

Se pudo obtener un breve mensaje descriptivo de las cualidades de un celular dado por la estructura *Phone* con ciertos valores almacenados en una instancia *my_phone*.

Nótese que la forma de acceder a los valores almacenados en esta instancia es:

<nombre de la estructura>.<nombre del campo>

Esto indica que, partiendo de una instancia, la compilación se dirige a la composición de dicha instancia y que dependiendo del campo que sea llamado, se mostrará su valor almacenado en memoria.

6.8 punteros y estructuras

La relación entre estos dos campos puede ser de gran utilidad cuando se maneja bastante cantidad de información, como cuando se desea ser más eficiente en el uso de la información. Por ejemplo:

*partiendo de la estructura anterior

```
…    …
   4 func main() {
   5
   6 my_phone := Phone{
   7 reference:         "iphone",
   8 battery:           3500,
   9 memory_capacity:   []int{1, 2},
  10 camera_definition: "18 Mpx",
  11 price:             1500.5,
```

```
12 }
13
14 my_phone2 := my_phone
15
16 fmt.Println(my_phone, "estructura my_phone")
17 fmt.Println(my_phone2, "estructura my_phone2")
18 my_phone2.battery, my_phone2.price = 2700, 1000
19 fmt.Println("Luego de hacer cambios")
20 fmt.Println(my_phone2, "estructura my_phone2") // estructura de luego realizar un pequeño cambio
21 fmt.Println(my_phone, "estructura my_phone")
22
23 }
```

Se crea una nueva variable *my_phone2* la cual se inicializa con los valores almacenados en la instancia de la estructura *my_phone*. Hasta el momento, se puede pensar que las dos variables mencionadas allí funcionan como una instancia de estructura y que tienen exactamente los mismos valores. Con ello, se debe entrar en el análisis de cuánta independencia tiene cada variable con sus valores almacenados, por lo que, ejecutando el código se obtiene lo siguiente:

```
> {iphone 3500 [1 2] 18 Mpx 1500.5} estructura my_phone
  {iphone 3500 [1 2] 18 Mpx 1500.5} estructura my_phone2
  Luego de hacer cambios
  {iphone 2700 [1 2] 18 Mpx 1000} estructura my_phone2
  {iphone 3500 [1 2] 18 Mpx 1500.5} estructura my_phone
```

La primera y segunda línea muestran los valores de *my_phone* y *my_phone2* sin aplicar ningún tipo de modificación. Se puede observar que sus valores son iguales, mientras que, en las líneas posteriores al mensaje de modificación, se muestra que independiente de los cambios realizados en *my_phone2*, los valores de *my_phone* mantienen intactos. Con esta situación, se puede deducir que, como sucede en los punteros con el paso de información por copias de esta, de igual manera pasa con las estructuras (ver apartado *7. Punteros*).

En este punto, se vuelve crucial el uso de punteros para el manejo de estructuras que almacenan información similar. Para esto, lo único que debe de hacerse es cambiar la

forma en cómo se inicializa la variable *my_phone2*, que de estar de por el paso de copias, debe de pasar a estar por el paso de referencias (direcciones):

```
...  ...
4    my_phone2 := &my_phone
...  ...
```

Al hacer únicamente este cambio y ejecutar el código se obtiene:

```
>    {iphone 3500 [1 2] 18 Mpx 1500.5} estructura my_phone
     &{iphone 3500 [1 2] 18 Mpx 1500.5} estructura my_phone2
     Luego de hacer cambios
     &{iphone 2700 [1 2] 18 Mpx 1000} estructura my_phone2
     {iphone 2700 [1 2] 18 Mpx 1000} estructura my_phone
```

Observando cuidadosamente, las dos variables manejadas en este momento pueden obtener las mismas modificaciones y por consiguiente iguales valores, gracias a la redefinición anterior.

6.9 Otras características de las estructuras

6.9.1 Estructuras anónimos

Las estructuras tienen la propiedad de acoplarse a un entorno global, es decir, estar abiertas a todo tipo de variable que quiera instanciar un dato de dicha estructura a partir de un nombre característico.

Por otra parte, se presentan un tipo de estructuras derivadas:

```
1    package main
2
3    Import “fmt”
4
5    func main(){
6    worker_1 := struct {
7              name   string
8              charge string
9              salary float32
10        }{name: "Jhon Dowy", charge: "Distribuidor de productos",
     salary: 3000.0}
11        fmt.Print(worker_1)
12   }
```

Al ejecutar el código anterior, se obtiene:

```
>  {Jhon Dowy Distribuidor de productos 3000}
```

Del código se observa la creación de una nueva estructura la cual no tiene asignado un nombre específico, sin embargo, esta se encuentra asignado como valor de una variable que es *worker_1*. Esta estructura sin nombre se define e inicializa de la misma forma que en casos anteriores. La única diferencia radica en que esta solo se encuentra disponible para la variable en la cual se ha almacenado.

Este tipo de datos se conocen como las estructuras anónimas, y hacen alusión a su nombre puesto que se encuentran disponibles únicamente para una parte del código, mientras que para el resto es invisible.

Este proceso es posible gracias a la característica que comparten las estructuras de ser también el tipo de valores asignable y comparable entre variables.

```
1   package main
2
3   import "fmt"
4
5   type cpu struct {
6         reference string
7         cores     int
8         Hz        int
9   }
10
11  type Phone struct {
12        reference string
13        cpu
14        battery           int
15        memory_capacity   []int //[storage,RAM]
16        camera_definition string
17        price             float32
18  }
19
20  func main() {
21
22        my_phone := Phone{
23              reference:         "iphone",
24              cpu:              cpu{reference: "snapdragon 817,", cores: 4, Hz: 100},
25              battery:           3500,
26              memory_capacity:   []int{1, 2},
27              camera_definition: "18 Mpx",
28              price:             1500.5,
29        }
30
31        fmt.Println(my_phone)
32  }
```

En el código se ha creado una nueva estructura *cpu* que define los campos de *reference*, *cores* y *Hz* que describen las características principales de la unidad de procesamiento principal de un celular (estructura *Phone*).

Se observa que dentro de la estructura *Phone* se ha creado un nuevo campo llamada *cpu*, el cual no se define con ningún tipo de variable conocido. En este caso, este tipo de definición hace alusión a la incrustación de una nueva estructura en forma de campo. Esto permitirá definir las otras características del celular referentes a su procesador (o cpu). Por ejemplo:

```
> {iPhone {snapdragon 817, 4 100} 3500 [1 2] 18 Mpx 1500.5}
```

Al ejecutar el código se obtiene toda la información almacenada por *my_phone* , el cual ha sido inicializado con distintos valores, tanto para la estructura de la cual es instancia (*Phone)* como para la incrustación de una nueva estructura (*cpu*) como parte de toda la representación que indica *my_phone*.

Cabe resaltar que el acceso a los datos internos del campo *cpu* es de la misma manera que con las definiciones anteriores, por ejemplo:

```
...  ...
4        fmt.Println(my_phone)
5        fmt.Println(my_phone.cpu.reference)
6        fmt.Println(my_phone.cores)
7        fmt.Println(my_phone.Hz)
...  ...
```

Como resultado se obtiene:

```
> {iphone {snapdragon 817, 4 100} 3500 [1 2] 18 Mpx 1500.5}
  snapdragon 817,
  4
  100
```

Nótese que el acceso a los campos internos en *cpu* puede hacerse de dos formas:

- Indicando la ruta:

  <nombre de la variable>.<campo>.<campo>...

-Indicando el nombre del campo especifico

Para el caso de la segunda opción, cada campo debe de tener un nombre único, puesto que, si se está realizando embedding con estructuras que contienen campos con nombres iguales, al momento de compilar código, la máquina entenderá el campo más próximo al que pueda llegar, es decir, la estructura más externa. Dicha situación sucedería con los campos *reference* de *Phone* y *cpu*:

- En este caso, la idea es obtener el valor almacenado en el campo de reference para la estructura *cpu*:

```
5 | fmt.Println(my_phone.reference)
```

Se obtiene:

```
> | iphone
```

6.9.3 Funciones de estructuras

Las estructuras, aparte de crear datos con características definidas por el usuario también permite crear procedimientos (funciones) que permitan trabajar únicamente con ese tipo de dato. Dichas funciones deben cumplir con la siguiente estructura de definición:

func (<nombre del parámetro> <nombre de la estructura>) <nombre de la función> (<parámetros de la función>) <valores que retorna la función> {
<bloque de instrucciones>
}

Por ejemplo, para formar una función propia de la estructura que ha servido de ejemplo en secciones anteriores se tendría lo siguiente:

```
… | …
4 | func (phone Phone) Modify(property string, newValue int){
… |       …
9 | }
```

Lo anterior, teniendo en cuenta su nombre podría indicar una función que modifica los valores de ciertas propiedades de la estructura de tipo *Phone* que utilice dicha función.

6.9.3.1 Algunos ejemplos

Continuando con lo anterior, se podrían tener las siguientes funciones:

```
1  package main
2
3  import "fmt"
4
5  type cpu struct {
6       reference string
7       cores     int
8       Hz        int
9  }
10
11 type Phone struct {
12      reference string
13      cpu
14      battery           int
15      memory_capacity   []int //[storage,RAM]
16      camera_definition string
17      price             float32
18 }
19
20 func (phone Phone) Modify(property string) {
21      var newValue, option int
22      switch property {
23      case "cpu":
24           fmt.Printf("Que desea cambiar de la cpu de su
       %v ? ", phone.reference)
25           fmt.Println("\n1. Cores")
26          fmt.Println("2. Frecuencia")
27          fmt.Scanf("%v", &option)
28          if option == 1 {
29               fmt.Println("CORES")
30               fmt.Println("Ingrese un nuevo valor -> ")
31               fmt.Scanf("%v\n", &newValue)
32               fmt.Printf("Cantidad de cores actual ->
       %v \n", phone.cpu.cores)
33               phone.cpu.cores = newValue
34               fmt.Println("Los cores han sido
35               fmt.Printf("Cantidad de cores actual ->
       %v \n", phone.cpu.cores)
36          } else {
37               fmt.Println("FRECUENCIA")
38               fmt.Println("Ingrese un nuevo valor -> ")
```

```
39                fmt.Scanf("%v\n", &newValue)
40                fmt.Printf("Frecuencia actual -> %v \n",
phone.cpu.Hz)
41                phone.cpu.Hz = newValue
42                fmt.Println("La frecuencia ha sido
43                fmt.Printf("Frecuencia actual -> %v \n",
phone.cpu.Hz)
44            }
45
46        case "battery":
47                fmt.Println("BATERIA")
48                fmt.Println("Ingrese un nuevo valor -> ")
49                fmt.Scanf("%v\n", &newValue)
50                fmt.Printf("Batería actual -> %v \n",
phone.battery)
51                phone.battery = newValue
52                fmt.Println("La batería ha sido cambiada
53               fmt.Printf("Batería  actual    -> %v  \n",
54
55      case "camera":
56                fmt.Println("CAMARA")
57                StringNewValue := ""
58                fmt.Println("Ingrese un nuevo valor (solo
número de pixeles) -> ")
59                fmt.Scanf("%v\n", &StringNewValue)
60                fmt.Printf("Pixeles actuales de la camara
-> %v \n", phone.camera_definition)
61                StringNewValue = StringNewValue + " Mpx"
62                phone.camera_definition = StringNewValue
63                fmt.Println("La batería ha sido cambiada
64                fmt.Printf("Pixeles actuales de la camara
-> %v \n", phone.camera_definition)
65            default:
66              fmt.Printf("No se puede ejecutar el cambio.
No existe la propiedad %v", property)
67          }
68  }
69
70  func (phone Phone) have_good_camera(pixels int) {
71        if pixels <= 12 {
72              fmt.Println("Su camara es de gama baja")
73        } else {
74              if pixels > 12 && pixels <= 24 {
75                    fmt.Println("Su camara es de gama
76              } else {
77                    fmt.Println("Su camara es de gama
```

```
78
79          }
80
81  }
82  func main() {
83
84      my_phone := Phone{
85              reference:          "iphone",
86              cpu:                cpu{reference: "snapdragon
    817,", cores: 4, Hz: 100},
87              battery:            3500,
88              memory_capacity:    []int{1, 2},
89              camera_definition: "18 Mpx",
90              price:              1500.5,
91              }
92
93              my_phone.have_good_camera(18)
94              fmt.Println("")
95              my_phone.Modify("cpu")
96              fmt.Println("")
97              my_phone.Modify("battery")
98              fmt.Println("")
99              my_phone.Modify("camera")
100 }
```

Del código anterior se crearon dos funciones:

-Modify(): Permite modificar las características de *cpu, battery* y *camera*. Utiliza *property* que almacena un dato *string* que indica la opción a modificar. Dentro de la estructura *switch* que maneja las opciones de modificación, dependiendo de la opción escogida se pregunta por su nuevo valor.

-Have_good_camera(): Esta función indica la calidad de la cámara con la que cuenta una instancia de la estructura *Phone*. Se clasifican como *gama baja* si la resolución almacenada en la instancia de la estructura es menor o igual a doce, *gama media* si la instancia es mayor que doce y menor o igual que veinticuatro, y *gama alta* si no es ninguna de las dos categorías anteriores. Observe que, en esta función a comparación de la anterior, se está utilizando un *if* anidado en lugar de un *switch*.

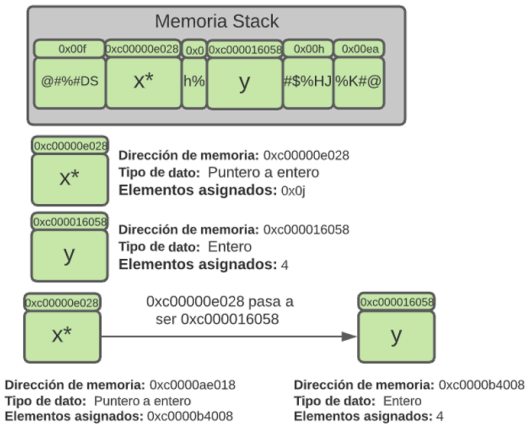
Ejecutando lo anterior se obtiene lo siguiente:

>	Su camara es de gama media
	¿Que desea cambiar de la cpu de su iphone ?
	1. Cores
	2. Frecuencia
	1
	CORES
	Ingrese un nuevo valor ->
	8
	Cantidad de cores actual -> 4
	Los cores han sido cambiados
	Cantidad de cores actual -> 8
	BATERIA
	Ingrese un nuevo valor ->
	4500
	Batería actual -> 3500
	La batería ha sido cambiada
	Batería actual -> 4500
	CAMARA
	Ingrese un nuevo valor (solo número de pixeles) ->
	45
	Pixeles actuales de la camara -> 18 Mpx
	La batería ha sido cambiada
	Pixeles actuales de la camara -> 45 Mpx

Punteros

Son un tipo de dato que almacena la dirección de memoria de una variable. La dirección mencionada hace referencia a la ubicación del dato dentro de la memoria principal en la cual se ejecuta el programa.

```
package main

func main() {


    var x *int
    y := 4




    x = &y //x almacena (o apunta a) la dirección de y


//Información de la variable x
    fmt.Printf("Dirección en memoria de la variable x -> %v \n", &x)
    fmt.Printf("Lo que almacena la variable x -> %v \n", x)
    fmt.Printf("Lo que almacena la dirección guardada en x -> %v \n", *x)

//Información de la variable y
    fmt.Printf("Dirección en memoria de la variable y -> %v \n", &y)
    fmt.Printf("Lo que almacena la variable y -> %v \n", y)
}
```




```
Dirección en memoria de la variable x -> 0xc00000e028
Lo que almacena la variable x -> 0xc000016058
Lo que almacena la dirección guardada en x -> 4
Dirección en memoria de la variable y -> 0xc000016058
Lo que almacena la variable y -> 4
```

Ilustración 11. Representación gráfica de los punteros

7.1 Creación de Punteros

Como Go es un lenguaje estáticamente tipado las variables puntero tienen que ser de un tipo de dato específico.

Para crear un puntero se utiliza el operador '*' antes del tipo de dato que necesitamos almacenar en esa dirección de memoria, por ejemplo:

*var p *int*

La variable P es un puntero que almacena la dirección de un dato entero. Al no estar inicializada, su valor es nulo (null) o vacío.

7.2 El operador de des- referenciación*

Antes de explicar la sintaxis, es necesario comprender el significado de una referencia. Son valores que permiten acceder indirectamente a otra información dentro de un programa. Pueden entenderse con el siguiente ejemplo, muchas de las casas poseen una dirección de domicilio explícita, ya sea descrita por números o nombres de calles cercanas. Esta dirección se usa para poder reconocer la ubicación exacta de la casa y poderla diferenciar del resto, lo que puede ayudar tanto al propietario como posibles visitantes en su llegada. En el caso de punteros, una variable que almacene una dirección sería un visitante a dicha casa, la cual conoce su ubicación por medio de la dirección que la describe, es decir su referencia.

Ahora bien, el proceso de des-referenciación indica la obtención del valor almacenado en una variable a partir de su dirección por parte de un puntero, o en palabras del ejemplo anterior, es conocer quien es la persona que vive en la casa que tiene asignada la dirección del visitante. Para usarse se debe utilizar el operador de construcción anterior '*', por ejemplo:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6       var x *string
7       y := "Punteros"
8
9       x = &y //x almacena (o apunta a) la dirección de y
10      fmt.Printf("Lo que almacena la variable y -> %v \n", y)
11      fmt.Printf("Lo que almacena la variable x -> %v \n", x)
12      fmt.Printf("Des-referenciación del valor almacenado en x
    -> %v \n", *x)
13 }
```

El resultado del código anterior es:

```
>  Lo que almacena la variable y -> Punteros
   Lo que almacena la variable x -> 0xc000010200
   Des-referenciación del valor almacenado en x -> Punteros
```

7.3 El operador de dirección &

El caso anterior proporcionaba el valor almacenado en variable a partir de una dirección de memoria, ahora, el operador de dirección obtiene la dirección de memoria de una variable asignada a un puntero, de otro modo, conocer la dirección que indica la ubicación de una casa. Para usarse se debe utilizar el operador '&' (es también conocido como ampersand), por ejemplo:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6       var x *string
7       y := "Punteros"
8
9       x = &y //x almacena (o apunta a) la dirección de y
10
11      fmt.Printf("Dirección en memoria de la variable y -> %v
   \n", &y)
12      fmt.Printf("Dirección en memoria de la variable x -> %v
   \n", &x)
13      fmt.Printf("Lo que almacena la variable y -> %v \n", y)
14      fmt.Printf("Lo que almacena la variable x -> %v \n", x)
15      fmt.Printf("Des-referenciación del valor almacenado en x
   -> %v \n", *x)
16
17 }
```

El resultado del código anterior es:

```
>  Dirección en memoria de la variable y -> 0xc00008e1e0
   Dirección en memoria de la variable x -> 0xc0000ae018
   Lo que almacena la variable y -> Punteros
   Lo que almacena la variable x -> 0xc00008e1e0
   Des-referenciación del valor almacenado en x -> Punteros
```

7.4 Los punteros y la asignación de valores a funciones

En capítulos anteriores se observó los conceptos y tipos de variables, así como la estructura y el manejo de funciones. Concretamente, el uso de las funciones se logra mediante el llamado de estas y la asignación de argumentos a los parámetros definidos. A este proceso se le conoce también como *pasos de variable*. Existen dos tipos, que son:

- **Paso por valor:** También conocido como "paso por copia", es el proceso de asignar copias de una variable a una función. Al trabajar con versiones distintas no es posible cambiar el valor de la variable original por medio de funciones.

- **Paso por referencia:** Teniendo en cuenta la definición anterior de referencia y entendiéndola como una dirección de una única variable, este procedimiento describe la asignación de variables originales a una función, es decir, entregar únicamente la variable creada y no recurrir a la construcción de copias. A diferencia del anterior, en este proceso si es posible modificar el valor de una variable por medio de funciones.

Los dos tipos pueden ser asimilados con una variación del ejemplo anterior, el cual es el siguiente:

Hay un grupo de constructores encarados de remodelar una casa con las siguientes características: la casa es de color rojo, tiene dos plantas, y en el segundo piso se encuentra un ventanal que ocupa aproximadamente todo el frontal (vista desde afuera) del piso. Este grupo de constructores conoce la locación general del lugar de trabajo, pero no la dirección exacta de la casa solicitada, aun así, ellos deciden dirigirse al sitio; al llegar se topan con un problema, en la ubicación se encuentran dos casas con exactamente las mismas características, pero con una diferencia que reside en sus direcciones de domicilio.

Teniendo en cuenta que no se sabe cuál de las opciones es la correcta existen dos escenarios posibles, el primero es que la remodelación se haga en la casa original y el segundo es que esta se lleve a cabo en la casa con iguales características. Se debe resaltar que la única manera de poder dar con la casa correcta se es necesario conocer su dirección de domicilio.

Es en este punto donde se hacen presentes los procesos de comunicación de variables a funciones, siendo el primer escenario un referente para el paso por referencia, mientras que el segundo lo es para el paso por valor.

Nota: Cabe resaltar que, el proceso interno de la comunicación entre variables y direcciones de memoria no es visto como un problema (como lo expuesto en el

ejemplo) puesto que, dependiendo del tipo de paso a utilizar, el sistema automáticamente se prepara para manejar ya sea copiar, como la variable original.

7.4.1 Ejemplo

A continuación, se presenta un código ilustrativo de todo lo mencionado acerca de paso por referencia y paso por valor.

```
1  package main
2
3  func cambiarDatoVALOR(a int) {
4       println("Valor de a Inicial dentro de función -> ", a)
5       a = 80
6       println("Valor de a Final dentro de función -> ", a)
7  }
8  func cambiarDatoREFERENCIA(a *int) {
9       println("Valor de a Inicial dentro de función -> ", *a)
10      *a = 80
11      println("Valor de a Final dentro de función -> ", *a)
12 }
13 func main() {
14
15      //pasos por valor (copia)
16      println("PASO POR VALOR")
17      a := 8
18      cambiarDatoVALOR(a)
19      println("Valor de a fuera de la función", a)
20
21      //Paso por referencia
22      println("PASO POR REFERENCIA")
23      cambiarDatoREFERENCIA(&a)
24      println("Valor de a fuera de la función", a)
25
26 }
```

Al ejecutar el código anterior se obtiene:

```
>   PASO POR VALOR
    Valor de a Inicial dentro de función ->  8
    Valor de a Final dentro de función ->  80
    Valor de a fuera de la función 8
    PASO POR REFERENCIA
    Valor de a Inicial dentro de función ->  8
    Valor de a Final dentro de función ->  80
    Valor de a fuera de la función 80
```

Obsérvese que se ejemplifican los dos pasos de variables. En el **paso por valor** se comprueba que los valores originales de *a* se mantienen intactos, aún existan cambios dentro de una función. Por otro lado, el **paso por referencia** comprueba el uso de la dirección de la variable original, permitiendo la asignación de nuevos valores de manera indirecta.

7.5 memoria STRACK y HEAP

Todos los programas de un computador necesitan cargar, leer y procesar datos en todo momento. De aquí la distribución de varios tipos de memoria, entiendo algunos como:

Disco duro: El que permite almacenamiento de información a largo plazo, o de otro modo, el que mantiene la información que no será utilizada instantáneamente.

- RAM: Memoria de acceso aleatorio, hace referencia a los elementos y procedimientos que utilizan memoria en un momento cualquiera para el procesamiento de datos. Estos permiten el funcionamiento de los programas.

- Caché: Memoria de rápido acceso que almacena direcciones de distintas fuentes de almacenaje, esto es, maneja los lugares en los cuales la RAM, Disco Duro, entre otras, trabajan. Este proceso permite que los futuros procesos de las aplicaciones usadas dentro de un computador se hagan de una forma rápida y eficiente, evitando inicializar todo desde cero.

En el mundo de los computadores existen distintos tipos de memoria, sin embargo, es crucial tener presente dos de las principales en el desarrollo de la programación, la memoria *Stack*(pila) y la memoria *Heap* (montón).

7.5.1 Memoria STACK

Conocida también como *pila* de control, funciones, llamados, entre otras. La memoria stack es una estructura de datos dinámica (ya que se puede acomodar al tamaño de

los datos) que almacena la información del llamado de las subrutinas en tiempo de ejecución de un programa. Esto es almacenar datos particulares de funciones (como el tipo de variables y sus asignaciones, el tipo de retorno, su dirección, etc.) en una estructura definida por el proceso LIFO (Last in First Out, "Ultimo Adentro Primero Afuera").

Gráficamente se puede representar de la siguiente manera:

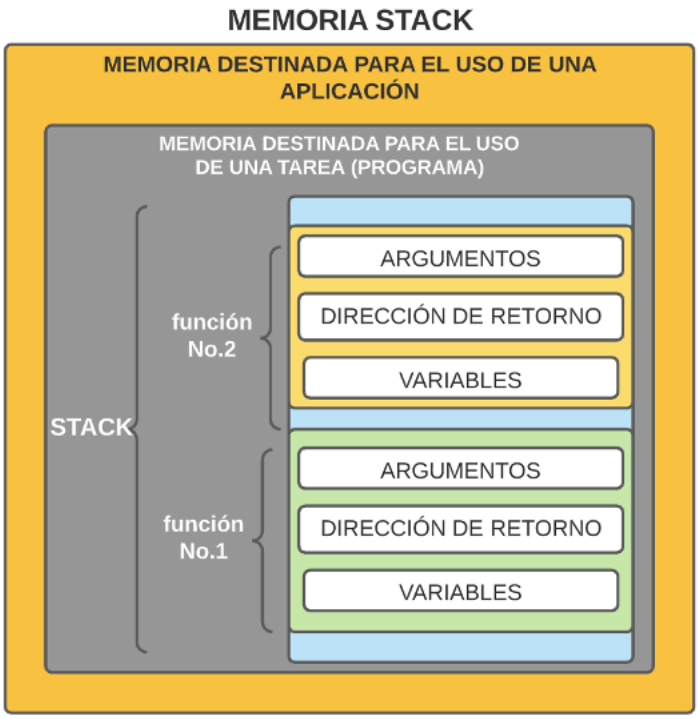


Ilustración 12. Representación gráfica de la memoria Stack

7.5.2 Memoria HEAP

En español es conocida como "monton", y también ha sido nombrada almacenamiento o zona libre. La memoria Heap es una estructura dinámica para el almacenamiento de datos, y en donde su actividad se basa en la asignación de memoria dinámicamente para la creación de objetos (tipos de datos conocidos en la Programación orientada a objetos, POO) y nuevos datos para posteriormente almacenarlos de forma continua en este tipo de memoria.

Este tipo de estructura no cumple ninguna metodología de organización por lo que el mismo nombre indica su forma de trabajo que consiste en partir de un monto

especifico designado para un programa, y del cual este puede valerse para la creación de nuevos datos.

Gráficamente se puede representar de la siguiente manera:

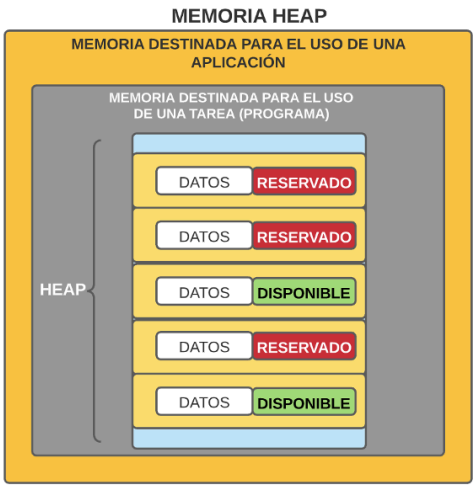


Ilustración 13. Representación gráfica de la memoria Heap

7.5.3 Ejemplo de STACK y HEAP

A partir del siguiente código se puede representar la idea que define cada uno de los tipos de memoria:

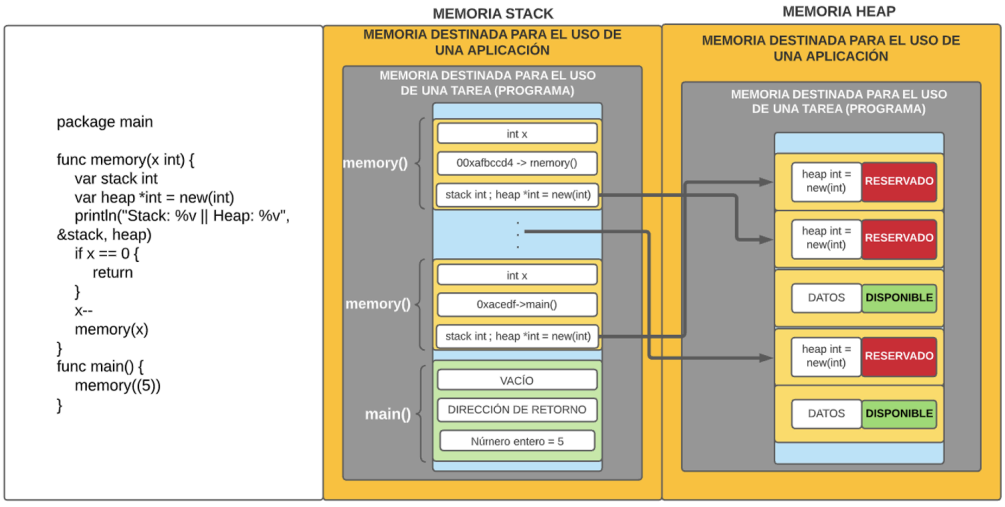


Ilustración 14. Representación en código del funcionamiento de la memoria Stack y Heap

El ejemplo usado es sencillo. Consta de dos funciones, *main()*, quien es la parte principal que ejecuta todo el funcionamiento del programa; y *memory()*, una función que de forma recursiva muestra el uso de la memoria Stack a través de las llamadas de subrutinas, y la memoria Heap mediante la asignación de nueva memoria a variable de tipo *int* (entero).

La estructura de *memory()* recibe un parámetro entero que servirá como contador del número de veces que se quiere hacer la recursión. Por otro lado, las variables *Stack y Heap* hacen alusión al funcionamiento de cada tipo de memoria descrito anteriormente. Observe que la variable *Stack* se define a partir de la estructura básica de variables, mientras que *Heap* es un puntero a entero. En la definición de esta última se utiliza una nueva función *new(int)* la cual se define como la función propia del lenguaje Go encargada de asignar dinámicamente la memoria a un tipo de dato, que a su vez asigna el valor cero como valor inicial de este último. Las asignaciones hechas por *new()* son mediante direcciones, es decir, al momento de usarla, el dato de retorno por su parte es una dirección de memoria perteneciente al nuevo bloque de almacenamiento destinado al tipo de dato solicitado.

De la misma manera, en la ilustración de la memoria las flechas que salen de *MEMORIA STACK* apuntan a las casillas de *MEMORIA DESTINADA PARA EL USO DE UN PROGRAMA* en *MEMORIA HEAP*.

De todo lo descrito anteriormente, se puede concluir que para el uso de nueva de memoria es necesario el uso de punteros que almacenen la dirección en la cual se proporciona la nueva memoria a cierto tipo de dato.

7.6 Comparación entre los dos tipos de memoria

A continuación, se presenta un cuadro ilustrativo de las comparaciones entre la memoria Stack y Heap según el criterio dado:

CRITERIO	STACK	HEAP
Asignación de memoria	•La memoria es lineal, se posiciona en bloques contiguos. •Su funcionamiento no depende del programador, es automático. Por ende, su liberación es manejada por el sistema operativo.	•La memoria se posiciona dependiendo el tamaño del dato a almacenar. •Su funcionamiento depende del programador. Por ende, liberar memoria depende del programador
Costo (en memoria)	Menor	Mayor
Implementación	Fácil, es automático	Difícil, todo el proceso depende del
Memoria	•Tiene un tamaño definido. •Puede quedarse sin memoria, hasta el punto de desbordarse y ocasionar un *"stack Overflow"*	•Tiene un tamaño definido más grande que el sack. •La memoria se fragmenta a conveniencia del dato almacenado.
Procesos (hilos)	Mantiene la memoria de un programa independiente de otros procesos. Puede crearse en ambientes que llevan a cabo trabajos multi-hilos, es decir, múltiples "procesos ligeros" o subrutinas (un subproceso o hilo se entiende como un conjunto de secuencias de tareas muy pequeñas	Queda al descubierto de otros procesos que se ejecuten en el computador, por lo que puede ocurrir un estado de confusión en donde se integre memoria de otros procesos.

Tabla 9. Comparación entre la memoria Stack y Heap

Nota: El lenguaje de Go maneja un recolector de basura (garbage collector) encargado de limpiar la memoria luego de que los objetos declarados dejan de ser usados. En este caso en particular no se es necesario liberar memoria de forma explícita, sin embargo, en otros lenguajes de programación se debe realizar (a menos que se gestione en el mismo lenguaje).

7.7 Cantidad de memoria usada según el tipo de dato

TIPOS DE DATOS	CANTIDAD DE MEMORIA
byte	8 bits
char (carácter)	8 bits – 1 byte
Int (entero)	32 bits – 4 bytes
float (flotante)	32 bits – 4 bytes
short (corto)	16 bits – 2 bytes
long (largo)	64 bits – 8 bytes
double (doble)	64 bits – 8 bytes
boolean	64 bits – 8 bytes

Tabla 10. Cantidad de memoria usada por cada tipo de dato

Otras estructuras de datos

8.1 Concepto sobre las estructuras de datos

Una estructura de datos se define como la herramienta que nos permite juntar y organizar datos de uno o distintos tipos, con el objetivo de proveer un uso más eficiente.

El uso de las estructuras de datos también permite manejar grandes volúmenes de datos, lo que ayuda en procesos del desarrollo del software como el internet, bases de datos, cálculos de gran magnitud; así como la implementación de algoritmos, permitiendo estandarizar la solución de un problema en un conjunto de pasos.

8.2 Clasificación

La distribución de las distintas estructuras de datos se hace con base en el tamaño y procesamiento. De lo anterior, se tienen las siguientes estructuras:

Tamaño

•**Estático:** Estructura de datos que mantiene el tamaño o cantidad total de datos fija. A medida que se opera con ella no se podrá cambiar el valor de su tamaño a menos que se utilice una nueva estructura y se re-posicionen nuevamente los datos almacenados en la estructura inicial.

Ejemplos: arreglos (vectores, matrices), structs.

•**Dinámico:** Estructura de datos que permite cambiar el tamaño o cantidad total de datos. Al operar con estas estructuras, en medio de la ejecución estas pueden almacenar o eliminar información, configurando a la vez el tamaño que las describe.

Ejemplos: Listas, pilas, colas, árboles, grafos, deque

8.3 Procesamiento

•Lineal: Los elementos dentro de estas estructuras se organizan uno tras otro.

Ejemplos: arreglos (vectores, matrices), listas, pilas, colas, deque.

•No lineal: En comparación con el caso anterior, los datos en estructuras no lineales se organizan de forma aleatoria.

Ejemplos: Árboles, grafos, maps,

Nota: Al mencionar la forma en cómo se ordenan los elementos, esta noción de organización se hace teniendo en cuenta la colocación y el uso de memoria al momento de ejecutar un programa. Este tema se abarca superficialmente en la sección de Punteros

Hasta este punto del desarrollo del texto de programación con Go, se han observado distintas estructuras de datos, como, por ejemplo:

-Arreglos unidimensionales y bidimensionales
-Slices
-Maps
-Structs

Cada uno representa una manera distinta de posicionar y manejar un cierto tipo de información, sin embargo, es recomendable para tener una visión más completa sobre las distintas estructuras existentes, tener en cuenta algunas de las siguientes estructuras:

- Lista
- Pila, Cola y Deque

8.4 Listas

Las listas son un conjunto de elementos que pueden ser de uno o más tipos. Es una de las estructuras más flexibles en la programación ya que siempre permite modificar su tamaño, haciendo uso de operaciones básicas de inserción y eliminación de elementos.

Una lista se representa de la siguiente manera:

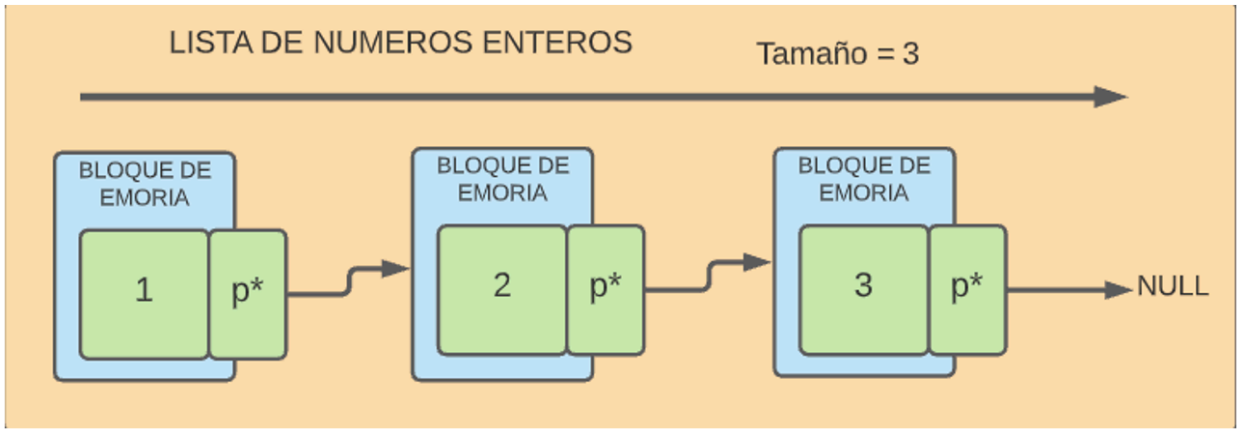


Ilustración 15. Representación gráfica de una lista enlazada

Se puede observar que cada elemento (también conocidos como nodos) se sitúa uno tras otro gracias a la flecha y al símbolo p_n^*. Este elemento es un puntero que almacena la dirección del siguiente dato introducido en la lista. Los elementos dentro de una lista enlazada pueden posicionarse en distintas locaciones dentro de la memoria (no necesariamente de forma consecutiva como en los arreglos), por lo que para poder acceder a cada uno se debe de almacenar la dirección en donde este se encuentra. Esto puede verse como secciones de memoria independientes por cada sección que describe un número y junto a este un apuntador. Con esto último, cada sección es independiente de las otras, es decir, cada una comparte un sitio de memoria específico, que a la vez contiene información sobre el siguiente elemento dentro de la estructura que encierra a todos las otras zonas.

Haciendo una breve comparación con los arreglos, estos se representaban como cuadrados contiguos o pegados entre sí que almacenan cierta información, y que, por ende, el conjunto de todos estos bloques conforma una sola sección de memoria. Las listas no cumplen con esta propiedad de juntar entre si sus elementos, como se menciona en el párrafo anterior. De esto último se deduce una característica particular de esta estructura (y otras que manejan el concepto de punteros) y es su forma de organización física.

Entre otros detalles, el tamaño de una lista se compone a partir del número total de sus elementos, y al ser una estructura dinámica dicho tamaño puede cambiar partiendo de estas situaciones:

1. Sin importar la memoria designada en el momento en que se crea
2. En medio de su ejecución o procesamiento

El elemento *null* presente en la ilustración 13 indica que no hay más direcciones hacia nuevos elementos, de otra forma, indica que es un puntero nulo, es decir, un puntero que no tiene dirección.

8.4.1 Otros tipos de listas

Existe una clasificación de las listas enlazadas según la comunicación entre sus elementos, que son la siguientes:

- **Listas doblemente enlazadas:** Estructuras que almacenan la dirección del elemento anterior y el elemento siguiente, es decir, cada elemento (o nodo) posee dos direcciones de memoria.
- **Lista circular:** Estructuras que comunican su último elemento con el primero.
- **Listas circulares doblemente enlazadas:** Una combinación de los dos tipos de listas anteriores. Consiste en una estructura en que cada nodo contiene la dirección del siguiente y anterior elemento, con la adición de que el último elemento está conectado con el primero.

8.4.2 Operaciones fundamentales sobre las listas

Las listas manejan ciertas operaciones fundamentales que son:

Operación	Concepto
Size()	Retorna el valor del tamaño de una lista
Add()	Inserta un elemento (puede ser al final, inicio, o en una posición aleatoria de la estructura)
Remove()	Remueve un elemento (puede ser de cualquier posición)
Get()	Obtiene el elemento de una posición dada
Set()	Cambia el elemento de una posición dada

Tabla 11. Operaciones fundamentales de las listas

En el concepto de lista se menciona "elementos de un cierto tipo" como parte de estas estructuras. Estos elementos o "nodos" se componen también de una estructura con un conjunto de operaciones, que son las siguientes:

Tabla 12. Operaciones fundamentales de los nodos

Operación	Concepto
Next()	Retorna la dirección del elemento siguiente
hasNext()	Retorna un valor booleano que indica si hay o no un elemento siguiente
setNext()	Cambia la dirección del elemento siguiente
Data()	Retorna la información almacenada en el nodo
SetData()	Cambia el elemento almacenado en el nodo

8.4.3 Código que describe las listas enlazadas

```
1  package main
2
3  import "fmt"
4
5  type Node struct {
```

```
 5 type Node struct {
 6      next    *Node
 7      element interface{}
 8 }
 9
10 func (Node *Node) getData() interface{} {
11      return Node.element
12 }
13
14 func (Node *Node) SetData(new_element interface{})
15      Node.element = new_element
16 }
17
18 func (Node *Node) Next() *Node {
19       return Node.next
20 }
21
22 func (Node *Node) SetNext(NewNode *Node) {
23       Node.next = NewNode
24 }
25
26 type List struct {
27      First *Node
28      Last  *Node
29      Size  int
30 }
31
32 func (L *List) isEmpty() bool {
33      if L.First == nil {
34           return true
35      } else {
36            return false
37      }
38 }
39
40 func (List *List) getSize() int {
41      return List.Size
42 }
43
44 func (List *List) Add(element interface{}) {
45       var node = &Node{}
46       node.SetData(element)
47
48       if List.isEmpty() {
49            List.First = node
50       } else {
51             List.Last.SetNext(node)
52       }
53       List.Last = node
54       List.Size++
55
56 }
```

```
57
58 func (List *List) Get(position int) *Node {
59
60       node := List.First
61       for index := 0; index < List.Size; index++ {
62            if index == position {
63                 break
64            }
65            node = node.Next()
66       }
67       return node
68 }
69
70 func  (List  *List)  go_Over(element  interface{})
71        var ReferenceNode *Node
72        if !List.isEmpty() {
73             for ReferenceNode = List.First;
   ReferenceNode != nil; ReferenceNode =
   ReferenceNode.Next() {
74                  if ReferenceNode.getData() ==
   element {
75                       break
76                  }
77            }
78     }
79     return ReferenceNode
80 }
81 func  (List  *List)  Set(ListElement,  Newelement
   interface{}) {
82
83        ReferenceNode := List.go_Over(ListElement)
84        if ReferenceNode != nil {
85             ReferenceNode.SetData(Newelement)
86        }
87
88 }
89
90 func  (List  *List)  AddIn(element,  ListElement
   interface{}) {
91
92        var node *Node = &Node{}
93        node.SetData(element)
94
9 o      ReferenceNode := List.go_Over(ListElement)
96
97        if ReferenceNode != nil {
98        node.SetNext(ReferenceNode.Next())
99        ReferenceNode.SetNext(node)
100       }
101       List.Size++
102 }
```

```
103
104 func  (List  *List)  RemoveElementIn(ListElement
    interface{}) {
105         nodeToRemove := List.go_Over(ListElement)
106
107         for node := List.First; node != nil; node =
    node.Next() {
108             if node.Next() != nil &&
    node.Next().getData() == nodeToRemove.getData() {
109                 node.SetNext(nodeToRemove.Next())
110                 nodeToRemove = nil
111                 break
112           }
113       }
114       List.Size--
115 }
116
117 func (List *List) RemoveElementFirst() {
118         List.First = List.First.Next()
119         if List.First == nil {
120               List.Last = nil
121         }
122         List.Size--
123 }
124 func (List *List) RemoveElementLast() {
125       var nodeBeforeLast *Node = &Node{}
126       for  node  :=  List.First;  node  !=  nil;  node  =
    node.Next() {
127             if node.Next() == List.Last {
128                   nodeBeforeLast = node
129                   break
130             }
131       }
132       List.Last = nodeBeforeLast
133       List.Last.SetNext(nil)
134       List.Size--
135 }
136
137 func (List *List) IterateList() {
138         var node *Node
139         for  node  =  List.First;  node  !=  nil;  node  =
    node.Next() {
140             fmt.Printf("%v -> ", node.getData())
141       }
142 }
143
144 func main() {
145       l := List{}
146
147       fmt.Println("Add")
148       l.Add(1)
```

```
149       l.Add("hellow")
150       l.Add(34)
151       l.Add(5.9)
152       l.IterateList()
153       fmt.Println("\nsize -> ", l.getSize())
154
155        fmt.Println("\nget")
156        fmt.Println(l.Get(2), "y el valor de la
dirección es valor de ", l.Get(2).getData())
157
158        fmt.Println("\nSet")
159        fmt.Println("Lista antes de aplicar Set")
160        l.IterateList()
161        l.Set(34, 678568)
162        fmt.Println()
163        fmt.Println("Lista después de aplicar Set")
164        l.IterateList()
165
166        fmt.Println("")
167        fmt.Print("\nAdd-In")
168        l.AddIn("Letter", 1)
169        fmt.Println()
170        l.IterateList()
171        fmt.Println("\nsize -> ", l.getSize())
172
173        fmt.Println("\nRemoveIn")
174        fmt.Println("Lista antes de remover un
elemento ")
175        l.IterateList()
176        l.RemoveElementIn("Letter")
177        fmt.Println("Lista después de remover un
elemento ")
178        l.IterateList()
179        fmt.Println("\nsize -> ", l.getSize())
180
181        fmt.Println("\nRemoveFirst")
182        fmt.Println("Lista antes de remover el
primer elemento ")
183        l.IterateList()
184        l.RemoveElementFirst()
185        fmt.Println("Lista después de remover el
primer elemento ")
186        l.IterateList()
187        fmt.Println("\nsize -> ", l.getSize())
188
189        fmt.Println("\nRemoveLast")
190        fmt.Println("Lista antes de remover el
último elemento ")
191        l.IterateList()
192        l.RemoveElementLast()
```

```
193         fmt.Println("Lista después de remover el
    último elemento ")
194         l.IterateList()
195         fmt.Println("\nsize -> ", l.getSize())
196
197 }
```

Análisis

Como ya se menciona en la teoría referente a las listas enlazadas, estas se componen de unidades que describen los elementos que estas almacenan, llamadas nodos. El conjunto de todos estos unidos por direcciones de memoria conformaría una lista enlazada.

Al comprender la descripción de los nodos, estos se pueden abstraer al código por medio de una herramienta que se comentó en secciones anteriores, la cual permite desarrollar nuevos tipos de datos con características propias definidas por un usuario. Dicha herramienta son las **estructuras** o **structs.** A partir de estas, se pueden desarrollar los datos como *Node* y *List*, con un conjunto de funciones que ejemplifican el comportamiento de la estructura de datos analizada.

•Estructura Node

```
5  type Node struct {
6       next    *Node
7       element interface{}
8  }
9
10 func (Node *Node) getData() interface{} {
11      return Node.element
12 }
13
14 func (Node *Node) SetData(new_element interface{}) {
15      Node.element = new_element
16 }
17
18 func (Node *Node) Next() *Node {
19      return Node.next
20 }
21
22 func (Node *Node) SetNext(NewNode *Node) {
23      Node.next = NewNode
24 }
```

A continuación, se analizará cada una de las funciones de la estructura node:

Nota: En las estructuras se hace fundamental el uso y comprensión de punteros

Type Node struct: Es la definición de cada Nodo que compone a las listas enlazadas. Contiene dos campos, *next* que representa un puntero que almacena la dirección de un nodo, y *element*, una *interface*. Esta última es un dato propio de Go-lang que permite definir el comportamiento de un dato a definir; en otras palabras, este tipo de dato almacena un conjunto de métodos que determinan el desarrollo de un dato. Este se utiliza con el fin de proporcionar múltiples tipos de datos para cada lista enlazada.

Nota: Para mayor entendimiento de las *interfaces*, se recomienda seguir la documentación de Go-lang en su página oficial.

De aquí en adelante se proponen las funciones de estructura, las cuales cumplen con una forma de definición distinta a la creación de funciones (ver más en sección *6. Mapas y Structs*). Dichas funciones son:

getData(): Esta función retorna la información de un nodo, la cual se almacena en el campo de *element*.

setData(): La función cambia la información almacenada en un nodo por una nueva. Esta última se recibe como el elemento de tipo *interface*, que posteriormente se intercambia con el elemento almacenado en *Node.element*.

Next(): Esta función retorna la dirección de memoria del siguiente elemento del nodo que llama a esta función.

SetNext(): Esta función cambia la dirección del elemento siguiente al nodo que llama a esta función. Esta nueva dirección se almacena en *NewNode* como un puntero de tipo *Node*.

En conjunto, todas las funciones anteriores generalizan la comunicación y desarrollo de un nodo a medida que se trabaja con una lista.

•Estructura Lista

```
26 type List struct {
27     First *Node
28     Last  *Node
29     Size  int
30 }
```

Type List Struct: La estructura de las listas se compone de tres campos, *First* y *Last* siendo punteros que almacenan la dirección del primer y último nodo que compone a la lista respectivamente, y *Size* que es un dato de tipo entero el cual indicará el tamaño o número de elementos que almacena la lista.

A continuación, se explican las funciones de la estructura *List*. Para no repetir toda la estructura, dejando lo más esencial, solo se explica cada función, más para poder detallar cada paso se deberá de ubicar en el código proporcionado anteriormente.

isEmpty(): Retorna un valor booleano que determina si la lista está vacía o no. Este procedimiento se determina al verificar si el *L.First* es un puntero nulo (*True*) o no (*False*), es decir, que no hay dirección almacenada en *First*.

getSize(): Obtiene el tamaño de la lista que utiliza esta función. Se determina por medio del dato *Size* con *List.Size*.

Add(): Esta función indica la adición de elementos a una lista enlazada. Utiliza como parámetro para almacenar el nuevo dato a *element interface{}*, que posteriormente se almacenará en el *node* definido dentro de la función con la función de la estructura ***Node SetData().*** Posteriormente se verifica si la lista está vacía, si esto es verdadero, se configura los elementos propios de *List*, *First* y *Last* con la dirección de *node*; si lo anterior es falso, Last configura su dirección hacia el siguiente elemento con la dirección de *node*, luego que reconfigura la dirección de *Last*, pasando a convertirse como último elemento de la lista el nodo añadido, *node*.

Nota: En la definición de un nuevo nodo, la forma &Node{} es equivalente a utilizar la función new()(ver apartado 3.2.1 Uso de make() y new() para la declaración de variables), solo que en este caso es más explicito en especificar la creación de un nuevo dato Node{} al cual se le almacenara en una variable su dirección.

Get(): A partir de un numero entero (almacenado en *position*) se obtiene el elemento almacenado en esa posición dentro de una lista. Su funcionamiento se basa en recorrer toda la lista por medio de un índice que incrementa y puede parar ya sea cuando el valor del índice es igual al dato almacenado en *position*, o bien cuando se recorre toda la lista y el índice es igual al tamaño de la lista.

Go_Over(): Esta función permite retornar un nodo de la lista a partir del elemento que este nodo contenga. Esta información es almacenada en *element interface{}*, para luego verificar si la lista está o no vacía. Nótese que el resultado de la condición anterior esta precedido por un '!' lo que indica que cada resultado booleano se negará, es decir, resultara en un valor opuesto al que realmente otorga la función. Esto permite determinar que cuando una lista está vacía, las operaciones de búsqueda y obtención del nodo no se puedan ejecutar. Caso contrario si la condición es falsa, es decir, que la lista no esté vacía.

En el caso donde no se tenga una lista vacía, el índice *ReferenceNode* toma cada nodo, desplazándose a partir de la dirección almacenada en el elemento propio de cada nodo *next*, hasta el punto en donde el elemento que se almacena en un nodo recorrido es igual al elemento almacenado en *element*. Al cumplirse esto último, se termina el ciclo de recorrido y se retorna *ReferenceNode* con la dirección de memoria del nodo con el elemento buscado.

Set(): Las instrucciones de *Set* permiten cambiar la información de almacenada en un nodo arbitrario dentro de una lista. Esto se lleva a cabo mediante dos parámetros, *ListElement* y *NewElement*, que almacenarán el nodo perteneciente a la lista que será cambiado y el nuevo elemento respectivamente. Se hace uso de la función go_Over() Para obtener directamente el nodo a modificar. Luego se verifica que el nodo retornado por la función anterior, el cual se almacena en *ReferenceNode*, no sea nulo. Caso verdadero y se cambia la información almacenada en *ReferenceNode* por *NewElement*. Caso contrario no se ejecuta ningún cambio.

AddIn(): Añade un elemento a la lista, con la particularidad que puede ser en cualquier posición. Utiliza *Element* y *ListElement,* cada uno de tipo *interface{},* para almacenar el elemento a insertar y el nodo referencia que ayudará a posicionar el nuevo elemento. Se usa *node* para almacenar el nuevo elemento y go_Over() para obtener el nodo referencia, el cual se almacena en *ReferenceNode*. Una vez obtenido, se verifica que este sea un puntero no nulo; caso verdadero y la dirección del siguiente elemento de *node* (dato que se adiciona) se cambia por la dirección del elemento siguiente a *ReferenceNode*, es decir, en este punto, tanto *node* como *ReferenceNode* tienen el mismo elemento siguiente a estos. Por último, se cambia la dirección del elemento siguiente de *ReferenceNode* por la dirección de *node*. Al terminar estas instrucciones se aumenta el tamaño de la lista con *List.size++*.

Los pasos anteriores se pueden resumir en que *ReferenceNode* se utiliza como nodo inicial para posicionar *node*, por lo que al final del procedimiento *node* quedará entre *ReferenceNode* y el elemento siguiente a este.

RemoveElement…(): Esta función de remover elementos se divide en tres partes, todas con el mismo concepto, pero objetivos de eliminación distintos.

- **RemoveElementIn():** Remueve un elemento dentro de la lista distinto a los que se encuentran almacenados en *First* y *Last*. Este a diferencia de los otros tipos utiliza el parámetro *ListElement* para almacenar el elemento que se debe eliminar, que posteriormente será encontrado el nodo que lo contiene con go_Over() y guardado con *nodeToRemove*. Posteriormente se recorre la lista con índices indicados en funciones anteriores. El ciclo para ya sea que el índice llegue a un puntero nulo, o bien que se determine verdadera la condición interna, la cual indica que el siguiente elemento del nodo almacenado en el índice (en el caso de la función *node*) no puede ser un puntero nulo (es decir, debe haber un elemento siguiente), y que la información almacenada en ese elemento siguiente sea igual a la almacenada en *nodeToRemove*.

Al generar una respuesta verdadera se utiliza el índice *nodo* y se cambia el elemento siguiente a este por el elemento siguiente a *nodeToRemove*, es decir, la información que depende del nodo a eliminar será transferida al nodo que guardaba la dirección de este último. Al finalizar, se disminuye el tamaño de la lista con *List.Size--*.

•**RemoveElementFirst():** Al igual que la función anterior, esta instrucción elimina un nodo de la lista, más específicamente el nodo almacenado en *First*. Esto se lleva a cabo en el momento en que se le reasigna a *List.First* una nueva dirección, en este caso es el elemento siguiente al elemento que contiene *First* antes de la reasignación. Luego de esto se determina si el nuevo valor de First es un puntero nulo, que en caso verdadero se cambia *Last* también por un puntero nulo, o caso Falso, no se ejecuta lo anterior y solo se disminuye el tamaño de la lista.

•**RemoveElementLast():** Elimina el último elemento de la lista. Utiliza un nuevo nodo *nodeBeforeLast* que almacenará el elemento anterior a *Last*, esto por medio de un ciclo que recorre la lista en análisis. En cada ciclo el índice toma cada nodo perteneciente a la lista, luego se ejecuta la condición de que, si el siguiente elemento del nodo almacenado por el índice es *Last*, Este nodo sea guardado en la variable de tipo puntero definida al principio. Posterior a esto se rompe el ciclo, y se hace la eliminación reasignando el valor de *Last* al valor de *nodeBeforeLast*, para luego poner el siguiente elemento del nuevo *Last* como una dirección nula y con ello disminuir el tamaño de la lista.

IterateList(): Esta función permite mostrar en la zona de consola o terminal los elementos almacenados en la lista. Funciona a partir de un recorrido, donde se va mostrando en pantalla con la función *fmt.Print()* cada elemento almacenado en cada nodo recorrido.

8.4.4 Pruebas sobre la estructura de Listas

Como se observa en el código del apartado anterior, hay una función *main()* que contiene un conjunto de instrucciones que ejemplifican el comportamiento de una lista enlazada. Dichas instrucciones al ser compiladas generan los siguientes resultados:

```
> Add
  1 -> hellow -> 34 -> 5.9 ->
  size ->  4

  get
  &{0xc00000c060 34} y el valor de la dirección es valor de 34

  Set
  Lista antes de aplicar Set
  1 -> hellow -> 34 -> 5.9 ->
  Lista después de aplicar Set
  1 -> hellow -> 678568 -> 5.9 ->

  Add-In
  1 -> Letter -> hellow -> 678568 -> 5.9 ->
  size ->  5

  RemoveIn
  Lista antes de remover un elemento
  1 -> Letter -> hellow -> 678568 -> 5.9 -> Lista después de
  remover un elemento
  1 -> hellow -> 678568 -> 5.9 ->
  size ->  4

  RemoveFirst
  Lista antes de remover el primer elemento
  1 -> hellow -> 678568 -> 5.9 -> Lista después de remover el
  primer elemento
  hellow -> 678568 -> 5.9 ->
  size ->  3

  RemoveLast
  Lista antes de remover el último elemento
  hellow -> 678568 -> 5.9 -> Lista después de remover el último
  elemento
  hellow -> 678568 ->
  size ->  2
```

8.5 Pilas y colas

8.5.1 Pilas

Conocida también como Stack (en inglés), las pilas son una estructura de datos que permite almacenar y recuperar información a través del sistema de acceso LIFO (Last In First Out), donde el último elemento en entrar es el primero en salir.

Son estructuras lineales que manejan elementos de un solo tipo, y los cuales son manipulados por los extremos del conjunto total de elementos, es decir, donde se ubican el primer y último elemento. De estos dos últimos, se conocerá como TOS (Top of stack o tope de pila) al último elemento, mientras que al primero se le llamará FOS (Front of stack o Frente de la pila).

Lo anterior se puede representar con la siguiente imagen:

Ilustración 16. Representación física de una pila

Las pilas pueden proporcionarse como herramienta en los siguientes escenarios:

- Reconocedores sintácticos de lenguajes independientes del entorno.
- Recursividad.
- Comprensión del manejo de memoria con el Stack de subprocesos.

Nota: Se recomienda que, para una mejor comprensión de las pilas, estas se pueden entender como un conjunto de platos amontonados, una tras otro, que forman una gran torre viéndolo desde una perspectiva general.

8.5.2 Operaciones fundamentales sobre pilas

Se definen las siguientes operaciones fundamentales sobre las pilas o Stack:

Operación	Concepto
Size()	Retorna el valor del tamaño de una pila
Push()	Inserta un elemento por el extremo final de la pila, esto es, donde se sitúa siempre el último elemento dentro de un conjunto lineal.
Pop()	Remueve el Top (Tos) de una pila
isEmpty()	Retorna un valor booleano que determina si la pila analizada está o no vacía

Tabla 13. Operaciones fundamentales de las pilas

Las operaciones de Push() y Pull() pueden representarse de la siguiente manera:

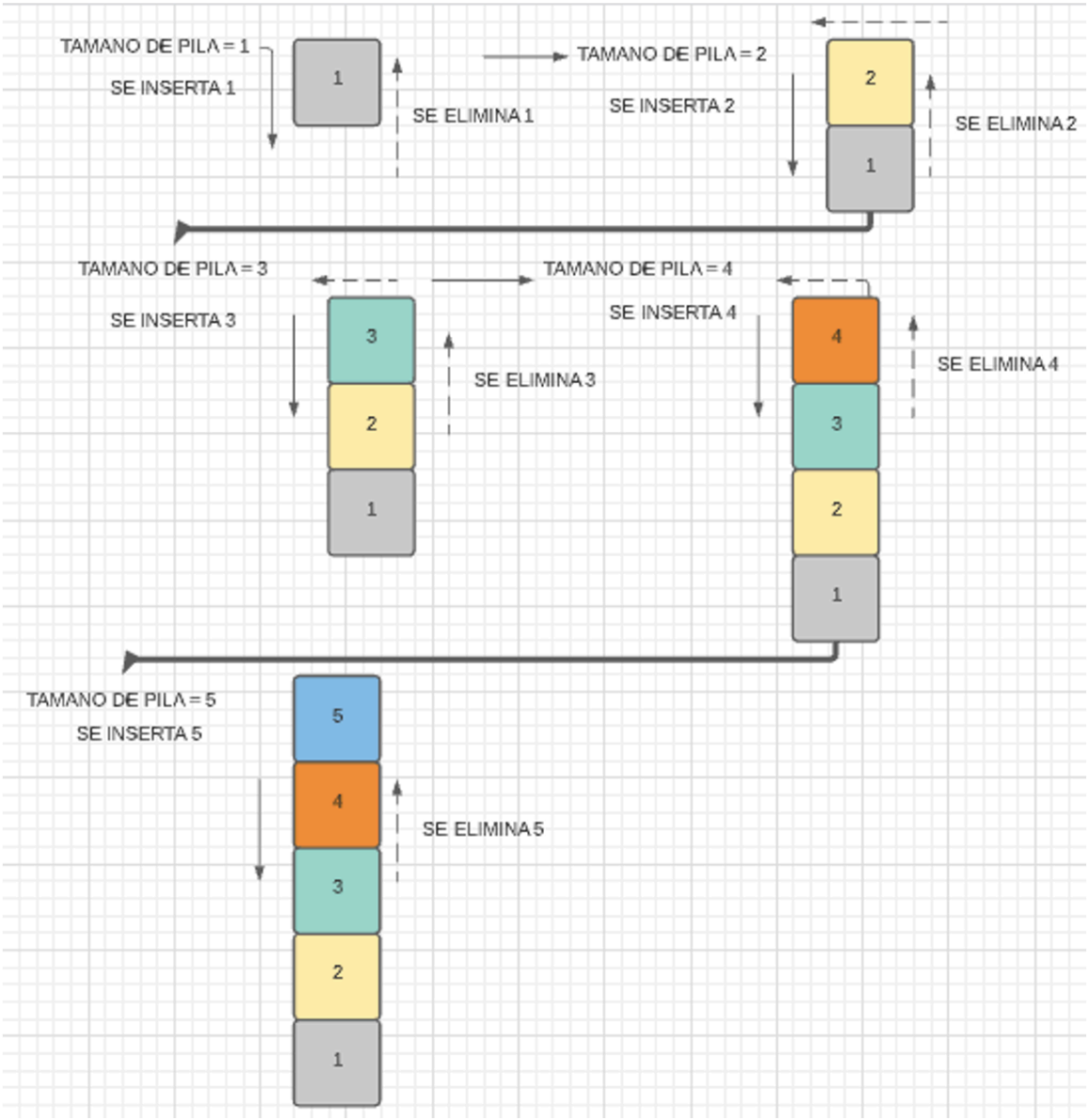


Ilustración 17. Representación de las operaciones Push() y Pul()

En la ilustración 15 se representa el desarrollo de las operaciones de Push() y Pull(). Las flechas continuas representan Push() (insertar) y las flechas punteadas representan Pull() (eliminar). Como se puede observar, la manipulación de cada elemento perteneciente a la pila se desarrolla siempre por un mismo extremo.

8.5.3 Código que describe las pilas

```
1  package main
2
3  import (
4        “fmt"
5  )
6
7  type Stack struct {
8       element []int
9       top     *int
10 }
11
12 func NewStack(size int) *Stack {
13      NewElement := make([]int, size)
14      var p *Stack = &Stack{}
15      if size == 0 {
16            p = &Stack{element: NewElement, top: nil}
17      } else {
18            p = &Stack{element: NewElement, top:
19      }
20
21      return p
22 }
23
24 func (st *Stack) IsEmpty() bool {
25      return len(st.element) <= 0
26 }
27
28 func (st *Stack) Top() *int {
29       if st.IsEmpty() {
30             fmt.Println("El stack está vacío")
31       }
32       return st.top
33 }
34
35 func (st *Stack) setTop(newTop *int) {
36       st.top = newTop
37 }
38
39 func (st *Stack) Size() int {
40       return len(st.element)
41 }
42
```

```
43 func (st *Stack) Push(element int) {
44       st.element = append(st.element, element)
45       newTop := &st.element[st.Size()-1]
46       st.setTop(newTop)
47 }
48
49 func (st *Stack) Pop() {
50
51       if st.Top() != nil {
52            newLen := st.Size()
53            st.element = st.element[:newLen-1]
54       }
55       if st.Size() == 0 {
56           st.setTop(nil)
57      } else {
58           newTop := &st.element[st.Size()-1]
59           st.setTop(newTop)
60      }
61
62      }
63
64 func (st *Stack) Print() {
65       StackSize := st.Size()
66       for index := 0; index < StackSize; index++ {
67             fmt.Printf("%v ->", *st.Top())
68             st.Pop()
69      }
70
71      println()
72 }
73
74 func main() {
75      s := NewStack(0)
76
77      fmt.Println("Añadiendo elementos")
78      s.Push(1)
79      s.Push(2)
80      s.Push(3)
81      fmt.Println("size -> ", s.Size())
82      fmt.Println("El Top es -> ", *s.Top())
83      s.Print()
84
85       fmt.Println("Eliminando elementos")
86       s.Push(1)
87       s.Push(2)
```

```
88          s.Push(3)
89          fmt.Println("Antes de eliminar, size", s.Size())
90          s.Pop()
91          fmt.Println("después de eliminar, size", s.Size())
92          fmt.Println("El Top es -> ", *s.Top())
93          s.Print()
94
95  }
```

Análisis

Como sucede en el caso anterior con las listas enlazadas, las pilas también se conformarán a partir de datos *structs*, y esta vez se usará una estructura ya vista anteriormente, los *arreglos* (*arrays*).

Una característica particular que también notar es el hecho de que solo se utiliza una estructura para componer toda la estructura. Esto es debido a que en las pilas (y posteriormente con colas) se trabaja con elementos de un mismo tipo y que además estas se caracterizan por ser estructuras secuenciales que se encuentran juntas una tras otra, por lo que la comunicación será mucho más directa (gracias a l uso de una estructura predefinida) que como sucede con las listas enlazadas.

•Estructura Stack

```
7   type Stack struct {
8       element []int
9       top     *int
10  }
```

Type Stack struct: La estructura de una pila se puede entender como la composición de un arreglo **(para esta implementación)** definido *element []int*, y en este caso, se representarán los números enteros; y un puntero a un número entero *top*, que servirá para almacenar el Top de la pila.

De aquí en adelante se proponen las funciones de estructura, las cuales cumplen con una estructura de definición diferente a la creación de funciones (ver más en sección *6.* Mapas y Structs). Dichas funciones son:

IsEmpty(): Retorna un valor booleano verdadero o falso dependiendo de si la pila analizada está o no vacía. Esto se determina midiendo el tamaño del arreglo que compone a la pila, es decir, la característica propia *element []int*, usando la función predefinida de Go-lang *len(...)*, y comparando si el resultado es menor o igual cero.

Top(): Esta función retorna la dirección almacenada en la característica propia de la estructura de pilas *top*. Allí se encuentra la dirección de memoria del elemento que está situado en el tope de capacidad de la pila. En esta instrucción primero se determina si la lista está o no vacía con la función IsEmpty(), de lo que, si se genera verdadero se produce un mensaje en consola o terminal que indica que la pila está vacía, mientras que si se genera falso se retorna el elemento *top*.

setTop(): El nombre indica que cambia a *Top*, es decir, cambia la dirección del elemento situado en el tope de la pila, que está siendo almacenado por la característica propia *top*. Utiliza el parámetro *newTop* como un puntero a entero para poder almacenar el nuevo puntero con la dirección del nuevo elemento tope, para luego reasignarlo a *st.top*.

Size(): Indica el tamaño de la pila, es decir, el número de elemento contenidos en esta. Se desarrolla a partir de la función predefinida *len(...)* que analiza el arreglo que compone a una pila, *element []int*.

Push(): Esta función ejemplifica las operaciones realizadas para la inserción de un elemento dentro de una pila. Utiliza el parámetro *element* para almacenar el elemento a introducir, luego, mediante la función predefinida *append* utilizada con el arreglo propio de la pila *element []int*, concatena element con este último. Por último, se debe reasignar el nuevo Top, que para ello se utiliza *newTop* variable que almacena la dirección de la última posición del arreglo de la pila. Con ello se reasigna el nuevo tope usando la función *setTop*.

Nota: Se debe recordar que los índices de un arreglo y de un slice inician desde cero en adelante, mientras que el valor otorgado por la función *len(...)* inicia desde uno cuando hay elementos. Por lo anterior, se le resta uno a dicho resultado para obtener el intervalo adecuado de los índices del arreglo. Si esta pequeña configuración no se tuviese en cuenta, se produciría un error de indexaciones no encontradas.

Pop(): Esta función elimina el elemento situado en el tope de la pila. Primero verifica que la información almacenada en *top*, dada por *st.Top()* no sea una dirección nula. En un caso verdadero se almacena el tamaño de la pila en variable *newLen* que luego indicará en la reasignación del arreglo *element* el intervalo de los datos que se desean adquirir, que en ese caso siempre ira desde el principio (dejando vacío el espacio antes de ':') hasta el *newLen* restándola una unidad, lo que indica que dejarían por fuera al tope de la pila de ese momento.

Luego de la eliminación se reasigna el tope de la pila. El procedimiento es similar a la función anterior con la diferencia de que primero se debe determinar si el tamaño de la pila es cero para asignarle al nuevo tope una dirección nula ya que no hay elementos. Por otro lado, si el tamaño es distinto de cero, se reasigna el tope de la manera descrita anteriormente.

Print(): Esta función permite imprimir en pantalla los elementos que están almacenados en la pila. La impresión se hace a partir de un ciclo que se recorre un número de veces menor o igual al tamaño de la pila. Por cada ciclo se muestra el elemento situado en el tope de la pila, para luego eliminarlo por medio de la función pop(), y de esta manera continuar hasta que se recorra la pila el número de veces que se definió el ciclo.

```
12  func NewStack(size int) *Stack {
13      NewElement := make([]int, size)
14      var p *Stack = &Stack{}
15      if size == 0 {
16          p = &Stack{element: NewElement, top: nil}
17      } else {
18          p = &Stack{element: NewElement, top: &NewElement[size-1]}
19      }
20
21      return p
22  }
```

NewStack(): Esta es una función fuera de la estructura, es decir, de uso global. Permite la creación de nuevas pilas con un tamaño definido por *size*. Esta se desarrolla definiendo un nuevo arreglo con la función prediseñada *make(...)*. A su vez se crean instancias de la estructura *Stack* en forma de punteros. El tamaño definido por la función genera una condición que determina que cuando este tamaño es igual a cero, se usará el arreglo almacenado en *NewElement* y el puntero que almacena la dirección del elemento tope de la pila tendrá una dirección nula. Caso contrario el único cambio sería que *top* tendría la dirección del último elemento en el arreglo *NewElement*.

Nota: La necesidad de usar una función global para poder crear instancias de la estructura se da puesto la dificultad y tamaño de sintaxis necesaria para cada creación de una pila, por lo que el uso de esta función automatiza el proceso.

8.5.4 Pruebas sobre la estructura de pilas

Como se observa en código del apartado anterior, hay una función *main()* que contiene un conjunto de instrucciones que ejemplifican el comportamiento de una pila. Dichas instrucciones al ser compiladas generan los siguientes resultados:

```
>  Añadiendo elementos
   size ->  3
   El Top es ->  3
   3 ->2 ->1 ->
   Eliminando elementos
   Antes de eliminar, size 3
   después de eliminar, size 2
   El Top es ->  2
   2 ->1 ->
```

8.5.5 Colas

Es un grupo de elementos del mismo tipo los cuales, similares al caso anterior, son insertados en memoria a través de un extremo (proceso conocido como encolar) y eliminados a través de otro extremo llamado frente (proceso conocido como desencolar). Este sistema de manipulación de datos es conocido como sistema FIFO (First In First Out), es decir, el primero en entrar es el primero en salir.

Esta dinámica se puede entender mejor con casos de la vida cotidiana como las filas en los supermercados; sus procesos se centran en el momento en que va llegando la información, es decir, la persona que llegue de primera a una caja del supermercado será el primer elemento en salir. De esa manera sucesivamente con cada uno de los clientes que compongan las filas.

La representación del concepto anterior es la siguiente:

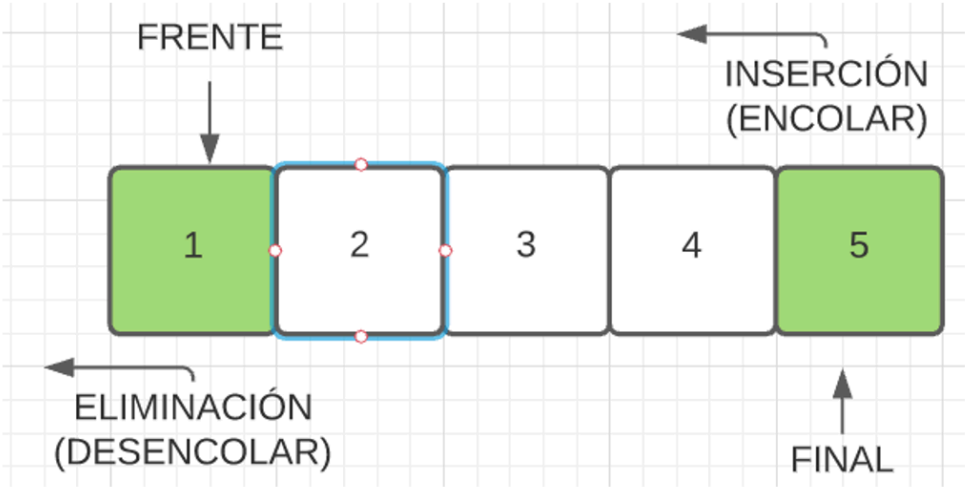


Ilustración 18. Representación gráfica de una cola

8.5.6 Operaciones fundamentales sobre colas

Se definen las siguientes operaciones fundamentales sobre las colas:

Operación	Concepto
Size()	Retorna el valor del tamaño de una cola
Enqueue()	Inserta un elemento por el extremo final de la cola, esto es, donde se sitúa siempre el último elemento dentro de un conjunto lineal.
Dequeue()	Remueve el Front(elemento del frente, es decir, situado en la primera posición) de la cola
isEmpty()	Retorna un valor booleano que determina si la cola analizada está o no vacía

Tabla 14. Operaciones fundamentales de las colas

8.5.7 Código que describe las colas

```
 1  package main
 2
 3  import (
 4       “fmt"
 5  )
 6
 7  type Queue struct {
 8       element []int // estudiar e tyoe assertion y el uso
    de interfaces
 9       front   *int
10  }
11
12  func NewQueue(size int) *Queue {
13       NewElement := make([]int, size)
14       var p *Queue = &Queue{}
15       if size == 0 {
```

```
16            p = &Queue{element: NewElement, front: nil}
17       } else {
18            p = &Queue{element: NewElement, front:
     &NewElement[0]}
19       }
20
21       return p
22  }
23
24  func (st *Queue) IsEmpty() bool {
25        return len(st.element) <= 0
26  }
27
28  func (st *Queue) Front() *int {
29        if st.IsEmpty() {
30             fmt.Println("El Queue está vacío")
31        }
32        return st.front
33  }
34
35  func (st *Queue) setFront(newFront *int) {
36        st.front = newFront
37  }
38
39  func (st *Queue) Size() int {
40        return len(st.element)
41  }
42
43  func (st *Queue) Enqueue(element int) {
44        st.element = append(st.element, element)
45        newTop := &st.element[0]
46        st.setFront(newTop)
47  }
48
49  func (st *Queue) Dequeue() {
50
51        if st.Front() != nil {
52             st.element = st.element[1:]
53        }
54
55        if st.Size() == 0 {
56             st.setFront(nil)
57        } else {
58              newFront := &st.element[0]
59              st.setFront(newFront)
60        }
61  }
62
63  func (st *Queue) Print() {
64       QueueSize := st.Size()
65       for index := 0; index < QueueSize; index++ {
66            fmt.Printf("%v ->", *st.Front())
```

```
67              st.Dequeue()
68          }
69
70          println()
71  }
72
73  func main() {
74          s := NewQueue(0)
75
76          fmt.Println("Añadiendo elementos")
77          s.Enqueue(1)
78          s.Enqueue(2)
79          s.Enqueue(3)
80          fmt.Println("size -> ", s.Size())
81          fmt.Println("El Top es -> ", *s.Front())
82          s.Print()
83
84          fmt.Println("Eliminando elementos")
85          s.Enqueue(1)
86          s.Enqueue(2)
87          s.Enqueue(3)
88          fmt.Println("Antes de eliminar, size", s.Size())
89          s.Dequeue()
90          fmt.Println("Despues de eliminar, size", s.Size())
91          fmt.Println("El Top es -> ", *s.Front())
92          s.Print()
93
94  }
```

Análisis

La estructura de una cola puede desarrollarse partiendo de la forma de una pila, con la diferencia de que las operaciones se harán en dirección opuesta. De manera similar, las colas también son una estructura secuencial en donde sus elementos se encuentran juntos entre sí, por lo que la comunicación entre estos es directa, evitando el uso de direcciones de memoria como en casos anteriores.

•Estructura Queue

```
7   type Queue struct {
8           element []int
9           front   *int
10  }
```

Type Queue struct: La estructura de una cola puede manejar las mismas características que una pila, que son el uso de un arreglo (y para esta representación será de números enteros) y un puntero que almacena la dirección del elemento situado al frente de la cola.

Al tomar como modelo de composición de las colas la forma de las pilas, el desarrollo de las funciones fundamentales entre este tipo de estructuras de datos es similar, sin embargo, se debe de notar ciertas diferencias:

Enqueue(), Dequeue() y NewQueue(): La primera encola un elemento dentro de la estructura mientras que la segunda desencola, es decir, elimina un elemento. La diferencia con respecto a su modelo de diseño radica en la reasignación del *Front*, que para este caso no se obtendrá la dirección del último elemento en el arreglo propio de la cola *element []int*, en su lugar se tendrá el primer elemento de este arreglo (en otras palabras, el elemento ubicado en *element[0]*).

8.5.8 Pruebas sobra la estructura de colas

Como se observa en el código del apartado anterior, hay una función *main()* que contiene un conjunto de instrucciones que ejemplifican el comportamiento de una cola. Dichas instrucciones al ser compiladas generan los siguientes resultados:

>	Añadiendo elementos
	size -> 3
	El Top es -> 1
	1 ->2 ->3 ->
	Eliminando elementos
	Antes de eliminar, size 3
	Despues de eliminar, size 2
	El Top es -> 2
	2 ->3 ->

8.6 Deque

También conocida como “Double ended Queue” o cola de doble terminación, es una estructura de datos que generaliza el funcionamiento de las pilas y las colas en una sola composición. Es una estructura lineal con un conjunto de elementos del mismo tipo los cuales son manipulados por ambos extremos de este conjunto, es decir, se pueden realizar operaciones de inserción y eliminación (dado su nombre también se conocen como Enqueue o Dequeue) tanto por el frente, es decir, donde se sitúa el primer elemento, como por el final, es decir, donde se sitúa el último elemento.

El funcionamiento y desarrollo de las operaciones de esta estructura es similar que los casos anteriores, ilustrándose en la siguiente representación:

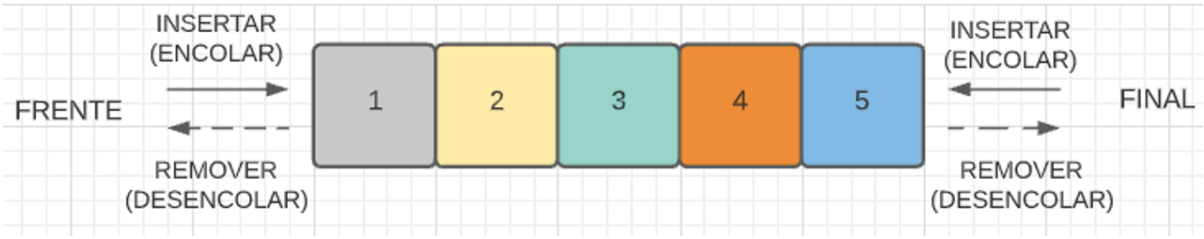


Ilustración 19. Representación gráfica de las Deque

8.6.1 Otros tipos de Deque

Estas estructuras se clasifican según la restricción que defina la dirección en que se manipula la entrada y salida de elementos.

- **Input Restricted Deque:** Estructura deque que permite la inserción de datos por un único extremo (ya sea frente o final) y la eliminación de datos por ambos extremos

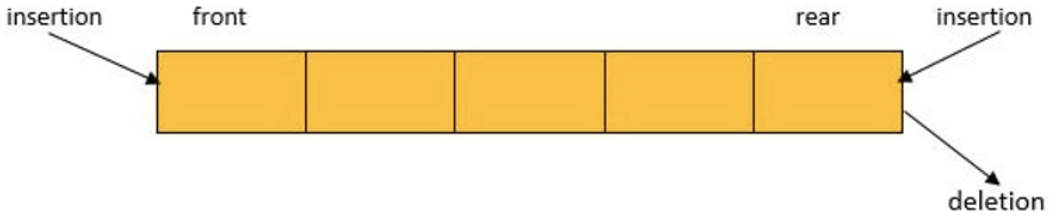


Ilustración 20. Inserción y eliminación de información en una Input Restricted Deque
Fuente: https://www.assignmenthelp.net/assignment_help/deque-data-structure

- **Output restricted Deque:** Estructura deque que solo permite la eliminación de elementos por un único extremo (ya sea frente o final) y la inserción de datos por ambos extremos

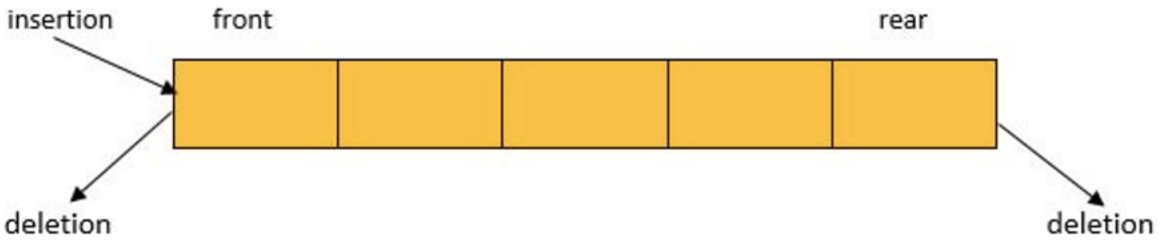


Ilustración 21. Inserción y eliminación de información en una Output Restricted Deque
Fuente: https://www.assignmenthelp.net/assignment_help/deque-data-structure

8.6.2 Algunas aplicaciones de las Deque

Este tipo de estructuras tienen cierta cantidad de aplicaciones, algunas de ellas son:

-**Operaciones de Hacer y deshacer,** aplicable en el desarrollo de software como por ejemplo los editores de texto
-**Almacenar y manipular** el historial de búsqueda de un navegador web
-**Implementar el "Steel job scheduling algortihm"** o algoritmo de periodizar el robo de trabajo. Es un algoritmo utilizado en sistemas informáticos con multiprocesadores que resuelve el problema de ejecutar dinámicamente los procesos dentro de una computadora.

8.6.3 Operaciones fundamentales sobre Deques

Se definen las siguientes operaciones fundamentales sobre las deque:

Operación	Concepto
Size()	Retorna el valor del tamaño de una deque
isEmpty()	Retorna un valor booleano que determina si la deque analizada está
EnqueueFront()	Inserta un elemento por el extremo inicial (frente) de una deque, esto es, donde se sitúa siempre el primer elemento dentro de un conjunto
DequeueFront()	Remueve el elemento situado en el extremo inicial (frente) de una
EnqueueLast()	Inserta un elemento por el extremo final de una deque, esto es, donde se sitúa siempre el último elemento dentro de un conjunto lineal.
DequeueLast()	Remueve el elemento situado en el extremo final de una deque.
SetFront()	Cambia el elemento situado en el extremo inicial (frente) de una
SetLast ()	Cambia el elemento situado en el extremo final de una deque.

Tabla 15. Operaciones fundamentales de las Deques

Al igual que en estructuras descritas anteriormente, el funcionamiento de inserción y eliminación de datos en una deque se puede describir en la siguiente imagen:

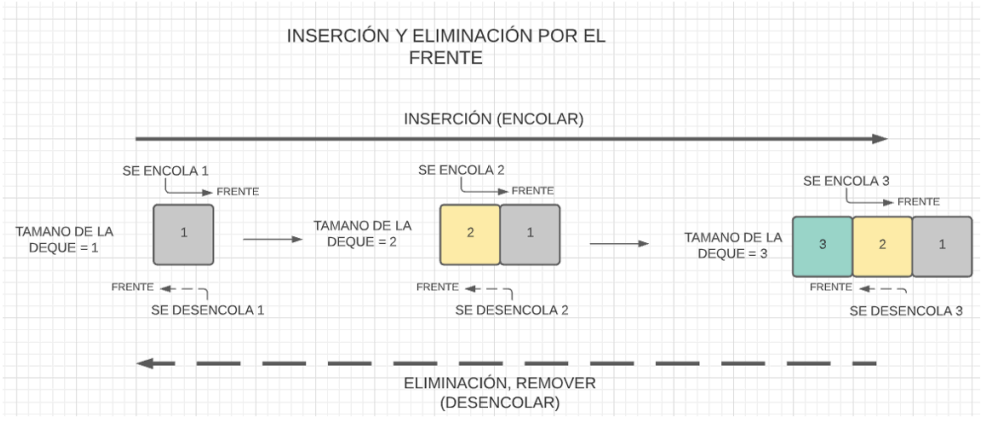


Ilustración 22. Representación gráfica de la inserción y eliminación de datos por el frente para una deque

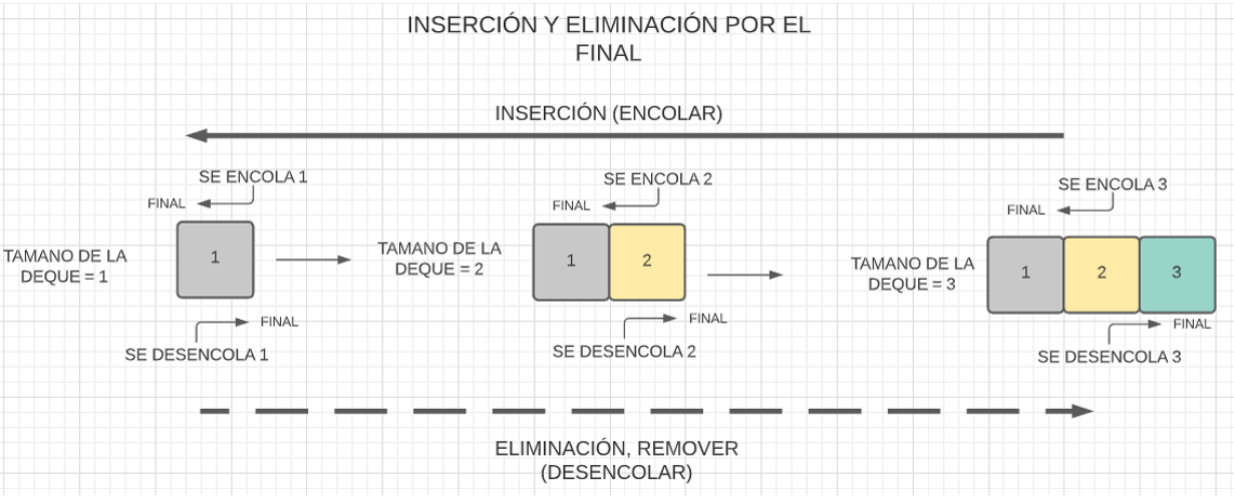


Ilustración 23. Representación gráfica de la inserción y eliminación de datos por el final para una deque

8.6.4 Código que describe las Deque

```
1  package main
2
3  import (
4        “fmt"
5  )
6
7  type Deque struct {
8       element []int // estudiar e tyoe assertion y el uso
   de interfaces
9       first   *int
10      last    *int
11 }
12
13 func NewDeque(size int) *Deque {
14      NewElement := make([]int, size)
```

```
15      var p *Deque = &Deque{}
16      if size == 0 {
17           p = &Deque{element: NewElement, first: nil, last:
18      } else {
19           p = &Deque{element: NewElement, first:
&NewElement[0], last: &NewElement[size-1]}
20      }
21
22      return p
23  }
24
25  func (dq *Deque) IsEmpty() bool {
26       return len(dq.element) <= 0
27  }
28
29  func (dq *Deque) First() *int {
30       if dq.IsEmpty() {
31            fmt.Println("El Deque está vacío")
32       }
33       return dq.first
34  }
35
36  func (dq *Deque) Last() *int {
37       if dq.IsEmpty() {
38            fmt.Println("El Deque está vacío")
39       }
40       return dq.last
41  }
42
43  func (dq *Deque) setFirst(newFirst *int) {
44       dq.first = newFirst
45  }
46
47  func (dq *Deque) setLast(newLast *int) {
48       dq.last = newLast
49  }
50  func (dq *Deque) Size() int {
51       return len(dq.element)
52  }
53
54  func (dq *Deque) EnqueueFront(element int) {
55       newElement := []int{element}
56       elementLen := len(dq.element)
57       for index := 0; index < elementLen; index++ {
58            newElement = append(newElement,
```

```
59        }
60        dq.element = newElement
61
62        if dq.Size() == 1 {
63             newFront := &dq.element[0]
64             dq.setFirst(newFront)
65             dq.setLast(newFront)
66        } else {
67             newFront := &dq.element[0]
68             dq.setFirst(newFront)
69        }
70  }
71
72  func (dq *Deque) DequeueFront() {
73        var settingVector []int
74        if !dq.IsEmpty() {
75             if dq.Size() <= 1 {
76                  dq.element = settingVector
77                  dq.setFirst(nil)
78                  dq.setLast(nil)
79             } else {
80                   dq.element = dq.element[1:]
81                   newFirst := &dq.element[0]
82                   dq.setFirst(newFirst)
83             }
84         }
85  }
86
87  func (dq *Deque) EnqueueLast(element int) {
88        dq.element = append(dq.element, element)
89        newLast := &dq.element[dq.Size()-1]
90        if dq.Size() == 1 {
91             newFirst := &dq.element[0]
92             dq.setFirst(newFirst)
93        }
94        dq.setLast(newLast)
95  }
96
97  func (dq *Deque) DequeueLast() {
98        if dq.Size() == 0 {
99             fmt.Print("Deque  vacía,  no  hay  elementos  para
100         } else {
101              if dq.Size() == 1 {
102                   var settingVector []int
103                   dq.element = settingVector
```

```
104                     dq.setFirst(nil)
105                     dq.setLast(nil)
106                 } else {
107                     dq.element = dq.element[:dq.Size()-1]
108                     newLast := &dq.element[dq.Size()-1]
109                     dq.setLast(newLast)
110                 }
111             }
112 }
113
114 func (dq *Deque) Print() {
115         DequeSize := dq.Size()
116         for index := 0; index < DequeSize; index++ {
117             fmt.Printf("%v ->", *dq.First())
118             dq.DequeueFront()
119         }
120
121         println()
122         println()
123 }
124
125 func main() {
126       s := NewDeque(0)
127
128         fmt.Println("Añadiendo elementos por el frente")
129         s.EnqueueFront(1)
130         s.EnqueueFront(2)
131         s.EnqueueFront(3)
132         s.EnqueueFront(4)
133         s.EnqueueFront(5)
134         s.EnqueueFront(6)
135         s.EnqueueFront(7)
136         fmt.Println("size -> ", s.Size())
137         fmt.Println("El Frente es -> ", *s.First())
138         fmt.Println("El ultimo es -> ", *s.Last())
139         s.Print()
140
141         fmt.Println("Eliminando elementos por el frente")
142         s.EnqueueFront(1)
143         s.EnqueueFront(2)
144         s.EnqueueFront(3)
145         s.EnqueueFront(4)
146         s.EnqueueFront(5)
147         s.EnqueueFront(6)
148         s.EnqueueFront(7)
```

```
149         fmt.Println("Antes de eliminar, size", s.Size())
150         s.DequeueFront()
151         s.DequeueFront()
152         s.DequeueFront()
153         fmt.Println("Despues de eliminar, size", s.Size())
154         fmt.Println("El Frente es -> ", *s.First())
155         fmt.Println("El ultimo es -> ", *s.Last())
156         s.Print()
157
158         fmt.Println("Añadiendo elementos por el ultimo")
159         s.EnqueueLast(1)
160         s.EnqueueLast(2)
161         s.EnqueueLast(3)
162         s.EnqueueLast(4)
163         s.EnqueueLast(5)
164         s.EnqueueLast(6)
165         s.EnqueueLast(7)
166         fmt.Println("size -> ", s.Size())
167         fmt.Println("El Frente es -> ", *s.First())
168         fmt.Println("El ultimo es -> ", *s.Last())
169         s.Print()
170
171         fmt.Println("Eliminando elementos por el ultimo")
172         s.EnqueueLast(1)
173         s.EnqueueLast(2)
174         s.EnqueueLast(3)
175         s.EnqueueLast(4)
176         s.EnqueueLast(5)
177         s.EnqueueLast(6)
178         s.EnqueueLast(7)
179         fmt.Println("Antes de eliminar, size", s.Size())
180         s.DequeueLast()
181         s.DequeueLast()
182         s.DequeueLast()
183         fmt.Println("Despues de eliminar, size", s.Size())
184         fmt.Println("El Frente es -> ", *s.First())
185         fmt.Println("El ultimo es -> ", *s.Last())
186         s.Print()
187
188 }
```

Análisis

El código que representa la estructura de una deque junta los elementos principales de los códigos de pilas y colas vistos anteriormente, estos son, componer una nueva estructura a partir de un arreglo que almacenará los elementos que se desean manipular, y dos punteros que identificarán los elementos situados en cada extremo de dicho arreglo. De la misma manera se desarrollan el resto de las funciones que definen el comportamiento de una deque.

•Estructura de una deque

```
7  type Deque struct {
8        element []int
9        first   *int
10       last    *int
11 }
```

Type Deque struct: Formada a partir del uso de estructuras, *Deque* toma como campos o características propias un arreglo *element []int* y dos punteros *first* y *last* respectivamente.

De aquí en adelante se proponen las funciones de estructura, las cuales cumplen con una forma de definición distinta a la creación de funciones (ver más en sección *6. Mapas y Structs*). Dichas funciones son:

NewDeque(): Esta función permite crear instancias de la estructura, y su razón de existencia se explica en la sección *8.4.3 Código que describe las Pilas*. Esta función utiliza como parámetro *size int* que almacenará el tamaño total de la estructura. Se crea un puntero nulo *p* que almacenará la dirección de la *deque* que se cree con el llamado de esta función y el arreglo que guarda cada elemento insertado, *NewElement*. Posteriormente, se determina los valores dependiendo de si el *size* tiene un valor igual o distinto de cero. Caso verdadero y el puntero con dirección nula se inicializa asignando *NewElement* como característica propia de las *deque*, es decir en *element*; de igual forma se realiza con los dos punteros propios *first* y *last* (que representan el "frente" y "final" respectivamente) quedando con direcciones nulas. Caso contrario se hacen las mismas asignaciones, con la diferencia de que *first* almacena la dirección del primer elemento y *last* almacena la dirección del último elemento, todos dos ubicados en *NewElement*. Al final de las instrucciones se retorna la variable *p*.

IsEmpty(): Retorna un valor booleano verdadero o falso si una deque está o no vacía respectivamente. Esto se determina a través del cálculo del tamaño del arreglo propio de una deque almacenado en *dq.element* en compañía de la función predefinida *len(...)*.

First() y Last(): Estas dos funciones retornan la dirección almacenada en los punteros propios de una deque, *first* y *last*, siendo estos el frente y final de esta última. Se resalta que antes de retornar cualquier información, primero se determina si la deque que utiliza esta función está o no vacía.

setFirst() y setLast(): Estas dos funciones como su nombre lo indica, cambian la información proporcionada por los punteros propios de una deque. Usan como parámetro *newFirst* y *newLast* respectivamente para almacenar la dirección del nuevo elemento a situar ya sea en el frente o en el final.

Size(): Retorna el tamaño de la deque analizada, el cual es calculado mediante el uso de la función predefinida *len(..)* en el arreglo propio de una deque *dq.element*.

EnqueueFront() y RemoveFront(): Estas dos funciones se encargan de insertar y eliminar elementos de la deque analizada, y más específicamente de hacerlo por el frente (o inicio) de la estructura. La explicación de cada una es:

-EnqueueFront(): Utiliza un parámetro *element* para almacenar el elemento que será insertado. Al saber que se está funcionando con arreglos, esta forma de definición de deque se encuentra con una situación particular y es la transición de posición de cada elemento almacenado en ese arreglo, esto puestos que al estar juntos entre si (desde el punto de vista de la memoria), el cambiar de posición un elemento que está en una primer índice con respecto a un arreglo de más elementos genera que cada uno se tenga que mover una posición demás (recordar que todo un arreglo en conjunto configura una memoria total de almacenamiento, a diferencia de lo que sucedía con las listas donde cada elemento se comportaba de forma independiente).

Con la situación anterior puesta en contexto se opta por utilizar siempre un vector auxiliar *newElement* el cual almacenará únicamente a *element*. Posteriormente por medio de un ciclo *for* que recorre a todo el arreglo propio de una deque *dq.element*, se concatena cada elemento de este último con el vector auxiliar, lo que permite tener el nuevo elemento en la posición inicial y a la vez recuperar el resto de los elementos de la estructura. Al finalizar el ciclo *dq.element* es reasignado por la información contenida en *newElement*. Para concluir con las instrucciones se deben de reasignar las direcciones que indican *first* y *last*, teniendo en cuenta la condición que se puede generar al tener deques de solo un elemento. Con ello se crea una condición que determina que cuando el tamaño de una deque es igual a uno, *first* y *last* estarán apuntando al mismo elemento, mientras que cuando el tamaño es distinto de uno el único elemento reasignado es *first* ya que el único cambio efectuado fue el nuevo primer elemento insertado.

-DequeueFront(): Al solo necesitar conocer la dirección del primer elemento en la estructura analizada, esta función puede eliminarlo, pero se debe tener en

cuenta una pequeña situación cuando el tamaño de la estructura deque cambia, esto es, cuando *dq.Size()* resulta menor o igual a uno. Si lo anterior se cumple, significa entonces que se han eliminado todos los elementos almacenados en la deque, y que por ende los punteros deberán almacenar una dirección nula. Por otro lado, si lo anterior es falso el único cambio generado es en la eliminación del primer elemento, por lo que la dirección afectada y que debe ser reasignada es la que se almacena en *first*.

Cabe resaltar que la eliminación de dicho elemento es gracias a la sintaxis proporcionada por los slices, donde *dq.element[1:]* indica la asignación de un nuevo arreglo que va desde el elemento ubicado en *dq.element[1]* hasta el final de este mismo.

EnqueueLast() y RemoveLast(): Estas dos funciones se encargan de insertar y eliminar elementos de la deque analizada, y más específicamente de hacerlo por el final (o último elemento) de la estructura. La explicación de cada una es:

-**EnqueueLast():** Esta nueva función a comparación de EnqueueFront() no tiene que usar un ciclo *for* para poder mover los datos almacenados en el arreglo propio *st.element*, en su lugar, al tener una inserción situada en el último elemento basta solo con concatenar un elemento al arreglo por medio de la función predefinida *append(...)*. Posterior a ello se debe reasignar la nueva dirección que indicara *last* teniendo en cuenta que cuando el tamaño de la deque analizada es igual a uno, *first* también debe ser reasignado.

-**DequeueLast():** Esta función elimina el elemento situado en *last* teniendo en cuenta el tamaño de la deque analizada. Si el tamaño mencionado es cero entonces se indica que la deque está vacía, por otro lado, si el tamaño es uno, el vector propio de la deque, es decir *st.element* queda vacío y los punteros *first* y *last* almacenan direcciones nulas. Sí el tamaño es distinto de uno, con la misma mecánica de la sintaxis de slices se resigna a *st.element* un nuevo vector que va desde el índice cero hasta el final del arreglo restándole una unidad, es decir, sin tener en cuenta a quien en ese momento se encuentre situado en *last*. De esto último se reasignan las direcciones y se termina la instrucción.

8.6.5 Pruebas sobre la estructura deque

A continuación, se presenta los resultados generados de la ejecución situadas dentro de la función *main*.

```
> Añadiendo elementos por el frente
  size ->  7
  El Frente es ->  7
  El ultimo es ->  1
  7 ->6 ->5 ->4 ->3 ->2 ->1 ->
  
  Eliminando elementos por el frente
  Antes de eliminar, size 7
  Despues de eliminar, size 4
  El Frente es ->  4
  El ultimo es ->  1
  4 ->3 ->2 ->1 ->
  
  Añadiendo elementos por el ultimo
  size ->  7
  El Frente es ->  1
  El ultimo es ->  7
  1 ->2 ->3 ->4 ->5 ->6 ->7 ->
  
  Eliminando elementos por el ultimo
  Antes de eliminar, size 7
  Despues de eliminar, size 4
  El Frente es ->  1
  El ultimo es ->  4
  1 ->2 ->3 ->4 ->
```

9. Referencias bibliográficas

GO, EL LENGUAJE DE PROGRAMACIÓN. ¡EN ESPANOL!. Nacho Pacheco, 2015-2016.

THE GO PROGRAMMING LANGUAGE, based ON A REPRESENTATION BY ROB PIKE (GOOGLE). Mooly Sagiv.

THE GO PROGRAMMING LANGUAGE. Aalan A. A. Donovan (GOOGLE INC), Brian W. Kernighan (Princeton University), 2016.

GET PROGRAMMING WITH GO. Nathan Youngman, Roger Peppé. 2019.

THE LITTLE GO BOOK. Karl Seguin. 2014.

GO BOOTCAMP. Matt Aimonetti. 2016.

Learning Go Programming. Vladimir Vivien. Packt, 2016.

Introducing Go: Build reliable and scalable programs. Caleb Doxsey. O'Reilly, 2016.

An introduction to programming in Go. Caleb Doxsey. 2012.

Data Structures in C++. Pat Morin.

9.1 Sitios de internet para programación con GO

DOCUMENTACIÓN OFICIAL DE GO Documentation - The Go Programming Language (golang.org)

TUTORIAL DE YOUTUBE POR FREECODECAMP.ORG (6) Learn Go Programming - Golang Tutorial for Beginners - YouTube