

# Estructuras de datos con Golang moderno

Isaac Julián Nieto Gallegos

December 31, 2025

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introducción</b>                                   | <b>2</b>  |
| 1.1      | El arte de la ciencia y la ciencia del arte . . . . . | 2         |
| 1.2      | Mi viaje ritual . . . . .                             | 2         |
| 1.3      | Como leer este trabajo? . . . . .                     | 3         |
| 1.4      | Ok, pero cómo logro esta última forma? . . . . .      | 4         |
| <b>2</b> | <b>Genéricos</b>                                      | <b>4</b>  |
| 2.1      | El mundo sin genéricos . . . . .                      | 4         |
| 2.2      | Un mundo con genéricos . . . . .                      | 8         |
| <b>3</b> | <b>Interludio: Preparando el entorno de trabajo</b>   | <b>10</b> |
| <b>4</b> | <b>Iteradores</b>                                     | <b>11</b> |
| 4.1      | Operador for-each . . . . .                           | 12        |
| 4.2      | Package iter . . . . .                                | 13        |
| <b>5</b> | <b>Colecciones</b>                                    | <b>15</b> |
| <b>6</b> | <b>Listas</b>   | <b>16</b> |
| 6.1      | Singly Linked Lists . . . . .                         | 17        |
| 6.1.1    | Nodo Simple . . . . .                                 | 17        |
| 6.1.2    | Esqueleto de la clase . . . . .                       | 18        |
| 6.1.3    | Intermedio: equals y toString en GoLang . . . . .     | 22        |
| 6.1.4    | Implementaciones de los métodos . . . . .             | 26        |
| <b>7</b> | <b>Ordenamientos</b>                                  | <b>29</b> |

# 1 Introducción

## 1.1 El arte de la ciencia y la ciencia del arte

En estos momentos que estoy escribiendo la introducción, soy un estudiante de Quinto Semestre de Ciencias de la Computación en la UNAM. Han pasado unos cuantos semestres desde que tomé el curso de Estructuras de Datos en la facultad, un curso que consideré clave en su momento para formar todos los conocimientos esenciales que iba a necesitar el resto de la carrera: es un curso denso que utilizando como recipiente lo que conocemos como estructuras de datos te presenta de manera discreta pero notoria -como la brisa de aire que sientes al llegar por primera vez a un lugar extranjero que no conocías- varios conocimientos y sutilezas que espero seguir utilizando el resto de mi vida como profesional: los algoritmos.

He visto en mi corta vida como computólogo como solemos pasar por alto el enfoque algorítmico de la profesión, sin darnos cuenta de que el análisis de algoritmos es lo que crea la parte artística de la computación. Y en los momentos en los que estoy escribiendo esto, justamente me estaba pasando esto.

Uno se deja absorber por las tecnologías que nos ofrece la computación moderna: frameworks que te permiten construir proyectos completos en cuestión de horas; metodologías “ágiles” que convierten el proceso de desarrollo de software en una cadena de comida rápida, donde se valora solamente la relación entre líneas de código escritas, tiempo invertido, y progresos logrados; agentes de IA que convierten el proceso de programación en una conversación con un ente todopoderoso que, bien aplicado, puede ser el equivalente a un equipo completo de becarios altamente entusiastas pero también tontos. Y en general, tecnologías que nos sirven para trabajar, pues evidentemente, en un mundo con un paso tan raudo y brutal como el nuestro, sería de tonto no rendirse a la producción en serie cuando el taller necesita competir con otras fábricas; pero que no poseen artesanía en su uso. La parte artística de la computación es muy fácil de ser olvidada, y justo eso es lo que me estaba pasando a mí: había dejado que el ritmo frenético de los proyectos universitarios y las herramientas “mágicas” opacaran el placer de entender, desde cero, cómo las estructuras de datos dan forma a los algoritmos con elegancia y precisión.

## 1.2 Mi viaje ritual

Para contrarrestar esto, comienzo este proyecto, en el cual voy a revisitar la materia que tanto me agració en sus inicios. Al puro estilo de uno de

los computólogos que más admiro, porque supo dar claramente con la parte artística de la computación y supo quedarse con ella: Donald Knuth. Knuth, con su monumental *The Art of Computer Programming*, no solo documentó algoritmos; los elevó a la categoría de literatura técnica, donde cada línea de código es parte de una narrativa mayor. Inspirado en esa filosofía, este documento adopta el enfoque de la programación literaria (*literate programming*), una técnica que él mismo pionerizó. Aquí, el código en Go no es un apéndice seco, sino un hilo tejido en el relato: explicaciones detalladas, decisiones de diseño y ejemplos ejecutables se entrelazan en un solo tejido narrativo, usando Org-mode como lienzo. Este no es un mero porting de estructuras de datos de otro lenguaje —aunque se inspira en el libro *Estructuras de Datos con Java Moderno* de Canek Peláez Valdez, reinterpretando sus conceptos para las idiosincrasias de Go: su simplicidad, concurrencia nativa y énfasis en la eficiencia idiomática—. Es un ritual personal de redescubrimiento. A lo largo de estas páginas, exploraremos desde lo básico (listas enlazadas, pilas, colas) hasta lo más sofisticado (árboles AVL, heaps binarios, grafos y hasta estructuras avanzadas como árboles de Fenwick y Segment Trees, que agrego por curiosidad propia). Cada sección no solo implementará el código, sino que reflexionará sobre por qué Go lo hace de manera única: ¿cómo la garbage collection de Go simplifica la gestión de memoria en árboles balanceados? ¿O cómo los canales y goroutines pueden potenciar grafos en entornos concurrentes? El objetivo es doble: para mí, es un ejercicio de artesanía computacional que me reconecta con las raíces algorítmicas de la disciplina; para ti, lector, espero que sea una guía clara y motivadora para dominar estructuras de datos en Go moderno, recordándonos que, en medio del caos tecnológico, siempre hay espacio para el arte de programar con intención y belleza.

### 1.3 Como leer este trabajo?

Este trabajo está escrito en org-mode con el paradigma de la programación lettrada, y como tal, puede ser leído de varias maneras:

La primera y la más fácil es en su forma de libro, la cual probablemente exista dentro de este repositorio. De esta forma , puedes leerlo como si directamente fuera un libro, con sus bloques de código como ejemplos del concepto de turno del que estemos hablando en ese momento. Esta forma definitivamente es la más rápida y directa de leer este trabajo.

Por otro lado, la mejor forma pero puede que la más difícil es la forma en la que también fue escrito: en un entorno Emacs adecuadamente configurado para tanglear y weavear este documento, lo cual convertirá este trabajo en

una pieza de lectura con bloques de código interactivos y ejecutables, al puro estilo de un Jupyter Notebook (plataforma interactiva que por cierto también nace como una forma de Programación Letrada), y además, en una pieza que generará todo el código fuente necesario para que estas estructuras de datos cobren vida como una librería con la que puedes trabajar y divertirte como si fuera una pieza de código tradicional.

#### 1.4 Ok, pero cómo logro esta última forma?

Sección pendiente jajaj

## 2 Genéricos

Muchas de las estructuras de datos que se van a revisar en este trabajo son **colecciones**, es decir, agrupaciones de elementos que permiten repeticiones (por lo que no son conjuntos).

Y nos interesa poder realizar estas operaciones de “guardar” y “retirar” elementos de este conjunto de una manera repetible y que nos garantice que funcionará sin importar el **tipo** de los elementos que guardaremos en el conjunto. Para dejar clara esta necesidad, coloquémonos en esta situación:

### 2.1 El mundo sin genéricos

Estamos desarrollando una pequeña aplicación para calcular distintos tipos de máximos. En particular vamos a pensar en dos tipos de máximos: el máximo entre dos números, y el máximo entre dos cadenas de texto.

Primero que nada, como que máximo entre dos cadenas de texto? Las cadenas de texto no poseen una “relación de orden” directamente notable que podamos usar, así que primero debemos de eliminar este nivel de ambigüedad. Diremos que pensamos en el orden lexicográfico, es decir, el orden alfabetico que usamos para guardar los libros en una biblioteca según su título.

Comenzaremos a implementar la primera, para quitarnos trabajo trivial de encima lo más rápido posible.

```
import "fmt"

func intMax(a int, b int) int {
    if a > b{
        return a
    }
}
```

```
}
```

```
    return b  
}
```

Y de paso creamos una pequeña prueba para ver que funciona correctamente.

```
func main() {  
    fmt.Println("El maximo entre 15 y 20 es: ", intMax(15,20))  
}
```

Para comprobar que todo funciona, veremos que si corremos el archivo, nuestra función trabaja bien.

```
go run miscs/intMax/intMax.go
```

Ahora, procedamos a hacer la otra función que nos falta.

Mi primer approach, y probablemente el que estés pensando tú también al leer este texto, sería implementar un orden lexicográfico a manita. Una cadena de pensamiento de la siguiente forma.

Como somos fans de las definiciones matemáticas, nos vamos a nuestra biblioteca (o libro digital) de confianza y comenzamos a buscar la definición formal de orden lexicográfico. Encontramos algo de este estilo:

Sean  $a, b$  cadenas vacías. Entonces un orden lexicográfico entre ellas se formaliza tal que:

$$\forall [a_1 \dots a_m], [b_1 \dots b_n] \in \Sigma^* : [a_1 \dots a_m] \leq [b_1 \dots b_n] \iff a_1 < b_1 \vee (a_1 = b_1 \wedge [a_2 \dots a_m] \leq [b_2 \dots b_n])$$

Inmediatamente observamos que esta definición se puede implementar de manera recursiva, pues tenemos los siguientes casos:

Caso Base: Una de las primeras letras de la cadena es menor a otra. Retornamos la cadena a la que pertenece la mayor y terminamos Paso recursivo: Las primeras letras de ambas cadenas son iguales, las retiramos y comparamos las que siguen

Para implementar esto de manera rápida lo haremos de manera recursiva, aunque está trivial hacerlo de manera iterativa.

```
package main  
import "fmt"  
  
func strMax(a string, b string) (string) {
```

```

if a[0] > b[0] {
    return a
}
if a[0] < b[0] {
    return b
}
return strMax(a[1:], b[1:])
}

```

Probemos si funciona

```

func main() {
    fmt.Println("La mayor entre las cadenas Alberto y Enrique es: ", strMax("Alberto", "Enrique"))
}

```

```
go run miscs/intMax/stMax.go
```

Existe, sin embargo, una forma más fácil. Y es que las strings en Golang implementan directamente la interfaz Ordered, y lo hacen con el orden lexicográfico, entonces en realidad podemos hacer solamente esto:

```

package main

import "fmt"

func strMax(a string, b string) (string) {
    if a > b {
        return a
    }
    return b
}

func main() {
    fmt.Println("La mayor entre las cadenas Alberto y Enrique es: ", strMax("Alberto", "Enrique"))
}

```

Lo cual, si la memoria no nos falla (y no ocupamos memoria, retrocede en el texto un poco) es **exactamente** el mismo código que usaremos para

comparar en la función intMax. Se nos antojaría entonces poder usar una misma función para ambos casos y así no escribir doble (Do Not Repeat Yourself)

El obstáculo que tenemos es que en ambas firmas necesitamos especificar los tipos que una función va a recibir, y qué nos va a retornar. Veamos ambas firmas:

```
func intMax (a int, b int) (int) {...}
func strMax (a string, b string) (string) {...}
```

Las firmas de nuestras funciones le están pidiendo mucho a los tipos que reciben, una está pidiendo directamente que sean enteros, otra que sean strings. Se nos antojaría poder reciclar el mismo código para cualquier tipo que implemente la interfaz necesaria, que nuestra función sólo le pida eso a sus entradas.

De la misma forma, cuando comenzemos a declarar nuestras estructuras de datos, pensemos por ejemplo en una Lista, nos encontraríamos con este problema análogo.

```
type ListInt struct {
    first *elementInt
    last *elementInt
    size int
}

type elementInt struct {
    value int
    prev *element
    next *element
}
```

Comenzamos a construir nuestra lista de enteros y nos encontraremos con el problema de que, bueno, sólo puede guardar enteros. Entonces, necesitaríamos construir también una lista de Strings, una lista de floats, una lista de MazdaMiata2005, y así por cada tipo que necesitemos, hasta el final de los tiempos.

Y como en realidad no estamos accediendo a ninguna propiedad especial de los tipos que guardamos, es decir, no vamos a guardar los enteros de una

manera distinta a la que guardamos las strings, pues todas estas estructuras en esencia serían la misma, solamente cambiando las declaraciones de sus elementos. Se nos antojaría que hubiera una mejor solución, no? Pues esos son los genéricos.

## 2.2 Un mundo con genéricos

Los genéricos son una característica que todo lenguaje de programación necesita. Su propósito principal es justamente permitir que las funciones, clases, y nuestras estructuras de datos, puedan funcionar con diferentes tipos de datos sin tener que reescribir el código para cada uno. En lugar de necesitar especificar un tipo de dato concreto y con eso pedirle mucho a nuestros códigos, podemos dejar estos “marcadores de posición” que generalizarán el tipo de dato. Este “marcador de posición” luego será manejado por nuestro compilador o intérprete para el tipo de dato que sea necesario durante la ejecución.

Los genéricos en Go existen desde la versión 1.18, y, al igual que en Java, se implementa en tiempo de compilación. Pero, contrario a Java, no se implementan usando borradura de tipos, se implementan usando monomorfización.

Lo que hace la monomorfización es crear una versión especializada del código genérico para cada tipo concreto que luego iba a usar este código genérico. Es decir, en esencia, el compilador de Go termina creando estas funciones especializadas por nosotros y las carga a nuestro binario por nosotros, haciéndolo de la siguiente manera:

1. Análisis: El compilador encunetra todas las invocaciones a código genérico
2. Generación de código: Por cada tipo concreto que se use (sea int, string, float64), el compilador va a generar una versión especializada del código genérico.
3. Sustitución: El compilador sustituye las llamadas al código genérico por llamadas al código especializado necesario que haya creado.

Este enfoque nos ofrece muchas ventajas y desventajas, como todo. La primera desventaja visible es que si el compilador está generando varias versiones especializadas de nuestro código genérico, eso evidentemente nos va a costar más peso en el binario final. Pero esa desventaja nos trae también la ventaja del rendimiento, pues, aunque el código es más amplio ahora, nos

ahorra hacer cualquier operación adicional en tiempo de ejecución, porque ya en los alambritos, el código que se llama en cada función es especializado y ya existente en el binario; es decir, todos los cálculos engorrosos respecto a los genéricos se hicieron en la compilación y ahí se quedaron.

Más aún, esta forma de implementación de los genéricos no tiene una forma de “tronar” como si la tiene la implementación por borradura de tipos de Java. Como el compilador se encarga por nosotros de realizar casi artesanalmente el código especializado que nos estamos ahorrando hacer nosotros, en esencia no estamos haciendo ningun compromiso por usar genéricos además del ya mencionado del tamaño del binario.

Pero qué pasaría si quisiéramos implementar genéricos en tiempo de ejecución en Golang como ya lo hacen lenguajes como Python? En general no habría mucho caso. Le estuve dando vueltas durante algo de tiempo a este asunto, y, fuera del compromiso que hacemos con el peso del binario (despreciable en hardware actual), la verdad es que esta implementación de Go está bastante bien alineada con lo que busca Go al ser un lenguaje compilado. Tal vez en lenguajes interpretados o en casos muy específicos podríamos añorar otra solución, pero en el caso general, la verdad es que esta implementación de genéricos es más que suficiente.

Con genéricos, nuestras dos funciones anteriores pasarían a ser una sola, tal que:

```
package main
import "fmt"
import "cmp"

func max[T cmp.Ordered] (a T, b T) (T) {
    if (a > b){
        return a
    }
    return b
}

func main() {
    fmt.Println("El maximo entre 15 y 20 es: ", max(15,20))
    fmt.Println("La mayor entre las cadenas Alberto y Enrique es: ", max("Alberto", "Enrique"))
}
```

Lo único que agregamos nuevo es `cmp.Ordered`, la cual es la interfaz

que deben implementar los tipos ordenables de Golang, es decir, los que se puedan operar con “`<,>`, `<=`, `>=`”

Los genéricos van a ser una herramienta con la que vamos a estar trabajando durante prácticamente todo este viaje, pues nos servirán para generalizar nuestras estructuras de datos, para poder guardar cualquier tipo en las mismas sin problemas.

### 3 Interludio: Preparando el entorno de trabajo

Antes de comenzar a ensuciarnos las manos construyendo esta biblioteca, debemos primero definirla para poder realizar una biblioteca utilizable por otros proyectos. Aunque dudo que realicemos un mejor trabajo que las implementaciones nativas (no por mucho, espero), creo que le daría una mayor formalidad a este trabajo el que pueda generar una biblioteca en condiciones, utilizable y todo.

El equivalente a Golang de las librerías son los módulos. Citando a la documentación de Golang:

A module is a collection of packages that are released, versioned, and distributed together. Modules may be downloaded directly from version control repositories or from module proxy servers.

A module is identified by a module path, which is declared in a `go.mod` file, together with information about the module's dependencies. The module root directory is the directory that contains the `go.mod` file. The main module is the module containing the directory where the `go` command is invoked.

Inicialmente, vamos a definir el módulo principal en donde vamos a estar agregando nuestros packages. Para esto, crearemos el archivo `go.mod` con el siguiente contenido:

```
module github.com/IsaacNietoG/goDataStructs

go 1.24.6
```

El nombre de un módulo en Go debe de darnos tanto la ruta para ser descargado como el nombre del mismo módulo en sí (y por extensión, lo que hace). De esta manera, los módulos en Go tienen esa virtud de que su mismo nombre también nos dice el lugar en donde lo podemos encontrar. Cosa que, si me preguntas, me parece algo bastante elegante y útil.

En el caso de este trabajo, lo podemos encontrar en mi repositorio de Github dedicado para este trabajo, dudo que en algún momento cambie esto, pero en ese caso tendríamos que realizar la modificación pertinente.

Como segunda línea, la versión para la que el módulo va a estar diseñado. Como en teoría no usaremos ningún módulo de terceros, no será necesario agregar un apartado de requires, ni cualquiera de los otros parámetros que puede contener este archivo. En dado caso que lo fuera, lo iremos tratando conforme avancemos.

## 4 Iteradores

Muchas de las estructuras de datos que vamos a implementar son **Iterables**. Que significa que son iterables?. El patrón de diseño Iterador nos dice que, si tenemos una estructura de datos que comprende varios elementos dentro de sí (sea colección, sea conjunto), esta misma estructura también nos debería de proporcionar una forma de recorrerla. Más formalmente:

El patrón de diseño iterador provee una forma para acceder a los elementos de un objeto de manera secuencial, sin exponer su representación subyacente. Define un objeto separado llamado iterador, el cual nos permite iterar sobre el iterable.

Este patrón de diseño nace con el propósito de no exponer la implementación interna del iterable, mientras nos permite recorrerlo e ir realizando acciones diversas sobre sus elementos.

Vamos a definir las especificaciones que queremos que cumplan nuestros **Iterables** mediante una interfaz.

Primero que nada, la firma de esta interfaz ya debe de implementar genéricos, pues su implementación nos debe de retornar “elementos” de algún tipo que evidentemente no conocemos en este momento, pues los conoceremos cuando comencemos a utilizar el iterable.

Esta, al igual que el resto de las interfaces que vamos a utilizar durante este trabajo, van a vivir bajo el directorio */interfaces* de nuestro módulo. Y, de una vez vamos a declarar el tipo.

```
package interfaces

type Iterable[T any] interface{
    Iterator() Iterator[T any]
}
```

Lo único que nos interesa de la interfaz Iterable, por el momento, es que justamente nos proporcione la garantía de que nos va a proporcionar un Iterador

Ya de paso, me permito señalar una nota sobre el comportamiento de Golang respecto a la privacidad de sus tipos. En Golang, el nombre de una función/atributo/tipo determina la privacidad del mismo hacia el mundo exterior. Si el susodicho empieza por mayúscula, entonces es público, y como tal, va a ser visible fuera del módulo. De esta manera, las funciones/atributos/tipos auxiliares o privados que necesitemos utilizar se verán implícitamente iniciados por letra minúscula. Lo digo de una vez, porque mientras escribimos los nombres de esta nuestra primera interfaz puede nacer esa duda.

Luego, definiremos la interfaz del Iterator. Un iterador, por convención, implementa los siguientes métodos:

- `hasNext`: Nos dice si existen más elementos “después” del elemento actual en el que está ubicado. El orden en el que se recorre una estructura puede ser trivial mientras hablamos de estructuras triviales como Listas, pero definir cuándo es que un elemento existe “después” de otro es más complicado cuando avancemos a estructuras como árboles.
- `next`: Avanza hacia el siguiente elemento en su recorrido, no sin antes retornarnos una referencia al elemento que acaba de pasar.

Entonces, la interfaz para Iterator quedaría de la siguiente forma:

```
type Iterator[T any] interface{
    hasNext() bool
    next() T
}
```

## 4.1 Operador for-each

Otra razón por la que nos interesa implementar iteradores es porque de aquí también nace luego el operador for-each.

El operador for-each es uno que ya conocemos, pues tiene su presencia en varios lenguajes de programación. Pero el más inmediato es obviamente Python. Este es un ejemplo del operador siendo usado.

```

frutas = ["banana", "manzana", "mango", "pera"]

for x in frutas:
    print(x)

```

En el caso de Golang, se usa de la siguiente forma, usando como ejemplo un Slice como estructura a iterar.

```

arreglo := [3]string{"Apple", "Mango", "Banana"}

for index,element := range arreglo {
    fmt.Println(index)
    fmt.Println(element)
}

```

Esto se implementa en el fondo justamente como una suerte de “azucar sintáctica” que reemplaza un bucle en el que llamamos al iterador y mientras haya un siguiente elemento, realizamos un acción. Entonces, nos interesa que nuestros iterables también sean compatibles con esto.

Es aquí donde usar Go nos va a cambiar un poco el paradigma respecto a esto. Porque para hacer que nuestras estructuras sean compatibles con este operador, tenemos que implementarlas de una forma distinta a como lo habríamos hecho en otros lenguajes. Por eso, el boceto de interfaces que hasta el momento habíamos hecho, no será el final (*jajaj, te cuentié mijx*). Es más, si te vas al código fuente, ni siquiera aparece.

Antes de Go 1.23, esta funcionalidad se habría tenido que implementar de alguna manera relacionada probablemente a *canales*, pero tenemos la suerte y buena fortuna de que justamente en esta versión se implementó finalmente la biblioteca *iter*. Esta biblioteca nos proporciona la capacidad de que nuestros iteradores funcionen con el bucle nativo *for range* que vimos anteriormente.

## 4.2 Package iter

El paquete nos provee de definiciones básicas y operaciones con las que vamos a poder implementar iteradores sobre secuencias en nuestras estructuras de datos. El paquete define los siguientes tipos de iteradores:

```
type (
```

```

Seq[V any]      func(yield func(V) bool)
Seq2[K, V any] func(yield func(K, V) bool)
)

```

Ambos iteradores están definidos como funciones que le van pasando elementos a una función generadora llamada “yield”. Esta función yield nos sirve como una suerte de hasNext(), que al igual que esta función, irá retornando true mientras haya más elementos que explorar, y false cuando llegue el momento de finalizar.

El iterador Seq devuelve un elemento, mientras que el iterador Seq2 regresa dos elementos a la vez. Pensaremos en utilizar Seq2 tal vez en diccionarios -para devolver llave y valor de un elemento a la vez- u otras estructuras en las que necesitemos que nuestra forma de iteración necesite devolver dos elementos a la vez.

De hecho, en el ejemplo de for-each de Go que vimos en páginas anteriores, el iterador que es utilizado es del tipo Seq2, pues al recorrer el slice, nos está regresando dos valores: el indice del elemento y el elemento en sí.

También debemos considerar que, para respetar las convenciones de nombrado de Golang, nuestros iteradores no van a ser devueltos por una función llamada Iterator, como habríamos hecho en otros lenguajes. En Golang, las funciones que retornan iteradores llevan el nombre del segmento de la estructura que recorren. Por ejemplo, para definir un iterador que recorra por completo una estructura, lo declararíamos de la siguiente forma:

```
func (s *Set[V]) All() iter.Seq[V]
```

Vamos a profundizar más en los iteradores conforme vayamos realizando sus implementaciones para cada estructura de datos que realicemos, por lo mientras, nos quedamos con esto. Pero antes de irnos, vamos a crear por nuestra cuenta una interfaz Iterable para hacer más legibles las definiciones de nuestras estructuras. Esta interfaz nos va a obligar a implementar la función All, que, como su nombre bajo la convención nos dice, nos proporciona un iterador que va a recorrer la estructura completa.

```

package interfaces

import "iter"

type Iterable[V any] interface{
    All() iter.Seq[V]
}

```

## 5 Colecciones

Casi todas, si no es que directamente todas las estructuras que vamos a implementar son colecciones... al intentar escribir esta parte del libro, y al intentar leerla, se antojaría definir lo que es una colección, pero de hecho acabo de tener un Deja Vu a mis clases de Álgebra Superior, donde nos dimos cuenta de que si quisiéramos dar una definición formal para conjunto/colección entonces estamos metidos en un grave problema relacionado con posibles definiciones circulares o vaguedad en las que no lo sean... entonces, para evitarnos este problema, vamos a definir nuestras colecciones no desde el significado real de esta palabra, si no desde el comportamiento que habríamos de esperar de una colección. Una colección debería de contener elementos, con los que podríamos realizar las siguientes operaciones:

- Agregar elementos.
- Eliminar elementos.
- Verificar si un elemento existe dentro de la colección.
- Saber si la colección está vacía.
- Obtener el número de los elementos de su interior.
- Limpiar la colección, es decir, despojarla de sus elementos.

Además, todas las estructuras que caigan bajo la definición de colección deberían de ser iterables. Es aquí donde resulta orgánico definirnos otra interfaz llamada Coleccion, la cual extienda a Iterable e implemente todos los métodos necesarios para cumplir con las funcionalidades que hemos platicado hemos de tener.

```
package interfaces

import "iter"

type Coleccion[V any] interface{
    Agrega(elemento V) (err Error)
    Elimina(elemento V) (err Error)
    Contiene(elemento V) (b bool, err Error)
    EsVacia() (b bool, err Error)
    GetElementos() (i int, err Error)
```

```
Limpia() (err Error)
All() iter.Seq[V]
}
```

Podrás notar que todos los métodos definidos siempre regresan un objeto Error, esto es debido a una característica particular de Golang relacionada al manejo de errores en Go. Por ahora no vamos a ahondar en ello, vamos a dejar simplemente que esto afecte en las declaraciones de los métodos, pero posteriormente se abordará de manera detenida.

También podemos señalar una característica importante de Golang respecto a la implementación de interfaces. En la mayoría de lenguajes de programación, para definir que una clase o tipo implementa una interfaz, es necesario hacerlo de manera explícita, con alguna palabra segura del lenguaje que nos permita escribir una declaración del tipo:

```
public interface Coleccion<T> extends Iterable<T> {
    ...
}
```

En cambio, en Golang, la implementación de una interfaz es implícita; es decir, un tipo debe definir todos los métodos definidos por una interfaz, y ya con esto se considera que dicho tipo (o interfaz en este caso) implementa a otro. Por eso, en la definición de interfaz de Colección dada previamente, la implementación de la interfaz Iterable viene implícita del hecho de que la interfaz pide implementar también el método All() iter.Seq[V]

Y finalmente, la definición de esta última interfaz nos deja preparados para comenzar a implementar la primera estructura de datos de este tour: Las Listas.

## 6 Listas

Si pensamos solamente en una “Lista” así nada más, con solamente esta palabra para buscar definirla, podemos acomodar dentro de esta definición a una amplia gama de estructuras de datos. Cuando hablamos de una “lista” más que hablar de una estructura de datos concreta, podemos pensar incluso en una categoría de estructuras de datos que suelen obedecer al concepto sencillo ejemplificable con el ejemplo cotidiano de “una lista de supermercado”.

La definición más débil y flexible de una lista es “una serie de elementos ordenados de manera consecutiva”, ya más adelante veremos que podemos hacernos muchos líos con esta definición tan sencilla y por eso digo que, a mi consideración, esto es más una categoría. En este capítulo, vamos a pasearnos por algunas de las estructuras de datos que se pueden considerar una “Lista”. Incluyendo los slices, la estructura de datos característica de Golang que puede entrar dentro de esta categoría y que veremos más adelante.

Y ya para terminar y ponernos manos a la obra, cabe decir que toda esta categoría de estructuras de datos puede y va a implementar la interfaz Colección que vimos en el capítulo anterior, porque esta interfaz engloba los comportamientos que esperamos de una Lista, incluyendo el comportamiento obvio de que debe de ser iterable, pues al tener elementos ordenados, podemos recorrerla de principio a fin con un iterador.

## 6.1 Singly Linked Lists

La forma de lista más sencilla en la cual podemos pensar para implementar es la Singly Linked List. Esta es la estructura de datos considerable como lista más primitiva en la que podemos pensar, pues fué implementada en 1957 en el lenguaje de programación IPL.

Para cubrir la definición más débil que vimos de una lista en la sección anterior, la forma más simple es hacer que cada elemento de una lista tenga una referencia o indicación de quién es su sucesor. De esta forma, como si fuera una serie de elementos amarrados por varios lazos, podemos tomar el primer elemento e ir recorriendo estos enlaces hasta llegar al último.

Antes de definir la estructura vamos a definir su unidad atómica: el nodo de lista.

### 6.1.1 Nodo Simple

Un nodo es para una lista lo que un eslabón es para una cadena (de las de acero que usamos para amarrar nuestras bicis); es esa estructura de datos auxiliar que vamos a usar para que cada uno de los elementos de nuestras listas tengan las referencias necesarias para que la lista sea recorrible. En esta primera estructura de Singly Linked Lists, este nodo se define de la siguiente manera:

```
type nodoSimple[V any] struct {
    elemento V
    siguiente *nodoSimple[V]
}
```

Un Nodo Simple (la forma en la que llamaremos el nodo de una Singly Linked List) engloba solamente dos cosas: el elemento que contiene y un puntero al siguiente Nodo Simple que sigue en la Singly Linked List.



Figure 1: Representación visual del Nodo Simple. Se almacena el elemento y el puntero al siguiente nodo.

Para respetar el principio de encapsulamiento, el Nodo Simple -al igual que cualquier abstracción interna que necesitemos para una estructura de datos- será una clase privada, en el sentido de que solamente va a ser usada dentro de la clase SinglyLinkedList, y ni su comportamiento ni funcionamiento será expuesto por arriba de la capa de abstracción que conforma la lista en sí.

Una vez dicho esto y definido el nodo simple, podemos comenzar a construir esta estructura de datos.

### 6.1.2 Esqueleto de la clase

```

package lists
import (
    "iter"
    "github.com/IsaacNietoG/goDataStructs/interfaces/coleccion"
)

<<definicionDeNodoSimple>>

type SinglyLinkedList[V any] struct{
    head *nodoSimple[V]
    tail *nodoSimple[V]
    size int
}

var _ Coleccion[any] = (*SinglyLinkedList[any])(nil)

```

La última linea de esta sección de código es un truco de Golang. Debido a que la implementación de Interfaces en Go es implícita, es muy fácil que se nos escape la implementación de algún método necesario para que una clase

pueda ser considerada implementación de una interfaz. Y es muy probable que esto nos lleve a errores posteriores en tiempo de ejecución, lo cual es altamente indeseable.

Con esta línea final podemos pedirle al compilador que nos “revise la tarea” pues lo que hace es:

- Intenta declarar una variable sin nombre (por eso el `_`), lo cual le da la pista al compilador de que no debe de guardarla en memoria
- La intenta declarar como una instancia de Colección.
- Le asigna el valor de un puntero hacia una SinglyLinkedList
- Esta SinglyLinkedList funcionalmente no existe, pues está siendo casteada desde `nil`, en realidad es un puntero fantasma solamente para intentar realizar la acción como tal.

Es buena práctica integrar esta verificación en nuestras clases de Golang, para detectar errores de manera temprana en el desarrollo y garantizar que sí estamos implementando las interfaces que queremos; y extender las clases que queramos también.

Luego declaramos los métodos necesarios para que la estructura implemente Colección

```
func (l *SinglyLinkedList[V]) Agrega(elemento V) (err Error){  
    <<implementacionAgregaSLL>>  
}  
  
func (l *SinglyLinkedList[V]) Elimina(elemento V) (err Error){  
    <<implementacionEliminaSLL>>  
}  
  
func (l *SinglyLinkedList[V]) Contiene(elemento V) (b bool, err Error) {  
    <<implementacionContieneSLL>>  
}  
  
func (l *SinglyLinkedList[V]) EsVacia() (b bool, err Error) {  
    <<implementacionEsVaciaSLL>>  
}
```

```

func (l *SinglyLinkedList[V]) GetElementos() (i int, err Error) {
    <<implementacionGetElementosSLL>>
}

func (l *SinglyLinkedList[V]) Limpia() (err Error){
    <<implementacionLimpiaSLL>>
}

func (l *SinglyLinkedList[V]) All() iter.Seq[V] {
    <<implementacionAllSLL>>
}

```

Así mismo, también vamos a agregar algunos métodos propios de la clase, que van a ser los siguientes:

```

func (l *SinglyLinkedList[V]) AgregaInicio() (err Error){
    <<implementacionAgregaInicioSLL>>
}

```

Por omisión, todos los elementos que agreguemos a una lista serán al final, pero también nos es conveniente tener un método por si queremos agregar un elemento al mero inicio de la lista.

```

func (l *SinglyLinkedList[V]) Inserta(indice int, elemento V) (err Error){
    <<implementacionAgregaInicioSLL>>
}

```

Queremos poder insertar un elemento en un lugar arbitrario, para eso es el método inserta, el cual dado un índice y elemento, inserta dicho elemento en dicho índice y recorre los consecuentes.

```

func (l *SinglyLinkedList[V]) EliminaPrimero() (elemento V, err Error){
    <<implementacionEliminaPrimeroSLL>>
}

func (l *SinglyLinkedList[V]) EliminaUltimo() (elemento V, err Error){
    <<implementacionEliminaUltimoSLL>>
}

```

Siempre podemos eliminar tanto el primer elemento de la lista como el último. Y de una vez podemos aprovechar para que estos métodos nos regresen el elemento que eliminan.

De la misma forma, a veces solamente queremos obtener estos elementos sin eliminarlos, para eso serán getPrimero y getLast

```
func (l *SinglyLinkedList[V]) GetPrimero() (elemento V, err Error){  
    <<implementacionGetPrimeroSLL>>  
}  
  
func (l *SinglyLinkedList[V]) GetUltimo() (elemento V, err Error){  
    <<implementacionGetUltimoSLL>>  
}
```

Luego, necesitamos un método para copiar la lista.

```
func (l *SinglyLinkedList[V]) Copia() (copia *SinglyLinkedList[V], err Error){  
    <<implementacionCopiaSLL>>  
}
```

Y obviamente vamos a querer un método para obtener el i-ésimo elemento de la lista.

```
func (l *SinglyLinkedList[V]) Get(i int) (elemento V, err Error){  
    <<implementacionGetSLL>>  
}
```

E inversamente, dada la referencia de un elemento (que sabemos existe en la lista) podemos saber el índice que ocupa en la misma con el siguiente método.

```
func (l *SinglyLinkedList[V]) IndiceDe(elemento *V) (i int, err Error){  
    <<implementacionIndiceDeSLL>>  
}
```

Debido a las raíces *Javaescas* de esta obra que tratamos en la introducción, me parece también correcto implementar equivalentes a los tradicionales métodos *equals()* y *toString()*. Para esto, vamos a hacer un pequeño paseo por la documentación y reglas de estilo de Golang para ver cuál es la mejor manera.

### 6.1.3 Intermedio: equals y toString en GoLang

1. Equals El método equals de una clase en Java nos permite revisar si dos objetos son iguales **a nivel semántico o de lógica de negocios**, es decir, más que darnos una verificación real de igualdad, nos da una verificación de **equivalencia**.

En Java esto es necesario porque si simplemente utilizamos el operador `==`, dos objetos que son equivalentes siempre nos van a dar un resultado negativo, porque **el operador == compara referencias, no los objetos como tal**. En Golang la cosa cambia, porque el operador `==` nos permite también comparar objetos.

La especificación de Go nos indica que podemos usar el operador `==` para comparar estructuras.

[...]Two struct values are equal if their corresponding non-blank field values are equal. The fields are compared in source order, and comparison stops as soon as two field values differ[...]

Pero de todas formas, esto no nos sirve para motivos de nuestras estructuras de datos; ya que, usando nuestro ejemplo de las listas, esta regla de comparación nos llevaría al caso extraño de que dos listas serían reconocidas como iguales por este comparador si tienen los mismos elementos de cabeza y cola, y la misma longitud, sin importar todos los elementos de en medio; obviamente esto no es correcto.

Y en Golang no existe una interfaz que exija implementar el método `equals`, ni tampoco un método `equals` por omisión como existe en otros casos. Golang en general nos da bastante flexibilidad respecto a este tema, incluso proveyéndonos de varios mecanismos para comparar estructuras que no voy a mencionar porque considero que se escapan del alcance de este texto.

Lo que podemos hacer es imitar a una estructura nativa de Golang que veremos más adelante: los slices.

En la implementación oficial de Slices de Go podemos encontrar dos métodos interesantes para este asunto: `Equal` y `EqualFunc`.

`Equal` recibe dos slices hechos de objetos genéricos que necesariamente implementan la interfaz comparable. Si implementan dicha interfaz, entonces estos objetos nos proporcionan una forma de -valga la redundancia- ser comparados. La implementación de `Equal` utiliza dicha forma para

comparar elemento por elemento del slice y retornarnos true si ambos slices tienen elementos “iguales”.

Si el slice está hecho de objetos genéricos no comparables, usaremos EqualFunc, la cual recibe tres cosas: los dos slices de objetos no comparables y una función comparadora que le proporcionaremos a EqualFunc para determinar si dos objetos de este estilo son “equivalentes” o no. Esto es similar a algo que aplicaremos más adelante cuando hablemos de 7, así que por ahora, guardemos esta idea en nuestra memoria y quedémonos con esto para el tema principal de este capítulo.

Usando estas dos funciones le damos flexibilidad a nuestra clase y un estilo bastante parecido a lo que ya existe en Golang, por lo que considero que respetamos el espíritu del lenguaje de programación en el que estamos trabajando.

Quedando de esta forma estas funciones:

```
func Equal[T comparable](l1, l2 *SinglyLinkedList[T]) (b bool, err Error){  
    <<implementacionEqualSLL>>  
}  
func EqualFunc[E1, E2 any](l1 *SinglyLinkedList[E1], l2 *SinglyLinkedList[E2], eq  
    <<implementacionEqualFuncSLL>>  
}
```

Es más, le dimos la suficiente flexibilidad a estas funciones que en realidad **no le estamos pidiendo a EqualFunc que las estructuras guarden los mismos tipos**, siempre que nos proporciona cualquiera que sea el método para comparar balones con globos, se puede realizar.

## 2. ToString

El método `toString` en Java se utiliza para retornar una representación escrita del objeto. Esta representación en su documentación se especifica como “concisa pero informativa que sea fácil de leer por una persona”. Si nos ponemos a investigar la documentación de Golang en busca de un método similar, podemos encontrar la interfaz `fmt.Stringer`. Esta interfaz pide implementar un método `String` el cual es el equivalente directo de un `toString` en Java, en el sentido de que es el que le indica a Golang cómo generar dicha representación escrita del objeto.

Entonces, vamos a agregar dicho método a nuestra estructura actual y a todas las subsecuentes que creemos.

```

func (l *SinglyLinkedList[V]) String() (s String, err Error){
    <<implementacionStringSLL>>
}

```

Con este último método, ya tenemos una lista de métodos que implementar para nuestra primera estructura de datos, quedando de la siguiente forma:

```

package lists
import (
    "iter"
    "github.com/IsaacNietoG/goDataStructs/interfaces/coleccion"
)

type nodoSimple[V any] struct {
    elemento V
    siguiente *nodoSimple
}

type SinglyLinkedList {
    head *nodoSimple[V]
    tail *nodoSimple[V]
    size int
}

var _ Coleccion[any] = (*SinglyLinkedList[any])(nil)

func (l *SinglyLinkedList[V]) Agrega(elemento V) (err Error){
}

func (l *SinglyLinkedList[V]) Elimina(elemento V) (err Error){
}

func (l *SinglyLinkedList[V]) Contiene(elemento V) (b bool, err Error) {
}

func (l *SinglyLinkedList[V]) EsVacia() (b bool, err Error) {
}

```

```

}

func (l *SinglyLinkedList[V]) GetElementos() (i int, err Error) {

}

func (l *SinglyLinkedList[V]) Limpia() (err Error){

}

func (l *SinglyLinkedList[V]) All() iter.Seq[V] {

}

func (l *SinglyLinkedList[V]) AgregaInicio() (err Error){

}

func (l *SinglyLinkedList[V]) Inserta(indice int, elemento V) (err Error){

}

func (l *SinglyLinkedList[V]) EliminaPrimero() (elemento V, err Error){

}

func (l *SinglyLinkedList[V]) EliminaUltimo() (elemento V, err Error){

}

func (l *SinglyLinkedList[V]) GetPrimero() (elemento V, err Error){

}

func (l *SinglyLinkedList[V]) GetUltimo() (elemento V, err Error){

}

func (l *SinglyLinkedList[V]) Copia() (copia *SinglyLinkedList[V], err Error){

```

```

}

func (l *SinglyLinkedList[V]) Get(i int) (elemento V, err Error){

}

func (l *SinglyLinkedList[V]) IndiceDe(elemento *V) (i int, err Error){

}

func Equal[T comparable](l1, l2 *SinglyLinkedList[T]) (b bool, err Error){

}
func EqualFunc[E1, E2 any](l1 *SinglyLinkedList[E1], l2 *SinglyLinkedList[E2], eq

)

func (l *SinglyLinkedList[V]) String() (s String, err Error)

```

#### 6.1.4 Implementaciones de los métodos

Una vez definidos los comportamientos que se nos antojaría que tuviera nuestra estructura de datos, procedemos a comenzar a implementarlos. Para implementar los comportamientos de una clase primero tenemos que entender claramente a nivel conceptual lo que queremos que nuestra clase sea y haga. Entonces, vamos a regresar al pizarrón para pensar en qué es lo que hace a una Singly Linked List una Singly Linked List.

La característica principal de este tipo de lista es que, como vimos previamente, cada uno de nuestros nodos solamente contiene una referencia a su siguiente nodo. De aquí proviene el nombre de *Singly* Linked List. La implementación más importante que genera este hecho es que solamente la podremos recorrer en una sola dirección: desde la cabeza hasta el rabo. No podemos recorrerla de reversa, porque una vez llegado a un nodo, no tenemos una ruta para regresar a su anterior. Visualmente esto se vería de la siguiente forma:

Visualmente podemos entonces comenzar a entender el mecanismo más general que vamos a utilizar para recorrer la lista; el cual es movernos referencia por referencia siempre hacia adelante, porque no tenemos forma de mirar atrás. Conceptualmente esto es sencillo de describir, como la mayoría

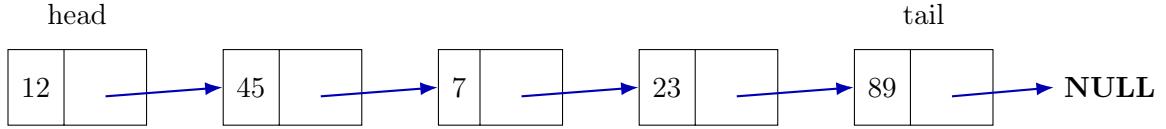


Figure 2: Representación de una lista enlazada simple con 5 elementos enteros.

de algoritmos necesarios para implementar estos comportamientos necesarios, sin embargo, van a ir naciendo ciertas particularidades cuando vayamos realizando las implementaciones.

Es importante señalar que los algoritmos existen como una entidad conceptualmente separada de sus implementaciones, un algoritmo puede ser implementado en varios lenguajes de programación, siempre que el programador responsable se encargue de ajustarse a las sutilezas del lenguaje en el que lo vaya a implementar.

En el libro del cual me inspiro para este texto, todos los algoritmos son vistos sin necesariamente ahondar en la implementación necesaria para que las estructuras vistas sean funcionales; sin embargo, para que la librería generada por este texto sea utilizable, es necesario que este texto sí se preocupe por las implementaciones de los algoritmos que veamos. La diferencia en este hecho reside en las filosofías de ambos textos y los resultados que buscan generar cada uno: uno es un libro con fines didácticos que busca que sus lectores se ensucien las manos para entender los conceptos, el otro (este mismo) es probablemente el soliloquio de un estudiante, que busca generar una biblioteca utilizable a través de programación letrada.

De todas formas, es importante también señalar que en muchos casos los algoritmos como tal son directamente replicados del libro que inspira este texto, excepto en los casos donde no lo sea, lo cual también será aclarado.

Entonces, en cada algoritmo siempre nos vamos a detener a ver las características de la implementacion decidida por el autor del texto (yo merengues).

1. Agrega Este método es para agregar elementos a una lista, por omisión se hace al final. Tenemos dos casos diferentes: cuando la lista es vacía, y cuando contiene al menos un elemento.

En cualquiera de los dos casos, las instrucciones en comúni consisten en construir un nuevo nodo  $n$  e incrementar el contador de elementos en la lista. Posterior a esto, realizamos la verificación necesaria para partir a uno de los dos casos, quedando hasta el momento de esta forma la

implementación:

```
n:= &nodoSimple[V]{elemento: elemento}
l.size++

if l.tail == nil {
<<caso1AgregaSLL>>
} else {
<<caso2AgregaSLL>>
}

return nil
```

Para verificar que la lista es vacía, podemos verificar sencillamente comprobando si el rabo *r* es 'nil'. Si esto se cumple, actualizamos cabeza y rabo a referencias de '*n*' y terminamos.

```
l.head = n
l.tail = n
```

Antes de continuar, decir que podemos suponer con seguridad que el puntero *siguiente* de un *nodoSimple* se inicializa en 'nil', mismo caso para los punteros *head* y *tail* de la estructura en sí. Esto es verdad porque en Golang se cumple que los atributos de tipo puntero ('\*T') se inicializan en 'nil', esto no es verdad para todos los atributos, pero concretamente para este caso sí lo es.

Si la lista no es vacía, entonces el rabo *r* es distinto de 'nil'; haremos que el siguiente del rabo sea '*n*' y actualizamos la referencia de tail a *n*.

```
l.tail.siguiente = n
l.tail = n
```

Así quedando la implementación final de este algoritmo:

```
n:= &nodoSimple[V]{elemento: elemento}
l.size++

if l.tail == nil {
    l.head = n
    l.tail = n
} else {
    l.tail.siguiente = n
    l.tail = n
}

return nil
```

2. Elimina

## 7 Ordenamientos