Algoritmos y Programación II

Ingeniería en Computación

UNTREF
UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Cátedra de Algoritmos y Programación II

Índice

1. Presentación

1.1 De qué trata la materia

En informática podemos entender a un algoritmo como una **secuencia de pasos que se deben ejecutar para realizar una determinada tarea**. Por ejemplo, existen varios algoritmos para ordenar una secuencia de elementos. Cada algoritmo de ordenamiento es una forma distinta de realizar la misma tarea. Algunos pueden ser más eficientes que otros, o más fáciles de escribir y mantener.

Los algoritmos se diseñan con una meta en particular, con un objetivo dado. Existen varias técnicas para diseñar y analizar algoritmos.

Los algoritmos no son exclusivos de la **informática**, sino que se utilizan también en otras disciplinas, estrechamente vinculadas a la computación, como las **matemáticas** y la **lógica**.

Cuando hablamos de **eficiencia**, en general nos referimos a la cantidad de recursos que necesita nuestro algoritmo para completar la tarea.

Un programa es una implementación de un algoritmo, escrita en un lenguaje de programación que la computadora es capaz de entender y ejecutar.

Los programas en general manipulan datos, toman datos como entrada, los procesan y generan salidas, ya sea en la forma de información desplegada en una pantalla o datos almacenados en un disco. Esta manipulación requiere que los datos estén organizados, lo que conocemos como **Estructuras de Datos**. Podemos pensar en las estructuras como bloques donde se almacenan y manipulan datos, y que pueden combinarse para generar estructuras más complejas.

¿Por qué es necesario estudiar algoritmos y estructuras de datos?

En principio, porque los algoritmos y las estructuras de datos son la base que permite a las computadoras realizar tareas cada vez más complejas. Por lo tanto, elegir las estructuras adecuadas y aplicar técnicas de diseño de algoritmos puede ser la diferencia entre obtener un programa eficiente o uno que no funcione bien o malgaste recursos. Por otro lado, su estudio fomenta el desarrollo del pensamiento lógico y abstracto, habilidades esenciales para profesionales de la computación.

Aprender a programar requiere **tiempo y esfuerzo**; es una tarea incremental que hay que llevar a cabo con constancia y paciencia. No alcanza con entender los conceptos teóricos o apropiarse de soluciones que desarrollaron otras personas. Aprender a programar se parece a aprender a tocar un instrumento musical. Al principio, hay que aprender las notas, las escalas y los acordes básicos, pero para convertirse en un experto se necesita practicar constantemente. Cada vez que tocamos una pieza, estamos fortaleciendo los dedos, mejorando la coordinación y desarrollando el oído musical. De la misma manera, al programar, cada línea de código que escribimos, cada error que encontramos y corregimos, y cada proyecto que completamos nos acerca más a convertirnos en mejores programadores. La clave está en **la repetición, la creatividad y la mejora continua**.

Cursada

El objetivo final de la materia es aprender las bases fundamentales de la programación, pero también desarrollar habilidades como el trabajo en equipo, el aprendizaje continuo y la comunicación efectiva.

Las clases se organizan alrededor de la práctica, por lo que es necesario que lean previamente los temas del día para no perder el ritmo. En clase se sintetizan los contenidos teóricos, se aclaran conceptos y nos ponemos manos a la obra. Los encuentros son híbridos, a veces virtuales y otras veces presenciales.

1.1.2.1 Evaluaciones

Para poder regularizar la cursada es necesario aprobar dos parciales; cada uno se aprueba con 4 o más puntos. Además, se debe presentar y aprobar un proyecto grupal que tendrá entregas parciales y una presentación final (oral y escrita).

Para aprobar la materia además hay que aprobar un examen final. Para poder rendirlo es necesario tener aprobada Algoritmos y Programación I.

Lenguaje de programación

Un lenguaje de programación es lo que nos permite comunicarnos con la computadora para que realice alguna tarea. Cada lenguaje de programación tiene su propia:

Sintaxis Reglas gramaticales para escribir instrucciones válidas y que la computadora pueda comprender.

Semántica Significado de cada instrucción.

Algunos lenguajes de programación, al igual que los idiomas hablados, permiten expresar mejor algunos conceptos que otros, por eso, para elegir un lenguaje de programación hay que tener en cuenta varios aspectos, como la naturaleza del proyecto y la experticia del equipo de trabajo, entre otros.

Teniendo en cuenta que esta materia es parte de la formación de ingenieras/os en computación, cuyo enfoque está en los sistemas integrados de *hardware* y *software*, elegimos Go como lenguaje por ser moderno, fácil de aprender y, a su vez, más cercano al *hardware* que otros lenguajes de alto nivel.

2. Taller de Go

2.1 Introducción a Go

Características principales

Go es un lenguaje de programación desarrollado por Google y lanzado oficialmente en 2009. Es un lenguaje compilado, es decir, una vez escrito el código fuente se debe traducir a código máquina para poder ejecutarlo; esta operación de traducción se conoce como compilación.

Figura 2.1: Logo de Go.

Los creadores de Go, Robert Griesemer, Rob Pike y Ken Thompson, han dicho que aunque la sintaxis del lenguaje está inspirada principalmente en C y en Python, y en menor medida en Java, el objetivo siempre fue crear un nuevo lenguaje simple y eficiente. Go fue diseñado para ambientes altamente productivos y concurrentes (es decir, donde varios programas se ejecutan al mismo tiempo y comparten recursos). Fue liberado como código abierto y está disponible para todos los sistemas operativos.

Atención

Los lenguajes compilados cuyo código fuente se traduce de antemano a código máquina, en general suelen ser muy eficientes, ya que se pueden ejecutar directamente sobre la máquina sin "intermediarios".

Go es un lenguaje **fuertemente tipado** y con **tipado estático**, es decir que al momento de compilar, debe estar claramente establecido de qué tipo son sus variables. Por lo tanto, cuando escribimos código, al declarar una variable se debe declarar de qué tipo es.

Fuertemente tipado significa que no se pueden realizar operaciones entre distintos tipos de datos que no están previamente establecidas por el lenguaje o el programador. Las conversiones entre distintos tipos se deben realizar explícitamente, por ejemplo, si queremos sumarle a un número entero un número en decimal (en punto flotante) primero debemos convertir el número en punto flotante a entero y así el resultado será otro entero.

Tipado estático significa que el tipo de una variable se determina en el momento de escribir el código y no puede cambiar durante la ejecución del programa.

Sistema de Tipos

Un sistema de tipos permite definir qué valores puede tomar una variable y qué operaciones se pueden hacer con esos valores.

2.1.1.1 Gestión de memoria

La gestión de memoria se refiere a cómo se reserva espacio para las variables y cómo se libera ese espacio cuando ya no se necesitan dichas variables. En Go, este proceso es **automático**, por lo que el programador no necesita pedir ni liberar memoria explícitamente (al estilo de `malloc` y `free` en C).

2.1.1.1.1 Reserva de memoria: Stack vs Heap

Go utiliza principalmente dos áreas de memoria para almacenar datos:

- **Pila (*Stack*):** Es una memoria muy rápida donde se almacenan las variables locales de las funciones y los parámetros. El espacio se reserva al declarar la variable y se libera automáticamente e instantáneamente cuando la función termina.
- **Montículo (*Heap*):** Es una región de memoria más grande que se utiliza para datos que deben persistir más allá de la ejecución de una función o que tienen un tamaño dinámico (como *slices* o mapas), es decir que el tamaño no se conoce al momento de declarar la variable.

Para reservar memoria, el programador simplemente declara la variable; el compilador e incluso el ambiente de ejecución (*runtime*) se encargan de asignar el espacio necesario:

Go

```go
var edad int                 // Reserva automática para un entero
nombres := make([]string, 0) // Reserva dinámica para un slice
```

El compilador decide automáticamente dónde colocar cada variable mediante un proceso llamado **análisis de escape** (*escape analysis*). Si detecta que una variable local "escapa" de su función (por ejemplo, si se devuelve un puntero a esa variable o se guarda en una estructura global), la moverá automáticamente al *heap*.

2.1.1.1.2 Liberación de memoria: Garbage Collector

Para liberar la memoria reservada en el *heap*, Go utiliza un **recolector de basura** (*Garbage Collector* o GC). El GC se ejecuta de forma periódica realizando las siguientes tareas:

1. **Rastreo:** Identifica qué objetos en el *heap* ya no son accesibles desde ninguna parte activa del código.
2. **Barrido:** Libera el espacio que ocupaban esos objetos para que pueda ser reutilizado.

Este enfoque evita errores críticos como las **fugas de memoria** (*memory leaks*) —olvidar liberar la memoria que reservamos— o el acceso a **punteros colgantes** (*dangling pointers*) —intentar usar memoria que el programa ya liberó—.

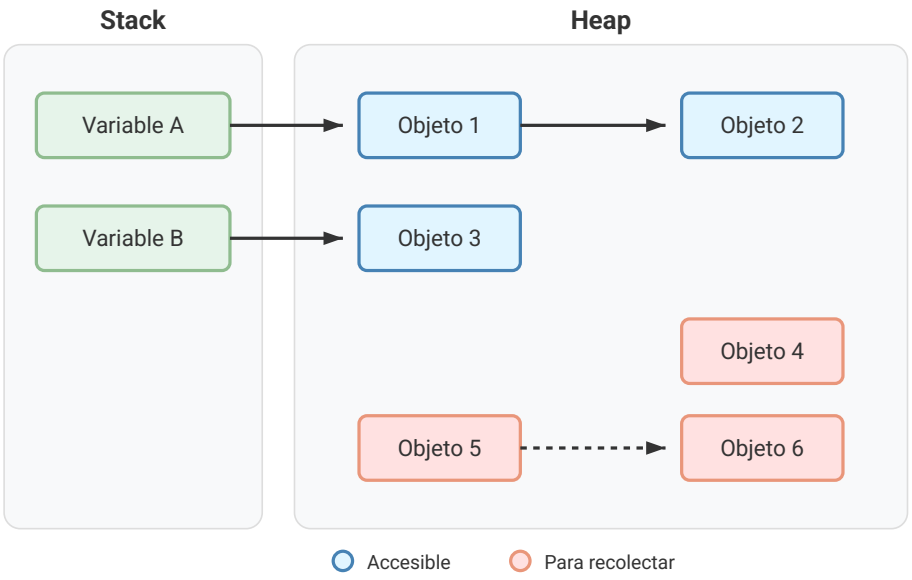


Figura 2.2: Funcionamiento del Garbage Collector: rastreo y liberación de objetos inalcanzables.

En la imagen los objetos 4, 5 y 6 ya no son referenciados por ninguna variable, por lo tanto el GC los eliminará en la próxima ejecución.

2.1.1.2 Orientación a objetos

Go no es un lenguaje orientado a objetos, es decir, no hay clases ni objetos como en Java, por ejemplo, sino que utiliza `struct`, a la C, lo que nos permite definir nuevos tipos de datos.

Go

```go
type Persona struct {
    Nombre string
    Edad   int
}
```

Define una estructura de datos llamada `Persona` con dos campos: `Nombre` de tipo `string` y `Edad` de tipo `int`.

Go

```go
var p Persona
p.Nombre = "Fabián"
p.Edad = 32
```

`p` es una variable de tipo `Persona` y podemos acceder a sus campos utilizando el operador punto (`.`), en este caso `p.Nombre` tiene el valor `"Fabián"` y `p.Edad` tiene el valor `32`.

En Go no existe la herencia, pero sí existe la composición, que nos permite crear nuevos tipos de datos a partir de otros. En el capítulo Estructuras e interfaces vamos a profundizar en las estructuras y veremos cómo funciona la composición en Go.

2.1.1.3 Ejemplos

En el repositorio taller-go encontrarán código de ejemplo para empezar a sumergirnos en Go.

El siguiente fragmento se encuentra en el archivo `01-introduccion/ejemplos/00-hola/main.go`

Go

```go
package main

import "fmt"

func main() {
    fmt.Println("¡Hola mundo!")
}
```

Si ejecutamos en una terminal:

console

```
go run main.go
```

obtendremos como resultado:

output

```
¡Hola mundo!
```

Aquí vemos un ejemplo de una función simple que recibe dos argumentos de tipo `int` y devuelve un nuevo valor de tipo `int`.

Go

```go
func sumar(a, b int) int {
    return a + b
}
```

Para utilizar dicha función solo debemos invocarla por su nombre y proporcionarle los parámetros requeridos.

Go

```go
fmt.Println(sumar(32, 7))
```

output

```
39
```

Instalación

El sitio oficial de Go es https://go.dev/ de donde se puede descargar versiones listas para instalar o el código fuente para compilar e instalar. Se recomienda seguir las instrucciones. Una vez finalizado el proceso se puede verificar la correcta instalación, abriendo una terminal y ejecutando el siguiente comando:

console

```
go version
```

Si la instalación fue exitosa, deberíamos ver una salida como la siguiente:

output

```
go version go1.24.0 linux/amd64
```

2.1.2.1 Go Playground

Go ofrece un servicio *online* llamado Playground https://go.dev/play que nos permite escribir y ejecutar fragmentos de código de forma simple y sin necesidad de instalar Go localmente.

Ejercicios

Los ejercicios de este capítulo están en `01-introduccion-go/ejercicios/` del repositorio taller-go.

Repositorio del taller de Go

El siguiente repositorio es el material de trabajo para este taller de Go. Está organizado por temas (intro, funciones, structs, etc.) y dentro de cada tema hay dos carpetas:

- **`ejemplos/`**: programas completos y fragmentos que ilustran los conceptos explicados en el apunte. Sirven para seguir la teoría en la práctica y experimentar.
- **`ejercicios/`**: esqueletos de código con partes incompletas que se corresponden con los ejercicios propuestos a lo largo de este taller. La idea es que completen esos archivos como práctica.

El repositorio se crea automáticamente al aceptar la asignación en Classroom50. Una vez aceptada, tienen acceso a todo el contenido en su máquina local.

Links útiles

- Página principal de Go
- Descargar Go
- Tour interactivo de Go
- Documentación basada en ejemplos
- Go FAQ
- Go Playground (entorno online para ejecutar código en Go)

2.2 Paquetes y módulos

Paquetes

Un **paquete** es una colección de archivos que contiene código fuente, definiciones de constantes, definiciones de tipos, etc. Todos los archivos de un paquete se encuentran en un mismo directorio.

Los paquetes organizan el código en forma lógica. Las variables, funciones y tipos de datos definidos dentro de un paquete son privados del mismo, es decir, no se pueden utilizar afuera, a menos que se exporten explícitamente.

Además de que todos los archivos estén en el mismo directorio, cada archivo que forma parte de un paquete debe declarar a qué paquete pertenece:

Go

```go
package mipaquete
```

Módulos

A su vez una colección de paquetes pueden conformar un **módulo**. Un módulo es la unidad fundamental de organización y distribución de código. Los módulos permiten gestionar dependencias, compartir código y versionarlo.

Un módulo se define en una carpeta que contiene un archivo `go.mod`. En este archivo se especifica el nombre del módulo, la versión de Go con la que se desarrolló y las dependencias de otros módulos.

Todos los paquetes (en general cada paquete en una subcarpeta) que están en la misma carpeta que contiene el archivo `go.mod` son parte del mismo módulo. En general un módulo corresponde a un proyecto.

2.2.2.1 Gestión de módulos

Para crear un módulo se usa el comando `go mod init` dentro de la carpeta que contendrá el proyecto, seguido del nombre del módulo. Usualmente el nombre del módulo es el repositorio donde se aloja. Ejemplo:

console

```
go mod init github.com/untref-ayp2/miproyecto
```

Este comando creará un nuevo archivo llamado `go.mod` con el siguiente contenido:

text

```
module github.com/untref-ayp2/miproyecto

go 1.24.1
```

Todos los archivos que se encuentren dentro de la misma carpeta en la que se haya creado `go.mod` son los que serán considerados parte del módulo `github.com/untref-ayp2/miproyecto`

text

```
miproyecto
├── otropaquete
│   ├── estructuras.go
│   └── estructuras_test.go
├── unpaquete
│   ├── utils.go
│   └── utils_test.go
├── go.mod
├── go.sum
└── main.go
```

2.2.2.2 Importar paquetes

El objetivo final de organizar nuestro código en paquetes y módulos es que podamos reusarlos en otros proyectos. Cada vez que necesitemos usar una dependencia tanto interna a nuestro módulo como externa, debemos usar la instrucción `import`, lo que nos permite acceder a las estructuras, funciones y utilidades de los distintos módulos.

En Go existen distintos paquetes que podemos importar, dependiendo de su procedencia. Por ejemplo, Go provee una serie de paquetes que son parte de la biblioteca estándar del lenguaje, como pueden ser:

`fmt` entrada y salida por pantalla y teclado
`math` constantes y funciones matemáticas
`time` funciones para medir y mostrar tiempos
`errors` funciones para manipular errores
`path` funciones para manipular carpetas y archivos
`testing` soporte para test automáticos
`strings` funciones para manipular cadenas de caracteres

Hasta ahora hemos utilizado mayormente este tipo de paquetes. También podemos utilizar referencias a paquetes dentro de nuestro mismo módulo.

Go

```go
package main
```

```go
import "github.com/untref-ayp2/miproyecto/saludo"

func main() {
    saludo.Saludar()
}
```

La ruta completa del import (`github.com/untref-ayp2/miproyecto/saludo`) indica que debemos importar el paquete `saludo`, que es un subpaquete de nuestro módulo. El nombre del paquete (`saludo`) es la última parte de la ruta y es la referencia que usamos para acceder a sus miembros exportados.

Otros tipos de módulos que podemos importar son los módulos de otros programadores, por ejemplo, podríamos pensar en utilizar un supuesto módulo de algoritmos de ordenamiento de la cátedra. Entonces, imaginemos que existe el módulo llamado `"untref.edu.ar/ayp2/busqueda"`. Podemos importarlo de la siguiente manera.

Go

```go
package main

import (
    "fmt"
    "untref.edu.ar/ayp2/busqueda"
)

func main() {
    lista := [10]int{2, 3, 9, 5, 1, 8, 4, -5, -4, 6}
    busqueda.Quicksort(lista)
    fmt.Println(lista)
}
```

El resultado será:

output

```
[-5 -4 1 2 3 4 5 6 8 9]
```

Hay varios puntos a notar:

- Siempre se importan **paquetes**.
- Cuando importamos un paquete, el texto luego de la última `/` será el nombre de la referencia que podemos utilizar para acceder a los miembros del paquete, que además coincide con el nombre del paquete.
- Podemos renombrar la referencia que se crea al importar un paquete, ya sea por conveniencia como en el caso de que haya un conflicto de nombres entre paquetes de distintos módulos.

Go

```
import b "untref.edu.ar/ayp2/busqueda"
```

Luego podremos utilizar la nueva referencia `b` en lugar de `busqueda`.

Dependencias

Cuando necesitamos utilizar un módulo de terceros, como podría ser un módulo que provee distintos métodos de ordenamiento, primero debemos encontrar el módulo, esto puede ser haciendo una búsqueda en el registro de go que podemos encontrar en https://pkg.go.dev/ o bien, buscando módulos en internet. Es posible instalar módulos directamente desde repositorios de Git.

Para instalar y agregar dicha dependencia en nuestro módulo, podemos utilizar la herramienta `go get` de la siguiente forma:

console

```
go get github.com/untref-ayp2/busquedas
```

Una vez ejecutado este comando, el archivo `go.mod`, donde se encuentra la definición de nuestro módulo, se verá similar a:

text

```
module github.com/untref-ayp2/miproyecto

go 1.24.1

require github.com/untref-ayp2/busquedas v0.1.0
```

De esta forma, si estamos trabajando con más personas en el mismo proyecto, o si alguien más quiere utilizar nuestro módulo como dependencia en su propio proyecto, será posible respetar los requerimientos que cada módulo en el proyecto necesita que se cumplan.

Si en lugar de instalar y agregar la dependencia con `go get`, tenemos una forma alternativa que es, primero usar la dependencia en nuestro código y luego ejecutar `go mod tidy`. Veamos un ejemplo.

Supongamos que creamos un módulo e importamos algunas cosas en nuestro archivo `main.go`.

2.2.3.1 Ejemplo de creación de un módulo simple

Primero creamos una carpeta con el nombre `mimodulo` y nos posicionamos en ella.

console

```
mkdir mimodulo && cd mimodulo
```

Inicializamos el módulo.

console

```
go mod init github.com/untref-ayp2/mimodulo
```

output

```
go: creating new go.mod: module github.com/untref-ayp2/mimodulo
```

console

```
cat go.mod
```

output

```
module github.com/untref-ayp2/mimodulo

go 1.24.1
```

Usando cualquier método que les resulte conveniente creamos el archivo `main.go` con el siguiente código.

console

```
cat main.go
```

Go

```go
package main

import (
    "fmt"
    "github.com/fatih/color"
)

func main() {
    msj := color.RedString("¡Texto en rojo!")
    fmt.Println(msj)
}
```

Ejecutamos nuestro programa, pero vemos que hay un error.

console

```
go run main.go
```

output

```
main.go:5:2: no required module provides package github.com/fatih/color; to add it:
    go get github.com/fatih/color
```

Podemos notar que los mensajes de error de Go muchas veces (no siempre) son muy descriptivos y hasta nos pueden sugerir la solución si prestamos especial atención a lo que nos dice.

Si bien en este ejemplo, queremos mostrar una forma alternativa de solucionar la falta de la dependencia, `go get` es otra forma de arreglar el problema que, intencionalmente, acabamos de crear.

Para ello usaremos `go mod tidy`, si consultamos la ayuda podemos aprender qué es lo que este comando realmente hace con nuestro módulo.

console

```
go help mod tidy
```

output

```
[...]

Tidy makes sure go.mod matches the source code in the module.
It adds any missing modules necessary to build the current module's
packages and dependencies, and it removes unused modules that
don't provide any relevant packages. It also adds any missing entries
to go.sum and removes any unnecessary ones.

[...]
```

Entonces, ya sabemos que `go mod tidy` se va a ocupar de agregar o corregir cualquier tipo de dependencia perdida en nuestro código.

console

```
go mod tidy
```

output

```
go: finding module for package github.com/fatih/color
go: downloading github.com/fatih/color v1.18.0
go: found github.com/fatih/color in github.com/fatih/color v1.18.0
go: downloading golang.org/x/sys v0.25.0
go: downloading github.com/mattn/go-colorable v0.1.13
```

Si ahora vemos cómo luce el archivo `go.mod` podemos comprobar que se agregó la directiva `require` donde se declaran las dependencias faltantes (pero adicionalmente `go mod tidy` también descargó esas dependencias en nuestro proyecto).

console

```
cat go.mod
```

output

```
module github.com/untref-ayp2/mimodulo

go 1.24.1

require github.com/fatih/color v1.18.0

require (
    github.com/mattn/go-colorable v0.1.13 // indirect
    github.com/mattn/go-isatty v0.0.20 // indirect
    golang.org/x/sys v0.25.0 // indirect
)
```

Si ahora ejecutamos nuevamente nuestro programa, todo debería funcionar como se espera.

console

```
go run main.go
```

output

```
¡Texto en rojo!
```

(el mensaje se imprime en color rojo en la terminal)

¡Para practicar!

Se recomienda recrear el ejemplo previamente presentado y ejecutarlo para corroborar que todo funciona correctamente.

Los ejemplos de este capítulo están en `02-paquetes-y-modulos/ejemplos/` del repositorio taller-go.

Ejercicios

Los ejercicios de este capítulo están en `02-paquetes-y-modulos/ejercicios/` del repositorio taller-go.

2.3 Elementos básicos

Tipos de datos básicos

Go ofrece un conjunto de tipos de datos básicos (o primitivos) que cubren la mayoría de las necesidades.

Categoría	Tipos	Descripción
Booleanos	`bool`	`true` o `false`
Cadenas	`string`	Texto Unicode inmutable
Enteros con signo	`int`, `int8`, `int16`, `int32`, `int64`	Tamaño fijo según el sufijo
Enteros sin signo	`uint`, `uint8`, `uint16`, `uint32`, `uint64`	Solo valores positivos
Punto flotante	`float32`, `float64`	Números decimales
Complejos	`complex64`, `complex128`	Números complejos
Alias	`byte` (= `uint8`), `rune` (= `int32`)	Para trabajar con bytes y caracteres Unicode

Los tipos `int` y `uint` tienen un tamaño que depende de la arquitectura del sistema: 32 bits en sistemas de 32 bits y 64 bits en sistemas de 64 bits.

Go

```go
var activo bool = true
var nombre string = "Algoritmos"
var edad int = 20
var altura float64 = 1.75
var letra byte = 'A'
var simbolo rune = '±'
```

Go es un lenguaje de **tipado estático**, lo que significa que el tipo de cada variable se conoce al momento de compilar y no puede cambiar durante la ejecución.

Go

```go
var x int = 10
x = "hola" // ERROR de compilación
```

Variables

2.3.2.1 Declaración con `var`

La forma más explícita de declarar una variable es con la palabra clave `var`, seguida del nombre y el tipo:

Go

```go
var edad int
var precio float64
var nombre string
var activo bool
```

Si no se asigna un valor inicial, la variable toma el **valor cero** de su tipo:

Tipo	Valor cero
`int`, `float64`, etc.	`0`
`string`	`""` (cadena vacía)
`bool`	`false`

Go

```go
var edad int       // edad == 0
var nombre string  // nombre == ""
var activo bool    // activo == false
```

2.3.2.2 Declaración con inicialización

Se puede declarar y asignar un valor inicial en la misma línea:

Go

```go
var edad int = 25
var nombre string = "Martín"
```

Cuando el tipo se puede inferir del valor, se puede omitir:

Go

```go
var edad = 25       // int
var nombre = "Martín" // string
var precio = 29.99  // float64
```

2.3.2.3 Declaración corta `:=`

Dentro de una función, Go ofrece una sintaxis abreviada con `:=` que declara la variable e infiere el tipo automáticamente:

Go

```
func main() {
    edad := 25
    nombre := "Martín"
    precio := 29.99
}
```

`:=` no se puede usar fuera de funciones (a nivel de paquete no hay una instrucción que ejecutar).

2.3.2.4 Declaración múltiple

Se pueden declarar varias variables en una misma línea:

Go

```
var x, y int = 10, 20
var nombre, apellido string = "Juana", "García"
a, b := 1, "dos"
```

O en un bloque:

Go

```
var (
    nombre   string = "Martín"
    edad     int    = 25
    activo   bool   = true
)
```

2.3.2.5 Asignación vs. declaración

Una vez declarada, una variable se actualiza con `=` (sin `:`):

Go

```
edad := 25
edad = 26 // ok
edad := 27 // ERROR: ya fue declarada
```

Para asignar nuevos valores a múltiples variables se puede usar:

Go

```
x, y := 1, 2
x, y = y, x // intercambia los valores
```

Constantes

Las constantes se declaran con `const` y, a diferencia de las variables, no se pueden modificar después de su declaración.

Go

```
const Pi = 3.1416
const Mensaje string = "Hola"
```

Al igual que con `var`, se pueden agrupar en bloques:

Go

```
const (
    StatusOK   = 200
    Status404  = 404
    Status500  = 500
)
```

Las constantes pueden ser **sin tipo** (sin especificar el tipo explícitamente) o **con tipo**:

Go

```
const n = 500          // sin tipo
const s string = "abc" // con tipo
```

Las constantes sin tipo tienen más flexibilidad: se pueden usar en contextos que esperan tipos compatibles. Por ejemplo, una constante numérica sin tipo se puede usar tanto con `int` como con `float64`:

Go

```
const n = 500
var entero int = n       // ok
var flotante float64 = n // ok
```

Go

```
func main() {
    const nombre = "Go"
    const anio = 2009

    fmt.Println(nombre, anio)

    // anio = 2010 // ERROR: las constantes no se pueden modificar
}
```

Condicionales

2.3.4.1 `if` / `else`

Go utiliza `if` y `else` de forma similar a C o Java, pero sin paréntesis alrededor de la condición:

Go

```
if edad >= 18 {
    fmt.Println("Mayor de edad")
} else {
    fmt.Println("Menor de edad")
}
```

Se pueden encadenar varios `else if`:

Go

```
if nota >= 8 {
    fmt.Println("Promocionado")
} else if nota >= 4 {
    fmt.Println("Regular")
} else {
    fmt.Println("Libre")
}
```

2.3.4.2 `if` con inicialización

Go permite ejecutar una instrucción antes de la condición, separada por punto y coma. La variable declarada en esa inicialización solo existe dentro del bloque del `if`:

Go

```
if err := procesar(); err != nil {
    fmt.Println("Error:", err)
}
```

2.3.4.3 Operadores de comparación y lógicos

Go tiene los operadores habituales: ==, !=, <, >, <=, >=, && (y), || (o), ! (no).

Go

```
if edad >= 18 && tienePermiso {
    fmt.Println("Acceso permitido")
}
```

2.3.4.4 `switch`

El `switch` en Go tiene algunas diferencias importantes con Java:

1. No necesita `break` — cada `case` termina automáticamente al finalizar su bloque
2. Se pueden agrupar varios valores en un mismo `case`
3. Se puede usar sin expresión (como un `if` encadenado)

Go

```go
switch dia := time.Now().Weekday(); dia {
case time.Saturday, time.Sunday:
    fmt.Println("Fin de semana")
default:
    fmt.Println("Día laborable")
}
```

Go

```go
hora := 15
switch {
case hora < 12:
    fmt.Println("Buenos días")
case hora < 19:
    fmt.Println("Buenas tardes")
default:
    fmt.Println("Buenas noches")
}
```

2.3.4.5 `fallthrough`

Si se quiere continuar al siguiente `case` (como en C), se usa la palabra clave `fallthrough`:

Go

```go
switch n := 1; n {
case 1:
    fmt.Println("uno")
    fallthrough
case 2:
    fmt.Println("dos")
}
// Imprime "uno" y luego "dos"
```

Ciclos

Go tiene una sola palabra clave para iterar: `for`. No tiene `while` ni `do-while`, pero el mismo `for` puede usarse de distintas formas para cubrir todos los casos.

2.3.5.1 For clásico

Similar a C o Java: inicialización, condición, incremento.

Go

```go
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

Las variables declaradas en la inicialización (`i := 0`) solo existen dentro del bloque del `for`.

2.3.5.2 For como `while`

Si se omite la inicialización y el incremento, queda solo la condición, comportándose como un `while`:

Go

```go
i := 0
for i < 10 {
    fmt.Println(i)
    i++
}
```

2.3.5.3 For infinito

Si se omite también la condición, el ciclo se ejecuta indefinidamente. Se sale con `break`:

Go

```go
i := 0
for {
    if i >= 10 {
        break
    }
    fmt.Println(i)
    i++
}
```

2.3.5.4 `range`

Go provee la palabra clave `range` para iterar sobre estructuras de datos como arreglos, *slices*, *strings*, mapas o canales. `range` devuelve dos valores: el índice y el elemento.

Go

```go
numeros := []int{10, 20, 30, 40, 50}
for indice, valor := range numeros {
    fmt.Printf("posición %d = %d\n", indice, valor)
}
```

Si solo interesa el valor, se puede ignorar el índice con _:

Go

```go
for _, valor := range numeros {
    fmt.Println(valor)
}
```

También se puede iterar sobre un `string`, en cuyo caso `range` itera por runas (caracteres Unicode), no por bytes:

Go

```go
for posicion, letra := range "Algoritmos" {
    fmt.Printf("%c ", letra)
}
```

2.3.5.5 `break` y `continue`

- `break`: sale del ciclo inmediatamente
- `continue`: salta a la siguiente iteración

Go

```go
for i := 0; i < 10; i++ {
    if i%2 == 0 {
        continue // salta los pares
    }
    if i > 7 {
        break // termina al llegar a 7
    }
    fmt.Println(i)
}
// Imprime: 1 3 5 7
```

Ejercicios

Los ejercicios de este capítulo están en `03-elementos-basicos/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.4 Funciones

A diferencia de Java, donde las funciones siempre pertenecen a una clase, en Go las funciones pueden ser independientes y no es necesario definirlas como métodos estáticos. Las funciones en Go son "ciudadanos de primera clase": una función es un valor que se puede almacenar en una variable, pasar como argumento o devolver como resultado de otra función.

Una función puede tomar cero o más parámetros, y puede devolver cero o más valores.

Parámetros

2.4.1.1 Sintaxis básica

Los parámetros se declaran con nombre y tipo, separados por coma:

Go

```go
func sumar(x int, y int) int {
    return x + y
}
```

En este ejemplo, la función `sumar` toma dos parámetros de tipo `int` y devuelve un `int`. Cabe destacar que, a diferencia de Java, el tipo de retorno se escribe después de los paréntesis.

Si quisiéramos generar algo similar en Java deberíamos declarar una clase con métodos estáticos:

Java

```java
public class Matemática {
    public static int sumar(int a, int b) {
        return a + b;
    }
}
```

2.4.1.2 Parámetros del mismo tipo

Cuando dos o más parámetros consecutivos comparten el mismo tipo, podemos declarar el tipo una sola vez al final:

Go

```go
func sumar(x, y int) int {
    return x + y
}
```

2.4.1.3 Parámetros variádicos

Go permite definir funciones que aceptan una cantidad variable de argumentos con `...` antes del tipo. Estos se conocen como parámetros variádicos y se reciben como un *slice* dentro de la función:

Go

```
func sumar(nums ...int) int {
    total := 0
    for _, n := range nums {
        total += n
    }
    return total
}
```

Go

```
import "fmt"

func main() {
    fmt.Println(sumar(1, 2, 3))
    fmt.Println(sumar(5, 10, 15, 20))
}
```

output

```
6
50
```

Pasaje por valor

En Go los argumentos siempre se pasan por valor. Esto significa que la función recibe una copia del valor original y cualquier modificación dentro de la función no afecta a la variable que se pasó como argumento:

Go

```
func duplicar(x int) {
    x = x * 2
}
```

Go

```
a := 5
duplicar(a)
fmt.Println(a)
```

output

```
5
```

El valor de `a` sigue siendo `5` porque `duplicar` recibió una copia. Esto aplica a todos los tipos: `int`, `float64`, `string`, `struct`, arreglos, etc.

Los *slices* y los *maps* son casos especiales: la estructura que los representa (puntero, longitud, capacidad) se copia, pero tanto el original como la copia comparten el mismo arreglo subyacente. Esto se cubre en detalle en el próximo capítulo.

Valores de retorno

2.4.3.1 Retorno múltiple

Go permite devolver múltiples valores desde una función. Esto se usa frecuentemente para reportar errores junto con el resultado esperado:

Go

```
import "errors"

func divisionSegura(dividendo, divisor float32) (float32, error) {
    if divisor == 0.0 {
        return 0.0, errors.New("división por cero")
    }

    return dividendo / divisor, nil
}
```

Cuando se devuelven múltiples valores, los tipos se encierran entre paréntesis en el orden correspondiente.

2.4.3.2 Valores de retorno nombrados

Go permite asignar nombres a los valores de retorno. Estos nombres actúan como variables declaradas dentro de la función, inicializadas con su valor cero. Al usar `return` sin argumentos se devuelven automáticamente los valores actuales de esas variables:

Go

```
func division(dividendo, divisor float32) (resultado float32, err error) {
    if divisor == 0.0 {
        err = errors.New("división por cero")
```

```
        return
    }

    resultado = dividendo / divisor
    return
}
```

El retorno desnudo (`return` sin valores) es útil cuando los nombres de retorno mejoran la legibilidad, pero su uso excesivo puede tener el efecto contrario.

2.4.3.3 Ignorar valores de retorno

Cuando una función devuelve múltiples valores y no necesitamos alguno de ellos, podemos ignorarlo con el identificador en blanco _:

Go

```
resultado, _ := divisionSegura(10, 2)
```

Esto evita tener que declarar variables que no se van a usar, lo que Go no permite.

Funciones como valores

Al ser "ciudadanos de primera clase", las funciones se pueden asignar a variables:

Go

```
f := sumar
fmt.Println(f(3, 4))
```

output

```
7
```

La variable `f` tiene tipo `func(int, int) int`. También podemos declarar el tipo explícitamente:

Go

```
var operacion func(int, int) int
operacion = sumar
fmt.Println(operacion(10, 5))
```

Esto permite pasar funciones como argumentos a otras funciones, lo que es la base de patrones como *callbacks* y funciones de orden superior.

Funciones anónimas

Go soporta funciones sin nombre que se pueden definir en el lugar donde se necesitan:

Go

```go
func() {
    fmt.Println("función anónima")
}()
```

Los paréntesis al final invocan la función inmediatamente (IIFE). También se pueden asignar a una variable:

Go

```go
saludar := func(nombre string) {
    fmt.Printf("Hola, %s\n", nombre)
}

saludar("Martín")
```

output

```
Hola, Martín
```

Ejercicios

Los ejercicios de este capítulo están en `04-funciones/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.5 Arreglos y Slices

Arreglos

En Go, los arreglos o *arrays* son estructuras de datos que almacenan una cantidad arbitraria de valores **del mismo tipo**. A nivel de memoria, todos sus elementos se encuentran en posiciones contiguas.

El tamaño de un *array* es definido al momento de su creación y determina su "tipo". Es decir, un array de enteros de 7 elementos tiene un tipo diferente a un array de enteros de 3 elementos.

Podemos declarar un array de la siguiente forma:

Go

```
var numeros [7]int
```

Aquí creamos una variable de tipo `[7]int` a la que referenciaremos con el nombre `numeros`.

Para acceder a los elementos de un arreglo o modificarlos, utilizamos su índice. Tal como sucede en la mayoría de los lenguajes de programación, los índices en Go comienzan en `0` y terminan en `largo - 1`.

Go

```
numeros[0] = 42
numeros[3] = 1337

fmt.Println(numeros[0] + numeros[3])
```

output

```
1379
```

En el caso de un *array* de 7 elementos, podremos acceder a los elementos en los índices desde el `0` hasta el `6` inclusive. Intentar acceder a un índice fuera de este rango va a causar un error.

Go

```
numeros[7]
```

output

```
panic: runtime error: index out of range [7] with length 7
```

A diferencia de los *slices* que veremos más adelante, cuando un *array* es pasado como argumento en una función o método, este se pasa por **valor**. Es decir, se crea una copia completa del arreglo, por lo que las modificaciones dentro de la función no afectan al arreglo original (a menos que usemos punteros).

En Go para conocer el largo de un *array* existe la función `len`.

Go

```go
len(numeros)
```

output

```
7
```

Para recorrer un *array* en Go existe la instrucción `range`, que genera un iterador sobre el array devolviendo el índice (`i`) y el valor (`v`), en cada iteración del `for`. Veamos un ejemplo:

Go

```go
nombres := [4]string{"Fabián", "Martín", "Valeria", "Santiago"}

for i, v := range nombres {
    fmt.Println(i, "|", v)
}
```

output

```
0 | Fabián
1 | Martín
2 | Valeria
3 | Santiago
```

Slices

Los *slices* (o tajadas) en castellano representan secuencias de longitud variable cuyos elementos son del mismo tipo. Un tipo de *slice* se escribe como `[]T`, donde los elementos son de tipo `T`; se asemeja a un tipo de *array* sin tamaño.

Los *arreglos* y los *slices* están estrechamente relacionados. Un *slice* es una estructura de datos ligera que da acceso a una subsecuencia (o a todos) los elementos de un arreglo. Ese arreglo se conoce como el **arreglo subyacente** del *slice*.

Un *slice* tiene tres componentes internos:

- **puntero**: apunta al primer elemento del arreglo accesible desde el *slice* (no necesariamente el elemento 0 del arreglo)
- **longitud** (`len`): cantidad de elementos que contiene el *slice*
- **capacidad** (`cap`): cantidad máxima de elementos disponibles desde esa posición hasta el final del arreglo subyacente

La longitud no puede superar la capacidad. Las funciones `len` y `cap` devuelven estos valores para cualquier *slice*.

Podemos imaginar que un *slice* es como una ventana que podemos deslizar sobre un arreglo y nos permite acceder a una parte del mismo.

Go

```go
var s []byte
```

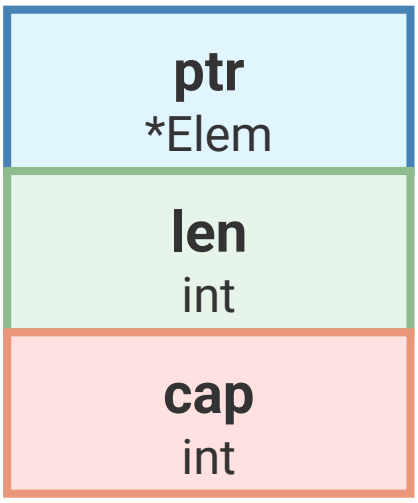


Figura 2.3: Estructura interna de un *slice*.

Go

```go
s = make([]byte, 5, 5)
```

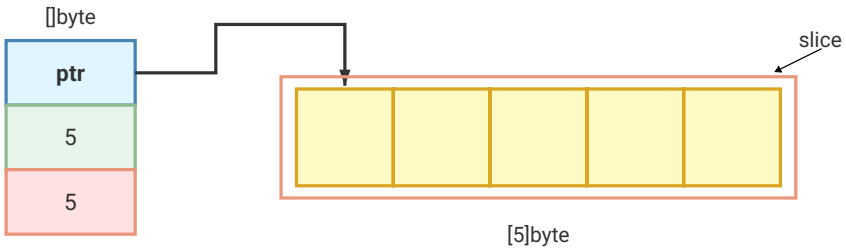


Figura 2.4: *Slice* de longitud 5 y capacidad 5: El arreglo subyacente tiene tamaño 5 y la ventana del slice "ve" todo el arreglo.

A medida que hacemos *slicing* de `s`, observamos los cambios en la estructura de datos del *slice* y su relación con el arreglo subyacente:

Go

```
s = s[2:4]
```

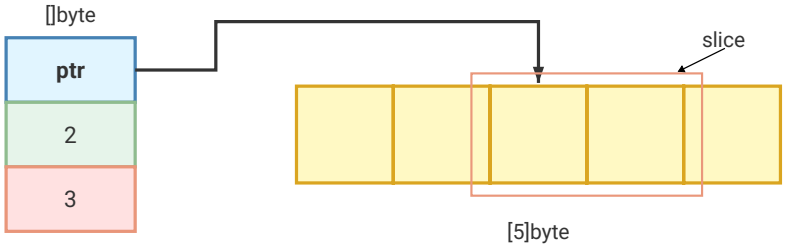


Figura 2.5: *Slice* de longitud 2 y capacidad 3: La ventana del slice ahora ve desde la posición 2 del arreglo subyacente hasta la posición 3 (longitud 2, el último elemento no se incluye). Sin embargo, la capacidad es 3, lo que indica que el slice todavía puede crecer una posición más sobre el mismo arreglo.

El *slicing* no copia los datos del *slice*. En su lugar, crea un nuevo valor de *slice* que apunta a otra porción del arreglo original. Esto hace que las operaciones con *slices* sean tan eficientes como manipular índices de arreglos. Modificar los elementos de un *slice* modifica los elementos del arreglo subyacente:

Go

```
s = s[:cap(s)]
```

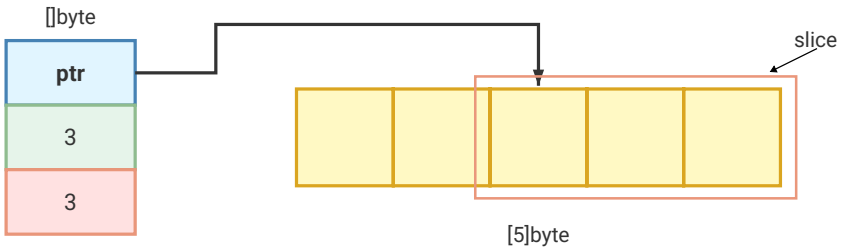


Figura 2.6: *Slice* de longitud 3 y capacidad 3: Ahora la ventana del slice se agrandó y ve desde la posición 2 hasta el final del arreglo subyacente.

Múltiples *slices* pueden compartir el mismo array subyacente y pueden referirse a partes superpuestas de ese array. La siguiente figura muestra un array de cadenas para los meses del año y dos *slices* superpuestos de este. El array se declara como:

Go

```
meses := [12]string{"Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
    "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"}
```

El operador *slice* `s[i:j]` crea un nuevo *slice* con los elementos desde `i` hasta `j-1` del *array* o *slice* `s`, donde $0 \leq i \leq j \leq$ "$cap(s)$" y el resultado tiene `j-i` elementos. Si se omite `i`, se toma 0; si se omite `j`, se toma `len(s)`.

Sobre `meses`, podemos crear slices que referencien subconjuntos. Por ejemplo:

- `meses[0:12]` y `meses[:]` abarcan **todos** los meses
- `meses[1:]` abarca de Febrero a Diciembre

Definamos slices superpuestos para el segundo trimestre y el invierno:

Go

```
t2 := meses[3:6]
invierno := meses[5:8]
```

Go

```
fmt.Println("t2 =", t2, "\ninvierno =", invierno)
```

output

```
t2 = [Abril Mayo Junio]
invierno = [Junio Julio Agosto]
```

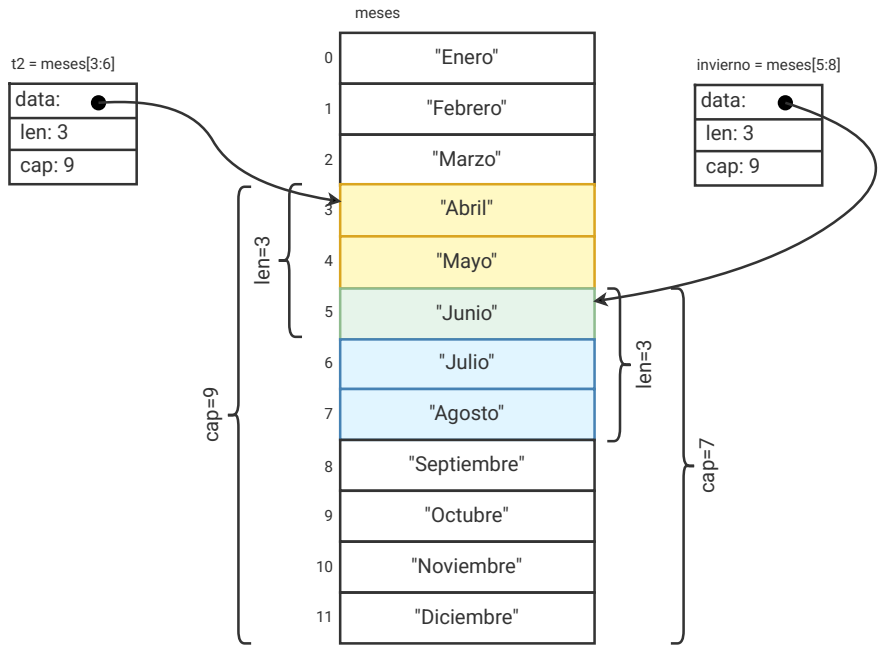


Figura 2.7: Dos *slices* sobre el mismo array de meses.

Hacer *slicing* más allá de `cap(s)` causa un pánico, pero hacer *slicing* más allá de `len(s)` extiende el *slice*, por lo que el resultado puede ser más largo que el original:

Go

```
fmt.Println(invierno[:20])
```

output

```
panic: runtime error: slice bounds out of range [:20] with capacity 7
```

Go

```
inviernoSinFin := invierno[:5]
fmt.Println(inviernoSinFin)
```

output

```
[Junio Julio Agosto Septiembre Octubre]
```

2.5.2.1 Agregando elementos a un *slice*

Para agregar elementos a un *slice* se usa la función `append`. Esta recibe un *slice* y uno o más elementos del mismo tipo, y devuelve un nuevo *slice* con todos los elementos originales más los nuevos.

Si el *slice* resultante entra en la capacidad actual, `append` reutiliza el mismo arreglo subyacente. Si no entra, `append` crea un nuevo arreglo subyacente con capacidad aproximadamente el doble de la original, copia todos los elementos y luego agrega los nuevos.

Go

```
s := []int{1, 2, 3}
fmt.Println(s, "\nlen =", len(s), "\ncap =", cap(s))
```

output

```
[1 2 3]
len = 3
cap = 3
```

Go

```
s = append(s, 4, 5)
fmt.Println(s, "\nlen =", len(s), "\ncap =", cap(s))
```

output

```
[1 2 3 4 5]
len = 5
cap = 6
```

Puede darse el caso en el que, si tenemos dos *slices* sobre un mismo *array* subyacente, al agregar un elemento a uno de los *slices*, el otro también se vea modificado:

Go

```
x := make([]int, 0, 4)
x = append(x, 0, 1, 2)

y := x
y = append(y, 3)

fmt.Println("x =", x, "\ny =", y)
```

output

```
x = [0 1 2]
y = [0 1 2 3]
```

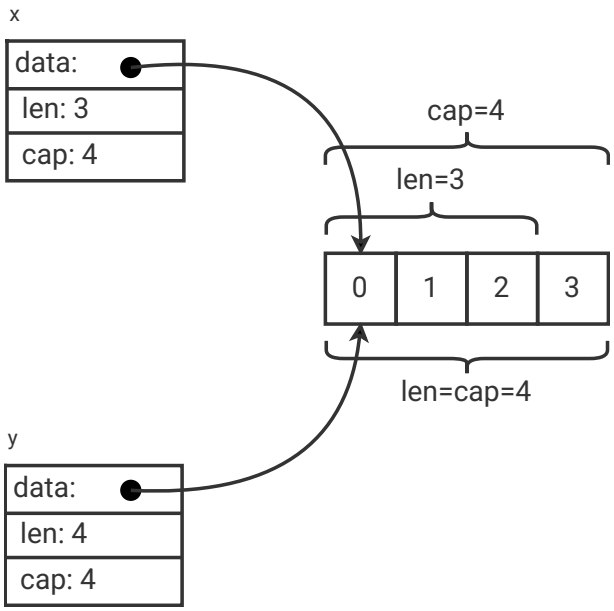


Figura 2.8: Slices x e y comparten el mismo arreglo subyacente.

Go

```
x = append(x, 4)

fmt.Println("x =", x, "\ny =", y)
```

output

```
x = [0 1 2 4]
y = [0 1 2 4]
```

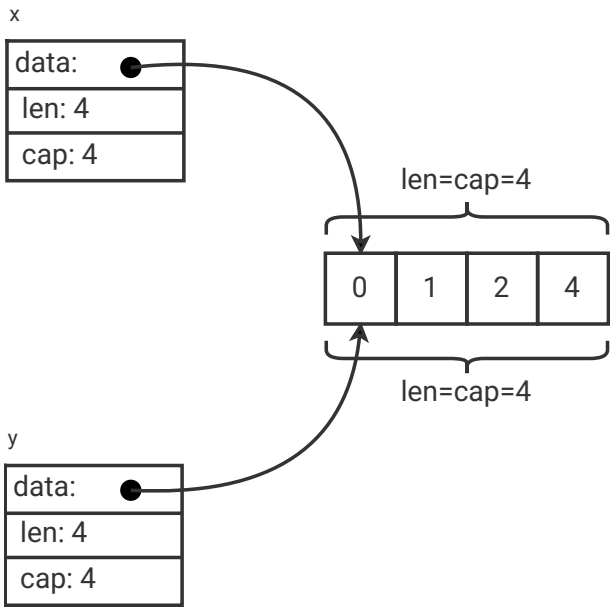


Figura 2.9: Modificar x también afecta a y al compartir el arreglo subyacente.

También si el *slice* sobre el que agregamos el nuevo elemento no tiene más capacidad para agregar elementos, se crea un nuevo *slice* con aproximadamente el doble de capacidad y se copian los elementos del slice original:

Go

```
y = append(y, 4)

fmt.Println("x =", x, "\ny =", y)
```

output

```
x = [0 1 2 4]
y = [0 1 2 4 4]
```

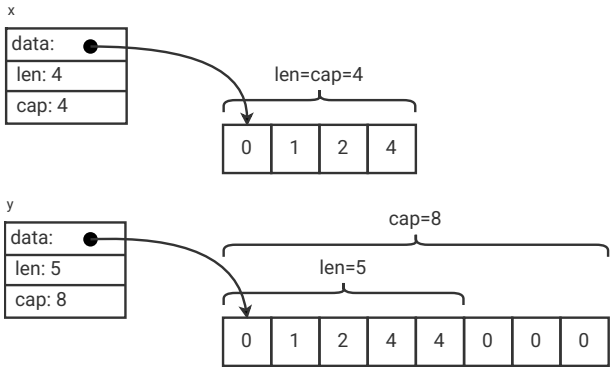


Figura 2.10: Al superar la capacidad, y apunta a un nuevo arreglo subyacente.

Cuando `y = append(y, 4)` superó la capacidad, `append` creó un nuevo arreglo subyacente para `y` y copió allí los elementos. El slice `x`, en cambio, sigue apuntando al arreglo subyacente original.

Por eso, modificar los valores de `y` ya no afecta a `x`:

Go

```
y[3] = 3

fmt.Println("x =", x, "\ny =", y)
```

output

```
x = [0 1 2 4]
y = [0 1 2 3 4]
```

Ejercicios

Los ejercicios de este capítulo están en `05-arreglos-slices/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

Enlaces recomendados

- Go Slices: usage and internals - Un artículo que explica el uso y la estructura interna de los *slices* en Go.

2.6 Mapas

Podemos pensar en los mapas como una generalización de los arreglos, donde en lugar de usar solo enteros como índices, podemos usar otros tipos de datos. Por ejemplo, podemos acceder a un elemento de un mapa usando una cadena como índice `edades["alice"]` en lugar de un entero como `edades[0]`.

Los mapas son una forma de asociar claves a valores, y son útiles para almacenar datos que se pueden identificar mediante una clave. Por ejemplo, en el caso de `edades`, la clave es el nombre de una persona y el valor es su edad.

En Go, un `map` es una referencia a una tabla hash[1], y el tipo de `map` se escribe como `map[K]V`, donde `K` y `V` son los tipos de sus claves y valores. Todas las claves en un mapa dado son del mismo tipo, y todos los valores son del mismo tipo, pero las claves no necesitan ser del mismo tipo que los valores.

> **Importante**
>
> El tipo de clave `K` se debe poder comparar usando `==`, para que el mapa pueda verificar si una clave ya está presente o no.

No hay restricciones sobre el tipo de valor `V`.

Los tipos que se pueden usar como clave son aquellos comparables con `==`: booleanos, números, strings, punteros, canales, y tipos compuestos como *structs* cuyos campos sean todos comparables o *arrays* con elementos comparables. En cambio, los *slices*, mapas y funciones no son comparables y no se pueden usar como clave.

Go

```go
// Válido
m1 := make(map[string]int)
m2 := make(map[int]string)
m3 := make(map[[3]int]string) // array de 3 ints como clave
```

Go

```go
// Inválido: slice como clave (no compila)
// m := make(map[[]string]int)
```

Los mapas son dinámicos, es decir, que pueden crecer o reducir la cantidad de elementos a medida que se agregan o eliminan.

La función *built-in* `make` se puede usar para reservar la memoria que usará un mapa:

[1] Durante el curso veremos qué es y cómo funciona una tabla de *hash*.

Go

```
edades := make(map[string]int)
```

También podemos crear un *mapa literal* para crear un nuevo mapa con algunos pares clave/valor iniciales:

Go

```
edades := map[string]int{
    "alice": 31,
    "charlie": 34,
}
```

Esto es equivalente a

Go

```
edades := make(map[string]int)
edades["alice"] = 31
edades["charlie"] = 34
```

Una expresión alternativa para un nuevo mapa vacío es `map[string]int{}`.

Cuando se conoce la cantidad aproximada de entradas, es más eficiente pre-asignar espacio con un segundo argumento en `make`:

Go

```
edades := make(map[string]int, 100) // capacidad inicial para ~100 entradas
```

Esto evita que el mapa tenga que redimensionar su tabla de *hash* internamente a medida que crece.

Los valores de un mapa también pueden ser otros mapas. Por ejemplo, para asociar nombres de estudiantes con sus notas por materia:

Go

```
notas := map[string]map[string]int{
    "alice": {"matematica": 8, "lengua": 9},
}

fmt.Println(notas["alice"]["matematica"]) // 8
```

Si el mapa interior no está inicializado, asignar un valor produce un error. Por eso es común inicializarlo antes de usarlo:

Go

```
if _, ok := notas["bob"]; !ok {
    notas["bob"] = make(map[string]int)
}
notas["bob"]["matematica"] = 7
```

Los elementos de un mapa se acceden mediante la notación habitual de subíndice:

Go

```
edades["alice"] = 32
edad := edades["alice"]
fmt.Println(edad)
```

output

```
32
```

y se pueden eliminar con la función *built-in* `delete`:

Go

```
delete(edades, "alice")
```

Todas estas operaciones son seguras incluso si el elemento no está en el mapa; una búsqueda en un mapa utilizando una clave que no está presente devuelve el *valor cero* para su tipo. Por ejemplo, lo siguiente funciona incluso cuando `"bob"` aún no es una clave en el mapa porque el valor de `edades["bob"]` será `0`.

Go

```
edades["bob"] = edades["bob"] + 1
```

Las formas abreviadas de asignación `x += y` y `x++` también funcionan para los elementos de un mapa, por lo que podemos reescribir la declaración anterior como

Go

```
edades["bob"] += 1
```

o incluso de forma más concisa como

Go

```
edades["bob"]++
```

Para enumerar todos los pares clave/valor en el mapa, usamos un bucle `for` basado en `range`, similar a los que vimos para *slices*. Las iteraciones sucesivas del bucle hacen que las variables `name` y `age` se configuren con el siguiente par clave/valor:

Go

```
for name, age := range edades {
    fmt.Printf("%s\t%d\n", name, age)
}
```

output

```
charlie  34
bob      3
```

Los mapas en Go no están ordenados y si mostramos todos los pares clave/valor almacenados es posible que el orden se modifique de una ejecución a la siguiente. Esto es intencional; hacer que la secuencia varíe ayuda a forzar que los programas sean robustos entre implementaciones.

Para enumerar los pares clave/valor en orden, debemos ordenar las claves explícitamente, por ejemplo, usando la función `Strings` del paquete `sort`:

Go

```
import "sort"

var names []string

for name := range edades {
    names = append(names, name)
}

sort.Strings(names)

for _, name := range names {
    fmt.Printf("%s\t%d\n", name, edades[name])
}
```

output

```
bob      3
charlie  34
```

Dado que conocemos el tamaño final de `names` desde el principio, es más eficiente asignar un array con el tamaño requerido de antemano. La siguiente declaración crea un *slice* que inicialmente está vacío pero tiene la capacidad suficiente para contener todas las claves del mapa `edades`:

Go

```go
names := make([]string, 0, len(edades))
```

En el primer bucle `range` mencionado anteriormente, solo necesitamos las claves del mapa `edades`, por lo que omitimos la segunda variable del bucle. En el segundo bucle, solo necesitamos los elementos del *slice* `names`, por lo que usamos el identificador en blanco _ para ignorar la primera variable, el índice.

El *valor cero* para un tipo mapa es `nil`, es decir, nulo. En otras palabras el mapa no tiene memoria asignada y no se puede usar. Un mapa `nil` es diferente de un mapa vacío, que es un mapa que tiene memoria asignada pero no tiene claves.

Go

```go
var edades map[string]int
fmt.Println(edades == nil)
fmt.Println(len(edades) == 0)
```

output

```
true
true
```

La mayoría de las operaciones sobre mapas, incluyendo la recuperación, `delete`, `len` y los bucles `range`, son seguras de realizar en un mapa `nil`, ya que se comporta como un mapa vacío. Sin embargo, almacenar en un mapa `nil` provoca un error:

Go

```go
edades["carol"] = 21
```

output

```
panic: assignment to entry in nil map
```

Antes de poder almacenar valores, se debe asignar memoria al mapa.

Acceder a un elemento de un mapa mediante subíndices siempre devuelve un valor. Si la clave está presente en el mapa, obtenemos el valor correspondiente; si no, obtenemos el *valor cero* para el tipo del elemento, como vimos con `edades["bob"]`.

Para muchos propósitos, eso está bien, pero a veces necesitamos saber si el elemento realmente estaba allí o no. Por ejemplo, si el tipo del elemento es numérico, podemos necesitar distinguir entre un elemento inexistente y un elemento que casualmente tiene el *valor cero*, utilizando una prueba como esta:

Go

```
age, ok := edades["bob"]
if !ok { /* "bob" no es una clave en este mapa; age == 0. */ }
```

Es muy común el siguiente patrón que combina las dos sentencias anteriores dentro de la condición del `if`, asignación y comparación en una sola línea:

Go

```
if age, ok := edades["bob"]; !ok { /* ... */ }
```

Utilizar un subíndice en un mapa en este contexto produce dos valores; el segundo es un booleano que indica si el elemento está presente. La variable booleana a menudo se llama `ok`, especialmente si se usa inmediatamente en una condición `if`.

Al igual que con los *slices*, **los mapas no se pueden comparar entre sí**; la única comparación legal es con `nil`. Para verificar si dos mapas contienen las mismas claves y los mismos valores asociados, debemos escribir un bucle.

Go

```
x := map[string]int{"a": 1}
y := map[string]int{"a": 1}
fmt.Println(x == y)
```

output

```
invalid operation: x == y (map can only be compared to nil)
```

Una aplicación común de los mapas es usarlos como un **conjunto** (*set*), es decir, una colección de elementos sin orden ni repetición[2]. Como Go no tiene un tipo `set` nativo, podemos simularlo con un mapa cuyas claves son los elementos del conjunto.

[2]Más adelante estudiaremos los conjuntos como estructura de datos. Por ahora nos alcanza con saber que los conjuntos no pueden tener valores repetidos.

La forma más sencilla es usar `map[Tipo]bool`:

Go

```go
conjunto := make(map[string]bool)
conjunto["manzana"] = true
conjunto["pera"] = true

fmt.Println(conjunto["manzana"]) // true
fmt.Println(conjunto["banana"])  // false (no está presente)
```

Para verificar si un elemento pertenece al conjunto, usamos la sintaxis de dos valores:

Go

```go
if conjunto["manzana"] {
    fmt.Println("manzana está en el conjunto")
}
```

Para eliminar se usa `delete` como con cualquier mapa.

Una variante más eficiente en memoria usa `struct{}` como tipo de valor, ya que `struct{}` ocupa 0 bytes (a diferencia de `bool` que ocupa 1). Esta es la forma preferida en código Go cuando la eficiencia importa:

Go

```go
conjunto := make(map[string]struct{})
conjunto["manzana"] = struct{}{}

if _, ok := conjunto["manzana"]; ok {
    fmt.Println("manzana está en el conjunto")
}
```

Ejercicios

Los ejercicios de este capítulo están en `06-maps/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

Enlaces recomendados

- Effective Go
- Go by Example: Maps
- The Go Programming Language Specification: Map types

2.7 Punteros

¿Qué es un puntero?

Un puntero es una variable que almacena una **dirección de memoria** de otra variable. En vez de contener un valor directamente, contiene la ubicación donde ese valor está guardado.

En Go, cada tipo `T` tiene un tipo puntero asociado `*T`, que es un puntero a un valor de tipo `T`. Por ejemplo, `*int` es un puntero a un entero, `*float64` es un puntero a un número de punto flotante, y `*Persona` es un puntero a una variable de tipo `Persona`.

El tipo del puntero no es solo una formalidad: el compilador necesita saber a qué tipo apunta para poder leer e interpretar correctamente los bytes en esa dirección. Un `*int` sabe que debe leer 8 bytes (en una arquitectura de 64 bits) e interpretarlos como un entero con signo, mientras que un `*float64` también lee 8 bytes pero los interpreta como un número de punto flotante. Un `*byte` lee un solo byte. Si el puntero no tuviera esta información, el compilador no podría generar el código correcto para acceder al valor.

Cuando asignamos una variable a otra en Go, se copia el contenido. Esto vale para `int`, `float64`, `string`, `struct`, arreglos, etc. Para lograr el mismo efecto que las referencias en Java (compartir el acceso a un mismo valor sin copiarlo), Go usa punteros de forma explícita.

Operadores `&` y `*`

Trabajar con punteros implica dos operadores fundamentales:

`&` (dirección de) Devuelve la dirección de memoria de una variable.
`*` (desreferencia) Accede al valor almacenado en la dirección que contiene el puntero. Si se usa en una declaración de tipo, indica que es un tipo puntero (ej: `*int`).

Veamos un ejemplo concreto:

Go

```go
import "fmt"

var a int = 7          // a es una variable int con valor 7
var pa *int = &a       // pa es un puntero a int que guarda la dirección de a

fmt.Printf("La variable a se encuentra en la dirección %p y su valor es %d\n", &a,
a)
fmt.Printf("pa contiene la dirección %p\n", pa)
fmt.Printf("Si desreferenciamos pa con *pa obtenemos %d\n", *pa)
```

output

```
La variable a se encuentra en la dirección 0xc00052a000 y su valor es 7
pa contiene la dirección 0xc00052a000
Si desreferenciamos pa con *pa obtenemos 7
```

Acá ocurre lo siguiente:

1. Declaramos una variable `a` de tipo `int` con valor `7`.
2. Declaramos `pa` de tipo `*int` (puntero a `int`) y le asignamos `&a`, es decir, la dirección de memoria donde vive `a`.
3. Al imprimir `pa` vemos la dirección hexadecimal `0xc00052a000`.
4. Al imprimir `*pa` **desreferenciamos** el puntero: seguimos la flecha hasta la dirección y obtenemos el valor que hay allí, que es `7`.

El operador `*` también permite **modificar** el valor apuntado:

Go

```go
*pa = 99                        // Cambiamos el valor en la dirección apuntada
fmt.Printf("a ahora vale %d (se modificó a través de *pa)\n", a)
```

output

```
a ahora vale 99 (se modificó a través de *pa)
```

Al hacer `*pa = 99` estamos yendo a la dirección que contiene `pa` y escribiendo `99` ahí. Como esa dirección es la de `a`, la variable `a` también cambió.

No confundir el operador * con el tipo *T

En la declaración `var pa *int`, el asterisco forma parte del tipo (`*int` significa "puntero a int"). En cambio, en `*pa` el asterisco es el operador de desreferencia que accede al valor apuntado. El contexto determina el significado.

Diagrama en memoria

La siguiente figura muestra cómo se relacionan las variables `pa` y `a` en memoria:

Figura 2.11: Variables `pa` y `a` en memoria.

`pa` contiene la dirección `0xc00052a000`. Al desreferenciar con `*pa` obtenemos el valor `7`. Toda modificación a través de `*pa` impacta directamente en `a`.

Pasar parámetros por referencia

En el capítulo de funciones vimos que los argumentos en Go siempre se pasan por valor: la función recibe una copia. Esto hace que el siguiente código no funcione como cabría esperar:

Go

```go
func duplicar(x int) {
    x = x * 2
}

a := 5
duplicar(a)
fmt.Println(a)
```

output

```
5
```

`duplicar` recibe una **copia** de `a`, por más que la variable se llame igual. Modificar `x` dentro de la función no afecta a la variable original.

Para modificar el original, necesitamos un puntero:

Go

```go
func duplicar(x *int) {
    *x = *x * 2
}

a := 5
duplicar(&a)
fmt.Println(a)
```

output

```
10
```

Ahora:

1. `duplicar` recibe `&a`, la dirección de `a`.
2. Dentro de la función, `x` es un `*int` que apunta a `a`.
3. `*x = *x * 2` desreferencia `x`, multiplica el valor por 2 y lo guarda en la misma dirección.
4. Al volver, `a` vale `10`.

¿Por qué no pasar siempre punteros?

Los punteros son útiles cuando necesitamos cambiar un valor desde otra función. Sin embargo, si solo necesitamos leer el valor, pasar una copia suele ser más seguro (evita efectos secundarios) y no necesariamente más lento. Go puede optimizar ciertos casos.

Puntero `nil`

El valor cero de un puntero es `nil`. Un puntero `nil` es aquel que no tiene una dirección de memoria válida a la cual apuntar; no está inicializado o se le asignó `nil` explícitamente. Si intentamos desreferenciar un puntero `nil` con `*p`, el programa entra en pánico porque no hay una dirección de memoria válida de la cual leer:

Go

```go
var p *int
fmt.Println(p)
```

output

```
<nil>
```

Go

```
p = nil
fmt.Println(*p) // panic!
```

output

```
panic: runtime error: invalid memory address or nil pointer dereference
```

Siempre debemos verificar que un puntero no sea `nil` antes de desreferenciarlo:

Go

```
if p != nil {
    fmt.Println(*p)
} else {
    fmt.Println("puntero nulo, no se puede desreferenciar")
}
```

output

```
puntero nulo, no se puede desreferenciar
```

Manipular arreglos con punteros

Un arreglo en Go se pasa por valor: pasarlo a una función copia todos sus elementos. Para evitarlo, podemos pasar un puntero al arreglo:

Go

```
func duplicarElementos(arr *[4]int) {
    for i := range arr {
        arr[i] *= 2
    }
}

nums := [4]int{1, 2, 3, 4}
duplicarElementos(&nums)
fmt.Println(nums)
```

output

```
[2 4 6 8]
```

Dentro del cuerpo de la función, Go nos permite escribir `arr[i]` directamente sobre un puntero a arreglo, sin necesidad de `(*arr)[i]`. Se trata de **azúcar sintáctico**[3] (o *syntactic sugar* en inglés): una forma más cómoda de escribir lo mismo.

Relación con slices

En la práctica, en lugar de pasar `*[N]T` se suele usar `[]T` (un slice). El slice ya contiene internamente un puntero al arreglo subyacente, como vimos en el capítulo de arreglos y slices. De hecho, lo siguiente es equivalente al ejemplo anterior pero usando slices:

Go

```
linenos: true
func duplicarElementos(arr []int) {
    for i := range arr {
        arr[i] *= 2
    }
}

nums := []int{1, 2, 3, 4}
duplicarElementos(nums)
fmt.Println(nums)
```

output

```
[2 4 6 8]
```

La diferencia es que cuando usamos `[]int`, el slice ya maneja internamente la contigüidad en memoria del arreglo subyacente, la longitud y la capacidad. Al pasar un slice no necesitamos usar `&` para obtener la dirección; el puntero al arreglo subyacente viaja dentro del slice.

Funciones que devuelven punteros

En Go es perfectamente válido que una función devuelva la dirección de una variable local:

Go

```
func nuevoContador(valorInicial int) *int {
    c := valorInicial
```

[3] **Azúcar sintáctico** es una construcción de un lenguaje de programación que no agrega poder expresivo (es decir, se puede expresar de otra forma), pero hace que el código sea más legible y fácil de escribir. En este caso, `arr[i]` sobre un puntero es azúcar sintáctico porque internamente el compilador lo traduce a `(*arr)[i]`.

```
    return &c
}

contador := nuevoContador(42)
fmt.Println(*contador)

*contador++
fmt.Println(*contador)
```

output

```
42
43
```

A simple vista parece peligroso: `c` es una variable local de `nuevoContador`, ¿no se destruye al salir de la función? Go resuelve esto mediante el *escape analysis* (análisis de escape). El compilador detecta que la dirección de `c` escapa del ámbito de la función, y automáticamente asigna `c` en el *heap* en vez del *stack*. Esto se cubrió en el capítulo de gestión de memoria.

Ejercicios

Los ejercicios de este capítulo están en `07-punteros/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.8 Estructuras e interfaces

Structs

En Go las *structs* son colecciones de campos. A diferencia de las clases en Java, una `struct` no soporta herencia, no tiene constructores explícitos ni métodos *static*. Sin embargo, puede tener métodos asociados (como veremos en breve) y puede incrustar otras *structs* como forma de composición. Podríamos pensar en una `struct` como una clase liviana que agrupa datos, sin el peso de un sistema de herencia ni la maquinaria de POO tradicional.

Go

```
type Direccion struct {
    calle, ciudad, provincia string
    numero                   uint
}

type Persona struct {
    nombre, apellido string
    edad             uint
    direccion        Direccion
}
```

Para acceder a un campo de una estructura se usa la notación de punto, como lo hacemos para acceder a un atributo en Java. También podemos declarar una variable de tipo `struct` de forma literal.

Go

```
func main() {
    var p1 Persona
    p1.nombre = "Marcelo"
    p1.edad = 27

    p2 := Persona{nombre: "Laura", apellido: "Medina", edad: 25}

    fmt.Println(p1.nombre, p1.edad)
    fmt.Println(p2.nombre, p2.apellido)
}
```

output

```
Marcelo 27
Laura Medina
```

2.8.1.1 Métodos

Go no tiene clases como Java; sin embargo, permite definir métodos sobre ciertos tipos.

Un método es una función con un argumento especial **receptor**. El **receptor** aparece en su propia lista de argumentos entre la palabra clave `func` y el nombre del método.

Go

```go
func (p Persona) NombreCompleto() string {
    return p.nombre + " " + p.apellido
}

func (p *Persona) CumplirAnios() {
    p.edad++
}
```

En este ejemplo, `NombreCompleto` tiene un receptor de tipo `Persona` llamado `p`. Al ser un receptor por valor, no modifica el original, solo accede a los campos. `CumplirAnios` tiene un receptor de tipo `*Persona` (puntero a `Persona`), necesario porque modifica el campo `edad`.

Go

```go
func main() {
    p := Persona{nombre: "Ana", apellido: "López", edad: 30}
    fmt.Println(p.NombreCompleto())
    p.CumplirAnios()
    fmt.Println(p.edad)
}
```

output

```
Ana López
31
```

2.8.1.2 Punteros a *struct*

Podemos declarar una variable que sea un puntero a una estructura:

Go

```go
func main() {
    p := &Persona{nombre: "Laura", apellido: "Medina", edad: 25}
    fmt.Println(p.nombre)  // equivalente a (*p).nombre
    p.nombre = "María"
```

```
    fmt.Println(p.nombre)
}
```

output

```
Laura
María
```

Go permite acceder a los campos sin dereferenciar explícitamente el puntero. `p.nombre` equivale a `(*p).nombre`. Es azúcar sintáctico.

Los punteros a struct son útiles para:

- Evitar copiar la estructura al pasarla como parámetro
- Poder modificar los campos desde otra función o método

2.8.1.3 Receptores: puntero vs. valor

Receptor valor	Receptor puntero
No modifica el original	Modifica el original
Copia la *struct*	No copia (más eficiente)
Útil para consultas	Útil para mutaciones

Go aplica azúcar sintáctico: si definimos un método con receptor puntero y lo llamamos sobre una variable no puntero, Go lo convierte automáticamente.

Go

```go
func main() {
    p := Persona{nombre: "Ana", edad: 30}
    p.CumplirAnios()       // Go lo trata como (&p).CumplirAnios()
    fmt.Println(p.edad)    // 31
}
```

output

```
31
```

Regla práctica: usar receptor puntero cuando el método modifica el receptor o cuando la *struct* es grande; usar receptor valor para consultas que no modifican estado.

2.8.1.4 ¿Métodos de clase?

Go no tiene un equivalente directo a *static* de Java ni métodos de clase. Para simular este comportamiento se utilizan **funciones del paquete**. Por convención, suelen llamarse `New<Tipo>`:

Go

```go
func NewPersona(nombre, apellido string, edad uint) Persona {
    return Persona{
        nombre:   nombre,
        apellido: apellido,
        edad:     edad,
    }
}

func main() {
    p := NewPersona("Laura", "Medina", 25)
    fmt.Println(p.nombre, p.apellido, p.edad)
}
```

output

```
Laura Medina 25
```

Estas funciones reemplazan el rol de los constructores o fábricas estáticas.

Interfaces

Como mencionamos anteriormente, en Go existe el concepto de interfaces, pero estas funcionan de forma algo diferente a como lo hacen en Java.

Un tipo `interface` se define como un conjunto de firmas de método[4]. Un valor de ese tipo de interfaz puede contener cualquier valor que implemente todos esos métodos.

2.8.2.1 Implementación de interfaces

En Go la implementación es **implícita** (o estructural): no hace falta declarar `implements`. Un tipo implementa una interfaz simplemente teniendo los métodos que la interfaz exige.

Go

```go
type Caminante interface {
    Avanzar(pasos int)
    Girar(grados float32)
}

func (p *Persona) Avanzar(pasos int) {
    fmt.Printf("%s avanzó %d pasos\n", p.nombre, pasos)
}

func (p *Persona) Girar(grados float32) {
    fmt.Printf("%s giró %.1f grados\n", p.nombre, grados)
}
```

[4]Una **firma de método** (o firma de función) es la declaración de su nombre, parámetros y tipo de retorno, sin incluir el cuerpo. Por ejemplo, en `func (p *Persona) Avanzar(pasos int)`, la firma es `Avanzar(pasos int)`. La implementación (el cuerpo) no forma parte de la firma.

Como `*Persona` implementa ambos métodos, automáticamente satisface la interfaz `Caminante`. Esto significa que una variable de tipo `*Persona` puede usarse donde se espere un `Caminante`.

Go

```go
func RealizarRecorrido(caminante Caminante) {
    caminante.Avanzar(5)
    caminante.Girar(180)
}

func main() {
    p := Persona{nombre: "Marcelo"}
    RealizarRecorrido(&p)
}
```

output

```
Marcelo avanzó 5 pasos
Marcelo giró 180.0 grados
```

Notar que pasamos `&p` porque los métodos están definidos sobre `*Persona`, no sobre `Persona`.

Veamos otro tipo que también implemente `Caminante`:

Go

```go
type Perro struct {
    nombre string
}

func (perro *Perro) Avanzar(pasos int) {
    fmt.Printf("%s avanzó %d pasos\n", perro.nombre, pasos)
}

func (perro *Perro) Girar(grados float32) {
    fmt.Printf("%s giró %.1f grados\n", perro.nombre, grados)
}
```

Tanto `*Persona` como `*Perro` implementan `Caminante`. Podemos escribir una misma función que trabaje con cualquiera de los dos:

Go

```go
func HacerCaminar(c Caminante) {
    c.Avanzar(10)
    c.Girar(90)
}

func main() {
```

```
    HacerCaminar(&Persona{nombre: "Ana"})
    HacerCaminar(&Perro{nombre: "Rex"})
}
```

output

```
Ana avanzó 10 pasos
Ana giró 90.0 grados
Rex avanzó 10 pasos
Rex giró 90.0 grados
```

La función `HacerCaminar` acepta cualquier tipo que implemente `Caminante`, sin importar su tipo concreto. Esto es **polimorfismo**: un mismo contrato (`Caminante`) es satisfecho por distintos tipos, y el código que usa la interfaz funciona igual para todos.

2.8.2.2 Implementación de varias interfaces

Un mismo tipo puede implementar varias interfaces a la vez.

Go

```go
type Trabajador interface {
    Trabajar(horas int) string
}

func (p *Persona) Trabajar(horas int) string {
    return fmt.Sprintf("%s trabajó %d horas", p.nombre, horas)
}

func main() {
    p := Persona{nombre: "Laura"}
    fmt.Println(p.Trabajar(8))
}
```

output

```
Laura trabajó 8 horas
```

Ahora `*Persona` implementa tanto `Caminante` como `Trabajador`.

2.8.2.3 Variables de tipo interfaz

Una variable declarada con un tipo interfaz puede almacenar cualquier valor que la implemente.

Go

```go
func main() {
    ana := Persona{nombre: "Ana", apellido: "López", edad: 30}

    var c Caminante = &ana
    c.Avanzar(5)

    var t Trabajador = &ana
    fmt.Println(t.Trabajar(8))
}
```

output

```
Ana avanzó 5 pasos
Ana López trabajó 8 horas
```

Una misma variable concreta (`ana`) puede verse a través de distintas interfaces: como `Caminante` para moverse o como `Trabajador` para trabajar. Esto es polimorfismo: el código que usa la interfaz queda desacoplado del tipo concreto.

Ejercicios

Los ejercicios de este capítulo están en `08-structs-interfaces/ejercicios/notificaciones/` del repositorio taller-go. El directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.9 Archivos

Trabajar con archivos es una tarea común en casi cualquier programa. Go provee el paquete `os` para operaciones básicas de entrada/salida y `bufio` para lectura y escritura con *buffer*.

Leer un archivo completo

La forma más simple de leer un archivo de texto es con `os.ReadFile`, que lee todo el contenido de una sola vez y devuelve un `[]byte`:

Go

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    datos, err := os.ReadFile("ejemplo.txt")
    if err != nil {
        fmt.Println("Error al leer el archivo:", err)
        return
    }
    fmt.Println(string(datos))
}
```

`os.ReadFile` se encarga de abrir y cerrar el archivo automáticamente. Devuelve los bytes leídos y un error. Si el archivo no existe, el error será del tipo que indica que el archivo no fue encontrado.

Escribir un archivo completo

Para escribir datos en un archivo, `os.WriteFile` recibe el nombre del archivo, los datos como `[]byte` y los permisos:

Go

```go
package main

import (
    "fmt"
    "os"
```

```
)

func main() {
    contenido := []byte("Hola, archivo!\n")
    err := os.WriteFile("salida.txt", contenido, 0644)
    if err != nil {
        fmt.Println("Error al escribir el archivo:", err)
        return
    }
    fmt.Println("Archivo escrito correctamente")
}
```

El tercer parámetro son los permisos en octal (`0644` significa lectura/escritura para el dueño, solo lectura para el resto). `os.WriteFile` crea el archivo si no existe, o lo trunca si ya existe.

Abrir y cerrar archivos manualmente

Para tener más control, podemos abrir un archivo con `os.Open` (solo lectura) u `os.Create` (escritura, trunca si existe). Es importante cerrar el archivo con `Close` cuando terminamos, generalmente usando `defer`:

Go

```
package main

import (
    "fmt"
    "os"
)

func main() {
    archivo, err := os.Open("ejemplo.txt")
    if err != nil {
        fmt.Println("Error al abrir el archivo:", err)
        return
    }
    defer archivo.Close()

    // leer del archivo...
    fmt.Println("Archivo abierto correctamente")
}
```

`defer archivo.Close()` garantiza que el archivo se cierre cuando la función termina, incluso si ocurre un error o *panic* después de abrirlo.

Leer línea por línea con `bufio.Scanner`

Cuando el archivo es grande o queremos procesarlo línea por línea, `bufio.Scanner` es la herramienta adecuada:

Go

```go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    archivo, err := os.Open("ejemplo.txt")
    if err != nil {
        fmt.Println("Error al abrir el archivo:", err)
        return
    }
    defer archivo.Close()

    scanner := bufio.NewScanner(archivo)
    linea := 1
    for scanner.Scan() {
        fmt.Printf("%d: %s\n", linea, scanner.Text())
        linea++
    }

    if err := scanner.Err(); err != nil {
        fmt.Println("Error al leer el archivo:", err)
    }
}
```

`scanner.Scan()` avanza a la siguiente línea y devuelve `true` mientras haya datos. `scanner.Text()` devuelve la línea actual (sin el salto de línea). Al finalizar, `scanner.Err()` indica si hubo algún error durante la lectura.

Escribir con formato

Para escribir datos formateados en un archivo, podemos usar `fmt.Fprintf`, que es como `fmt.Printf` pero escribe en un archivo (o cualquier otro `io.Writer`):

Go

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    archivo, err := os.Create("notas.txt")
    if err != nil {
```

```
        fmt.Println("Error al crear el archivo:", err)
        return
    }
    defer archivo.Close()

    fmt.Fprintf(archivo, "Alumno: %s\n", "Martín")
    fmt.Fprintf(archivo, "Nota: %d\n", 9)
    fmt.Fprintf(archivo, "Materia: %s\n", "Algoritmos II")
    fmt.Fprintln(archivo, "Primera línea")
    fmt.Fprintln(archivo, "Segunda línea")
}
```

Ejercicios

Los ejercicios de este capítulo están en `09-archivos/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.10 Errores

Si bien Go no implementa excepciones como una construcción propia del lenguaje, es posible hacer un control de errores por medio del módulo `errors`.

Por eso, en Go es común que los errores sean devueltos "normalmente" desde una función y quien invoca dicha función es responsable de manejar un posible error.

Crear errores simples

Go

```go
package main

import (
    "errors"
    "fmt"
)

func test(input int) error {
    if input < 0 {
        // Creamos un error con un mensaje estático
        return errors.New("el nro. ingresado debe ser mayor o igual a cero")
    }
    return nil
}

func main() {
    err := test(-1)
    if err != nil {
        fmt.Println(err)
    }
}
```

output

```
el nro. ingresado debe ser mayor o igual a cero
```

Crear errores con formateo dinámico

Además, Go incluye en el paquete `fmt` la función `fmt.Errorf`, que permite crear errores con mensajes formateados, similar a `fmt.Sprintf`. Esto es muy útil para **agregar información dinámica** al mensaje de error.

Go

```go
package main

import "fmt"

func test(input int) error {
    if input < 0 {
        // Se agrega información dinámica al mensaje
        return fmt.Errorf("el nro. ingresado debe ser mayor o igual a cero, "+
            "en su lugar se recibió: %d", input)
    }
    return nil
}

func main() {
    err := test(-1)
    if err != nil {
        fmt.Println(err)
    }
}
```

output

```
el nro. ingresado debe ser mayor o igual a cero, en su lugar se recibió: -1
```

En este ejemplo, la función retorna un error más informativo que incluye el valor de entrada que causó el problema. Este tipo de mensajes con contexto es de gran ayuda para depuración y *logging*.

En los dos ejemplos anteriores, podemos observar la forma típica en la que se maneja un error en Go. Van a encontrar `if err != nil` en múltiples lugares de un programa de Go.

La interfaz `error`

Internamente, `error` es una **interfaz** incorporada en el lenguaje:

Go

```go
type error interface {
    Error() string
}
```

Cualquier tipo que implemente `Error() string` cumple automáticamente la interfaz `error` y puede usarse como valor de error. Tanto `errors.New` como `fmt.Errorf` devuelven tipos que implementan esta interfaz, por eso funcionan donde se espera un `error`.

Si definís tu propio tipo con un método `Error() string`, también podés usarlo como error. Esto permite crear errores con más información:

Go

```go
package main

import "fmt"

type ErrEdad struct {
    Valor int
}

func (e *ErrEdad) Error() string {
    return fmt.Sprintf("edad inválida: %d", e.Valor)
}

func validarEdad(edad int) error {
    if edad < 0 || edad > 150 {
        return &ErrEdad{Valor: edad}
    }
    return nil
}

func main() {
    err := validarEdad(-5)
    if err != nil {
        fmt.Println(err)
    }
}
```

output

```
edad inválida: -5
```

Acá `ErrEdad` implementa la interfaz `error` porque tiene `Error() string`. La función `validarEdad` devuelve un `*ErrEdad` (que es un `error`) cuando la edad es inválida.

Errores centinela

Un patrón muy común en Go es definir errores como **variables globales** (llamados errores centinela). Sirven para comparar con == y saber exactamente qué error ocurrió:

Go

```go
package main

import (
    "errors"
    "fmt"
)
```

```go
var ErrNotFound = errors.New("elemento no encontrado")

func buscar(numeros []int, target int) (int, error) {
    for _, n := range numeros {
        if n == target {
            return n, nil
        }
    }
    return 0, ErrNotFound
}

func main() {
    nums := []int{10, 20, 30}
    _, err := buscar(nums, 25)
    if err == ErrNotFound {
        fmt.Println("elemento no encontrado")
    }
}
```

output

```
elemento no encontrado
```

Notar que `ErrNotFound` se declara afuera de cualquier función, como variable del paquete. Por convención, estos errores se nombran con el prefijo `Err` (son exportados) o `err` (si son internos al paquete).

Tipos de error personalizados

Cuando necesitás que un error lleve **más información** que solo un mensaje, podés definir un struct que implemente la interfaz `error`:

Go

```go
package main

import "fmt"

type ValidationError struct {
    Campo string
    Valor int
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("error en %s: %d no es válido",
        e.Campo, e.Valor)
}

func validarEdad(edad int) error {
```

```
    if edad < 0 {
        return &ValidationError{
            Campo: "edad",
            Valor: edad,
        }
    }
    return nil
}

func main() {
    err := validarEdad(-1)
    if err != nil {
        fmt.Println(err)
    }
}
```

output

```
error en edad: -1 no es válido
```

Esto es útil porque quien recibe el error puede **consultar los campos** del `ValidationError` haciendo una conversión de tipo (*type assertion*) para obtener más detalles.

Wrapping de errores con `%w`

Muchas veces una función llama a otra que puede fallar, y queremos **agregar contexto** al error original sin perderlo. Go permite esto con `fmt.Errorf` y el verbo `%w`:

Go

```
package main

import (
    "errors"
    "fmt"
    "os"
)

func cargarUsuario(id int) (string, error) {
    ruta := fmt.Sprintf("usuarios/%d.txt", id)
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return "", fmt.Errorf("cargando usuario %d: %w", id, err)
    }
    return string(datos), nil
}

func main() {
    _, err := cargarUsuario(42)
    if err != nil {
```

```
        if errors.Is(err, os.ErrNotExist) {
            fmt.Println("el archivo del usuario no existe")
        } else {
            fmt.Println("error inesperado:", err)
        }
        return
    }
}
```

output

```
el archivo del usuario no existe
```

Notar que el mensaje del error original (`os.ErrNotExist`) queda preservado dentro del error que creamos con `%w`. Por eso `errors.Is` puede detectarlo aunque esté envuelto en una capa con más contexto. Así podemos reaccionar distinto según el tipo de error que ocurrió, sin importar cuánto contexto se haya agregado en el camino.

Convenciones en el manejo de errores

Para cerrar, algunas convenciones que se ven en todo código Go:

1. **El error es siempre el último valor de retorno.** Si una función devuelve un resultado y puede fallar, la firma es `(resultado, error)`.
2. **Revisar `err != nil` inmediatamente.** Apenas recibís un error, lo *checkeás*. No lo guardás para después.
3. **No ignorar errores.** Si una función devuelve `error` y estás seguro de que no va a fallar, usar `_ = llamada()` como señal explícita de que decidiste ignorarlo.
4. **Preferir `errors.New` y `fmt.Errorf` a `panic`.** Salvo excepciones muy contadas, los errores se manejan con valores, no con `panic`.

Manejo de errores con archivos

Como viste en el capítulo 2-8, las funciones del paquete `os` devuelven un `error` que tenés que revisar. Combinado con lo que vimos antes, podés **agregar contexto** al error con `%w` para saber en qué operación falló:

Go

```
package main

import (
    "fmt"
    "os"
)
```

```go
func leerArchivo(ruta string) (string, error) {
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return "", fmt.Errorf("error al leer %s: %w", ruta, err)
    }
    return string(datos), nil
}

func main() {
    contenido, err := leerArchivo("mensaje.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Print(contenido)
}
```

output

```
error al leer mensaje.txt: open mensaje.txt: no such file or directory
```

Lo mismo al escribir:

Go

```go
package main

import (
    "fmt"
    "os"
)

func escribirArchivo(ruta, texto string) error {
    err := os.WriteFile(ruta, []byte(texto), 0644)
    if err != nil {
        return fmt.Errorf("error al escribir %s: %w", ruta, err)
    }
    return nil
}

func main() {
    err := escribirArchivo("salida.txt", "Hola, archivo!\n")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("archivo escrito correctamente")
}
```

output

```
archivo escrito correctamente
```

Este patrón — función que delega en `os.ReadFile`/`os.WriteFile` y envuelve el error con `fmt.Errorf` + `%w` — es muy común en programas Go que trabajan con archivos.

Ejercicios

Los ejercicios de este capítulo están en `10-errores/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.11 ¿Orientación a objetos?

De FAQ: Is Go an object-oriented language?:

> Sí y no. Aunque Go tiene tipos y métodos y permite un estilo de programación orientado a objetos, no existe una jerarquía de tipos. El concepto de "interfaz" en Go proporciona un enfoque diferente que creemos que es fácil de usar y, en cierto modo, más general. También hay formas de incrustar tipos en otros tipos para proporcionar algo análogo, pero no idéntico, a la creación de subclases. Además, los métodos en Go son más generales que en C++ o Java: se pueden definir para cualquier tipo de datos, incluso tipos integrados, como enteros "sin caja". No están restringidos a estructuras (clases).
>
> Además, la falta de una jerarquía de tipos hace que los "objetos" en Go parezcan mucho más ligeros que en lenguajes como C++ o Java.

En este capítulo vamos a comparar cómo se modelan objetos con estado y comportamiento en Java frente a Go. Para eso vamos a usar un ejemplo concreto: una `Persona` con datos básicos y un `Estudiante` que extiende a `Persona` agregando información propia.

Java: herencia de clases

En Java, la relación entre `Persona` y `Estudiante` se modela con herencia: `Estudiante extends Persona`. Empecemos por `Persona`:

Java

```java
public class Persona {
    private String nombre;
    private String direccion;
    private String dni;

    public Persona(String nombre, String direccion, String dni) {
        this.nombre = nombre;
        this.direccion = direccion;
        this.dni = dni;
    }

    public String saludar() {
        return "¡Hola! Soy " + nombre;
    }

    public String caminar() {
        return nombre + " está caminando";
    }

    public String getNombre() {
```

```
        return nombre;
    }
}
```

Ahora `Estudiante` hereda de `Persona` y agrega su propio atributo y método:

Java

```
public class Estudiante extends Persona {
    private int legajo;

    public Estudiante(String nombre, String direccion, String dni, int legajo) {
        super(nombre, direccion, dni);
        this.legajo = legajo;
    }

    @Override
    public String saludar() {
        return "¡Hola! Soy " + getNombre() + " y soy estudiante (legajo: " + legajo
+ ")";
    }

    public String cursarMateria(String materia) {
        return getNombre() + " se inscribió en " + materia;
    }
}
```

Notar que:

- Usamos `super()` para llamar al constructor de `Persona`.
- Sobrescribimos (`@Override`) el método `saludar()` para personalizarlo.
- `caminar()` no se sobrescribe, así que `Estudiante` usa la versión de `Persona`.
- Como los atributos de `Persona` son `private`, accedemos a `nombre` mediante el getter `getNombre()`.

2.11.1.1 Polimorfismo en Java

Un `main` que aprovecha el polimorfismo:

Java

```
public class Main {
    public static void main(String[] args) {
        Persona[] personas = new Persona[2];
        personas[0] = new Persona("Marcelo", "Av. Siempre Viva 123", "12345678");
        personas[1] = new Estudiante("Laura", "Calle Falsa 456", "87654321",
2025001);

        for (Persona p : personas) {
            System.out.println(p.saludar());
            System.out.println(p.caminar());
```

```
            if (p instanceof Estudiante) {
                System.out.println(((Estudiante) p).cursarMateria("Algoritmos
II"));
            }

            System.out.println();
        }
    }
}
```

output

```
¡Hola! Soy Marcelo
Marcelo está caminando

¡Hola! Soy Laura y soy estudiante (legajo: 2025001)
Laura está caminando
Laura se inscribió en Algoritmos II
```

El array `Persona[]` contiene tanto `Persona` como `Estudiante`. Al iterar, Java resuelve qué versión de `saludar()` ejecutar según el tipo real del objeto (ligadura dinámica). A `cursarMateria()` solo lo podemos llamar si preguntamos con `instanceof` y casteamos.

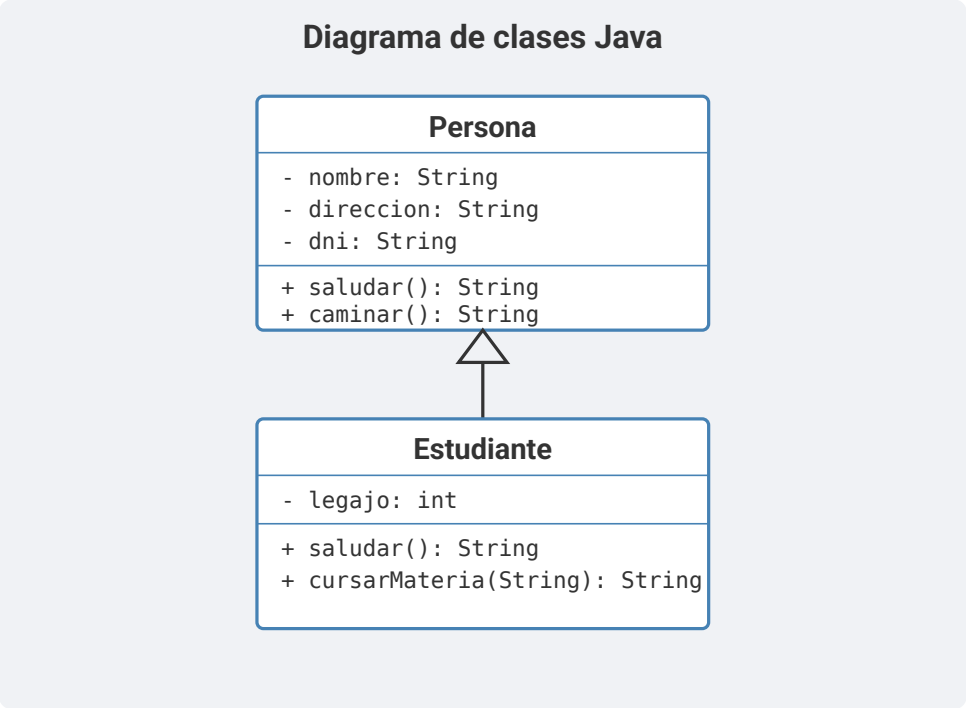


Figura 2.12: Diagrama de clases Java: herencia

Go: composición con *structs*

Go no tiene herencia. En su lugar, usa **composición**: se arma un tipo nuevo incrustando otro tipo existente. Veamos paso a paso cómo modelar el mismo ejemplo.

2.11.2.1 Persona

Primero definimos la estructura y sus métodos:

Go

```go
package persona

import "fmt"

type Persona struct {
  nombre    string
  direccion string
  dni       string
}

func (p Persona) Saludar() string {
  return fmt.Sprintf("¡Hola! Soy %s", p.nombre)
}

func (p Persona) Caminar() string {
  return fmt.Sprintf("%s está caminando", p.nombre)
}
```

Los campos `nombre`, `direccion` y `dni` empiezan con minúscula. En Go esto significa que son **no exportados**: solo se puede acceder a ellos desde el mismo paquete. Los métodos `Saludar` y `Caminar` arrancan con mayúscula, así que son **exportados** (visibles desde afuera del paquete).

2.11.2.2 Constructor con validación

Go no tiene constructores como Java. Por convención, se usa una función `NewPersona` que devuelve el puntero y, si algo sale mal, un error:

Go

```go
func NewPersona(nombre, direccion, dni string) (*Persona, error) {
  if dni == "" {
    return nil, fmt.Errorf("el DNI no puede estar vacío")
  }
  return &Persona{
    nombre:    nombre,
    direccion: direccion,
    dni:       dni,
  }, nil
}
```

Acá aparecen dos conceptos importantes:

1. **Exportado/no exportado**: `NewPersona` arranca con mayúscula, entonces quien importe el paquete `persona` puede llamarlo. Si arrancara con minúscula (`newPersona`), sería interno del paquete.
2. **Error como valor**: en lugar de lanzar una excepción (como en Java), devolvemos un `error`. Quien llama decide qué hacer con él.

2.11.2.3 Estudiante con composición

Para modelar `Estudiante`, en lugar de heredar **componemos** `Estudiante` a partir de `Persona`, la *struct* `Estudiante` contiene una `Persona` como campo embebido (*embedded*) y le agrega sus propios campos y métodos. `Persona` no tiene un nombre de campo definido.

Go

```
type Estudiante struct {
  Persona
  legajo int
}
```

Al escribir `Persona` sin nombre de campo, decimos que `Estudiante` está **compuesto por** una `Persona` a la que se le agregan atributos y comportamiento. Esto significa que:

- Los campos de `Persona` (`nombre`, `direccion`, `dni`) se **promueven** a `Estudiante`: como `Estudiante` contiene internamente una `Persona`, puede acceder a sus campos directamente como `e.nombre`, `e.direccion`, etc.
- Los métodos de `Persona` (`Saludar`, `Caminar`) también se **promueven**: están disponibles en `Estudiante` sin necesidad de redefinirlos, a menos que queramos escribir una versión distinta.
- Como `Persona` y `Estudiante` están en el mismo paquete, los campos no exportados de `Persona` son accesibles desde `Estudiante`.

2.11.2.4 *Override* de métodos y nuevos métodos

Podemos definir un método con el mismo nombre para que haga algo distinto:

Go

```
func (e Estudiante) Saludar() string {
  return fmt.Sprintf("¡Hola! Soy %s y soy estudiante (legajo: %d)", e.nombre,
e.legajo)
}

func (e Estudiante) CursarMateria(materia string) string {
  return fmt.Sprintf("%s se inscribió en %s", e.nombre, materia)
}
```

Notar que:

- Accedemos a `e.nombre` directamente (está promovido desde `Persona`).
- `Caminar()` no se redefine, así que `Estudiante` usa la versión de `Persona`.
- El constructor `NewEstudiante` llama a `NewPersona` internamente y agrega su propia validación.

2.11.2.5 Constructor de Estudiante

Go

```go
func NewEstudiante(nombre, direccion, dni string, legajo int) (*Estudiante, error)
{
  if legajo <= 0 {
    return nil, fmt.Errorf("el legajo debe ser positivo, se recibió: %d", legajo)
  }
  p, err := NewPersona(nombre, direccion, dni)
  if err != nil {
    return nil, err
  }
  return &Estudiante{Persona: *p, legajo: legajo}, nil
}
```

Si el legajo es inválido, retornamos error sin crear el `Estudiante`. Si la validación de `NewPersona` falla (por ejemplo DNI vacío), propagamos ese error.

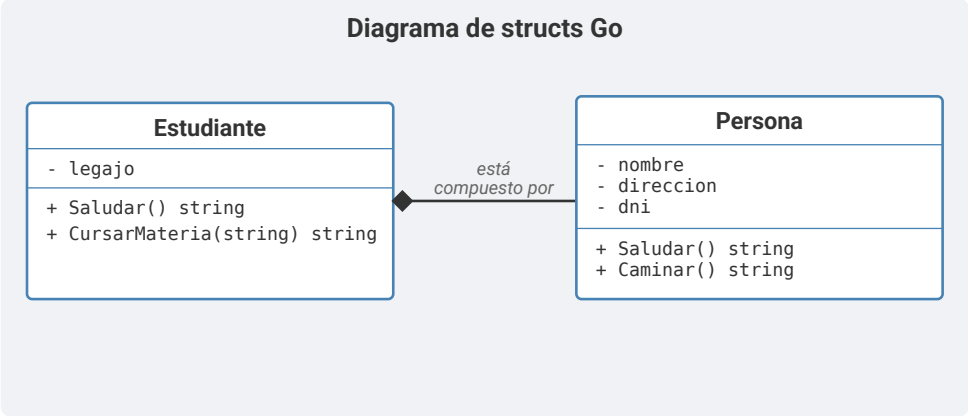


Figura 2.13: Diagrama de *structs* Go: composición

La relación entre `Estudiante` y `Persona` no es herencia sino **composición**: `Estudiante` está compuesto por una `Persona` más el campo `legajo`. El rombo relleno en el diagrama indica esto.

Polimorfismo en Go con interfaces

En Java el polimorfismo viene dado por la herencia: un `Estudiante` es una `Persona`. En Go el polimorfismo se logra con ***interfaces***. Definimos una interfaz que describa el comportamiento común:

Go

```go
type PersonaInterface interface {
  Saludar() string
  Caminar() string
}
```

Tanto `*Persona` como `*Estudiante` implementan esta interfaz automáticamente (porque ambos tienen `Saludar() string` y `Caminar() string`). No hace falta declarar `implements`.

Ahora podemos escribir un `main` que itere sobre un *slice* de `PersonaInterface`:

Go

```
func main() {
  marcelo, err := persona.NewPersona("Marcelo", "Av. Siempre Viva 123", "12345678")
  if err != nil {
    fmt.Println(err)
    return
  }

  laura, err := persona.NewEstudiante("Laura", "Calle Falsa 456", "87654321",
2025001)
  if err != nil {
    fmt.Println(err)
    return
  }

  personas := []persona.PersonaInterface{marcelo, laura}

  for _, p := range personas {
    fmt.Println(p.Saludar())
    fmt.Println(p.Caminar())

    if e, ok := p.(*persona.Estudiante); ok {
      fmt.Println(e.CursarMateria("Algoritmos II"))
    }

    fmt.Println()
  }
}
```

output

```
¡Hola! Soy Marcelo
Marcelo está caminando

¡Hola! Soy Laura y soy estudiante (legajo: 2025001)
Laura está caminando
Laura se inscribió en Algoritmos II
```

Algunas observaciones:

- El comportamiento es idéntico al ejemplo de Java, pero el mecanismo es distinto.
- En Java, `Persona[]` funciona porque `Estudiante` *es una* `Persona` por herencia.
- En Go, `[]PersonaInterface` funciona porque tanto `*Persona` como `*Estudiante` implementan la interfaz.
- Para llamar a `CursarMateria` hacemos una **type assertion** (`p.(*persona.Estudiante)`) y verificamos con `ok`. Es análogo al `instanceof` y casteo de Java. Esta limitación existe porque, al no haber herencia, no podemos tratar a `Estudiante` como una `Persona` en tiempo de compilación. La interfaz permite polimorfismo sobre los métodos comunes, pero los métodos específicos de `Estudiante` requieren preguntar por el tipo concreto.

Una ventaja que Go tiene sobre Java es que un tipo puede implementar **varias *interfaces* a la vez**, lo que permite simular la **herencia múltiple** (un tipo que tenga comportamiento de varias fuentes distintas). En Java las clases solo pueden heredar de una, aunque pueden implementar varias *interfaces*. En Go no hay herencia, pero cualquier *struct* puede implementar cualquier cantidad de *interfaces* con solo definir los métodos correspondientes.

Comparación: herencia vs composición

Concepto	Java	Go
Reutilización	Herencia (`extends`)	Composición (incrustación)
Jerarquía de tipos	Sí, obligatoria	No existe
Polimorfismo	Ligadura dinámica por herencia	Interfaces implícitas
Visibilidad	`private`, `protected`, `public`	Mayúscula = exportado, minúscula = no exportado
Constructores	Con el mismo nombre de la clase	Función `NewTipo()` que devuelve `(*Tipo, error)`
Errores	Excepciones (`try/catch`)	Valores de retorno (`error`)
“this” / “self”	`this` implícito	Receptor explícito (`p Persona`)

Conclusión

Go no es un lenguaje orientado a objetos en el sentido clásico. No tiene clases, ni herencia, ni `@Override`. Pero ofrece las herramientas necesarias para modelar objetos:

- **Structs** para agrupar datos.
- **Métodos con receptores** para asociar comportamiento.
- **Incrustación** para composición y reutilización.
- **Interfaces implícitas** para polimorfismo.

El resultado es un estilo más liviano, donde la jerarquía de tipos no domina el diseño. En vez de preguntar “¿qué clase es esto?”, en Go preguntamos “¿qué comportamientos tiene?”.

Ejercicios

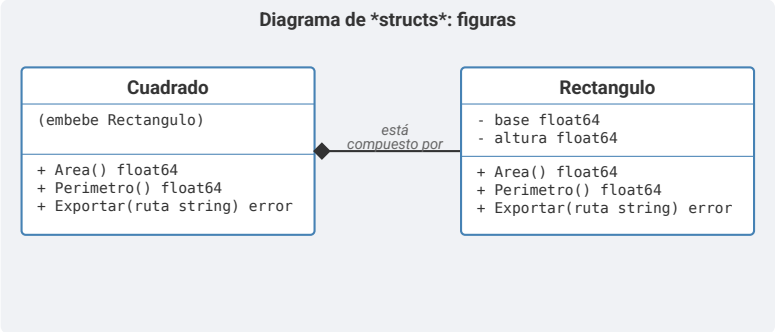


Figura 2.14: Diagrama de *structs* del ejercicio de figuras geométricas.

Los ejercicios de este capítulo están en `11-oop/ejercicios/sistema-figuras/` del repositorio taller-go. El directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

2.12 Tipos parametrizables (genéricos)

En capítulos anteriores trabajamos con funciones y tipos concretos: una función que suma enteros suma solo enteros, una que ordena *strings* ordena solo *strings*. Si queríamos la misma lógica para otro tipo, teníamos que reescribir la función. En este capítulo vamos a ver cómo escribir código que funcione para cualquier tipo usando **tipos parametrizables** (genéricos).

El problema: lógica repetida para cada tipo

Supongamos que queremos una función que busque un elemento en un slice. Para enteros haríamos:

Go

```
func ContieneInt(arr []int, elem int) bool {
    for _, v := range arr {
        if v == elem {
            return true
        }
    }
    return false
}
```

Si después necesitamos lo mismo para *strings*, escribimos otra función casi idéntica:

Go

```
func ContieneString(arr []string, elem string) bool {
    for _, v := range arr {
        if v == elem {
            return true
        }
    }
    return false
}
```

La única diferencia es el tipo de los parámetros. El resto del código es exactamente igual. Esto viola el principio de no repetirse (DRY, por su sigla en inglés *Don't Repeat Yourself*; Wikipedia) y hace que el código sea más difícil de mantener.

Nota

Este problema de código repetido es el mismo que motiva el uso de plantillas (*templates*) en C++ o *generics* en Java.

2.12.1.1 Solución clásica: `interface{}`

Antes de que Go tuviera genéricos, la solución era usar el tipo vacío `interface{}` (que también puede escribirse como `any`). Como `interface{}` no impone ningún método, cualquier valor lo satisface:

Go

```go
func ContieneAny(arr []interface{}, elem interface{}) bool {
    for _, v := range arr {
        if v == elem {
            return true
        }
    }
    return false
}
```

Pero esta solución tiene problemas:

- **Sin verificación en compilación**: si se intentan hacer operaciones como sumar dos valores de tipo `any`, el código compila pero falla en ejecución. El compilador no ayuda a encontrar el error.
- **Código verboso**: para usar el valor hay que preguntar por su tipo concreto con `valor.(Tipo)`, y si el tipo no coincide, el programa explota con un *panic*.
- **Pérdida de información**: adentro del `interface{}` el compilador no sabe si hay un `int`, un `string` o lo que sea.

Go

```go
func main() {
    nums := []interface{}{10, 20, 30}
    fmt.Println(ContieneAny(nums, 20))    // true, funciona bien
    fmt.Println(ContieneAny(nums, "hola")) // false, compila pero no tiene sentido
}
```

output

```
true
false
```

El problema es que el compilador no nos protege: pasar un `string` donde debería ir un `int` no genera ningún error en compilación. Si la función hiciera operaciones aritméticas, el error aparecería recién en ejecución.

Tipos parametrizables (*type parameters*)

Desde Go 1.18, podemos escribir funciones que acepten cualquier tipo usando tipos parametrizables. La sintaxis usa corchetes `[]`:

Go

```
func Contiene[T comparable](arr []T, elem T) bool {
    for _, v := range arr {
        if v == elem {
            return true
        }
    }
    return false
}
```

Analicemos la sintaxis:

- `[T comparable]` declara un tipo parametrizable `T` con la *restricción* (*constraint*) `comparable`.
- `T` se usa como tipo de los parámetros `arr` (slice de `T`) y `elem` (`T`).
- Dentro de la función, los valores son del tipo concreto `T`, no `interface{}`..

La restricción `comparable` indica que `T` debe soportar los operadores `==` y `!=`, que son los que usamos dentro de la función.

Al llamar a la función, Go infiere el tipo automáticamente:

Go

```
func main() {
    numeros := []int{10, 20, 30, 40, 50}
    fmt.Println(Contiene(numeros, 30))   // true
    fmt.Println(Contiene(numeros, 99))   // false

    nombres := []string{"Ana", "Luis", "Pepe"}
    fmt.Println(Contiene(nombres, "Luis")) // true
}
```

output

```
true
false
true
```

La misma función `Contiene` funciona con `int`, `string`, y cualquier otro tipo que sea comparable. Go genera el código necesario para cada tipo concreto en tiempo de compilación, sin penalidad en ejecución.

A diferencia de la solución con `interface{}`, acá el compilador no permite mezclar tipos: si declaramos un slice de enteros, solo podemos buscar enteros dentro de él:

Go

```go
func main() {
    numeros := []int{10, 20, 30}
    fmt.Println(Contiene(numeros, 20))    // bien: int con int

    // Esto NO compila:
    // fmt.Println(Contiene(numeros, "hola")) // error: string no es int
}
```

El error en compilación nos protege de equivocarnos antes de ejecutar el programa.

2.12.2.1 Inferencia de tipos

Al llamar a una función genérica, el compilador deduce los tipos parametrizables a partir de los argumentos:

Go

```go
Contiene(numeros, 30)   // T se infiere como int
Contiene(nombres, "Luis") // T se infiere como string
```

También podemos especificar el tipo explícitamente si hace falta:

Go

```go
Contiene[int](numeros, 30)
```

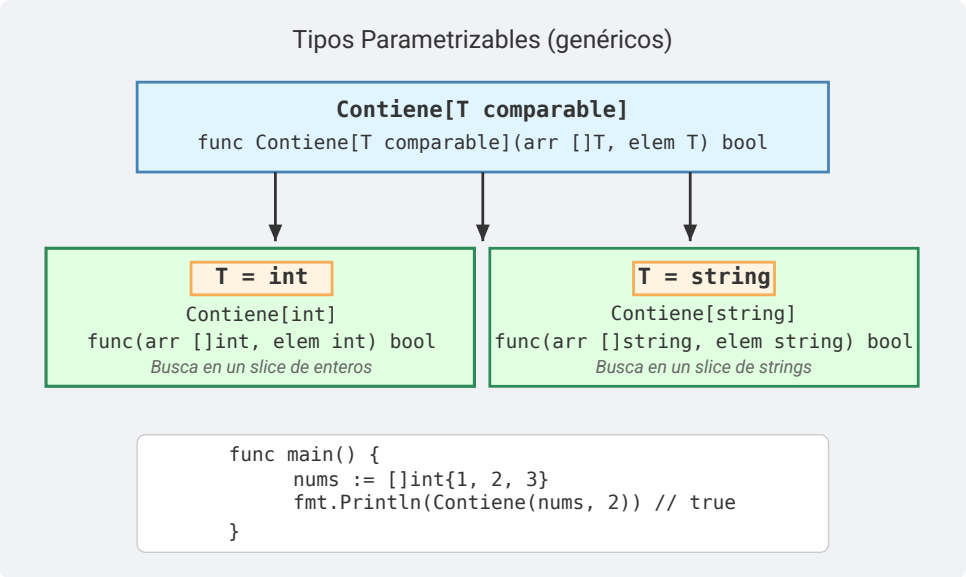


Figura 2.15: Los tipos parametrizables permiten que una misma función trabaje con diferentes tipos concretos.

Restricciones (*constraints*)

Los *constraints* definen qué operaciones puede hacer `T` dentro de la función. Go provee algunos *constraints* predefinidos y permite crear los nuestros.

2.12.3.1 `any`

`any` es un alias para `interface{}`. Acepta cualquier tipo, pero no permite ninguna operación específica sobre los valores (no se puede usar ==, <, +, etc.):

Go

```go
func Imprimir[T any](arr []T) {
    for _, v := range arr {
        fmt.Println(v)
    }
}
```

Esta función solo puede usar operaciones válidas para cualquier tipo: asignar, pasar como parámetro, imprimir con `fmt.Println` (que usa `interface{}` internamente).

2.12.3.2 `comparable`

`comparable` restringe `T` a tipos que soporten == y `!=`. Todos los tipos básicos (enteros, *strings*, booleanos) son comparables, así como punteros, *structs* con campos comparables y arreglos.

Go

```go
func BuscarLineal[T comparable](arr []T, elem T) int {
    for i, v := range arr {
        if v == elem {
            return i
        }
    }
    return -1
}
```

2.12.3.3 Restricciones personalizadas (*custom constraints*)

Para operaciones como <, >, <=, >=, no existe un *constraint* predefinido en la biblioteca estándar (hasta Go 1.21). Podemos definir el nuestro propio:

Go

```go
type Ordenable interface {
    ~int | ~float64 | ~string
}
```

El operador ~ indica que el tipo subyacente debe ser `int`, `float64` o `string`. Esto permite usar tanto los tipos literales (`int`, `string`) como tipos definidos a partir de ellos (por ejemplo, `type Edad int`).

Ahora podemos escribir funciones que usen operadores de comparación:

Go

```go
func Maximo[T Ordenable](arr []T) T {
    max := arr[0]
    for _, v := range arr[1:] {
        if v > max {
            max = v
        }
    }
    return max
}

func main() {
    enteros := []int{10, 5, 8, 3, 12}
    fmt.Println(Maximo(enteros)) // 12

    alturas := []float64{1.75, 1.80, 1.65, 1.90}
    fmt.Println(Maximo(alturas)) // 1.9

    nombres := []string{"Ana", "Luis", "Pepe", "Beatriz"}
    fmt.Println(Maximo(nombres)) // Pepe (orden lexicográfico)
}
```

output

```
12
1.9
Pepe
```

Importante

Los *constraints* con uniones de tipos (`~int | ~float64 | ~string`) solo pueden definirse como interfaces en el contexto de un tipo parametrizable. No pueden usarse como tipos de variables regulares.

Alternativa: función comparadora

En lugar de definir un *constraint* personalizado, podemos pasar una función que compare elementos. Acá el tipo es `[T any]` porque no hacemos operaciones directamente sobre `T`: toda la lógica de comparación está dentro de la función `menor`, que recibe los `T` y decide el orden.

Go

```go
func OrdenarSeleccion[T any](arr []T, menor func(T, T) bool) {
    n := len(arr)
    for i := 0; i < n-1; i++ {
        minIdx := i
        for j := i + 1; j < n; j++ {
```

```
            if menor(arr[j], arr[minIdx]) {
                minIdx = j
            }
        }
        arr[i], arr[minIdx] = arr[minIdx], arr[i]
    }
}

func main() {
    numeros := []int{4, 2, 7, 1, 9}
    OrdenarSeleccion(numeros, func(a, b int) bool { return a < b })
    fmt.Println(numeros) // [1 2 4 7 9]

    nombres := []string{"Pepe", "Ana", "Luis"}
    OrdenarSeleccion(nombres, func(a, b string) bool { return a < b })
    fmt.Println(nombres) // [Ana Luis Pepe]
}
```

output

```
[1 2 4 7 9]
[Ana Luis Pepe]
```

La función `menor` recibe dos elementos `T` y devuelve `true` si el primero debe ir antes que el segundo. Esto nos permite ordenar cualquier tipo, incluso *structs* sin un orden natural:

Go

```
type Persona struct {
    Nombre string
    Edad   int
}

func main() {
    personas := []Persona{
        {"Ana", 30},
        {"Luis", 25},
        {"Pepe", 35},
    }

    OrdenarSeleccion(personas, func(a, b Persona) bool {
        return a.Edad < b.Edad
    })

    fmt.Println(personas) // [{Luis 25} {Ana 30} {Pepe 35}]
}
```

output

```
[{Luis 25} {Ana 30} {Pepe 35}]
```

Tipos genéricos

Los tipos parametrizables también funcionan con *structs*. Esto permite definir contenedores que almacenen valores de cualquier tipo:

Go

```go
type Caja[T any] struct {
    valor T
}

func (c *Caja[T]) Set(valor T) {
    c.valor = valor
}

func (c *Caja[T]) Get() T {
    return c.valor
}

func main() {
    cajaInt := Caja[int]{}
    cajaInt.Set(42)
    fmt.Println(cajaInt.Get()) // 42

    cajaStr := Caja[string]{valor: "hola"}
    fmt.Println(cajaStr.Get()) // hola
}
```

output

```
42
hola
```

Un tipo genérico también puede tener múltiples parámetros:

Go

```go
type Dicc[K comparable, V any] struct {
    datos map[K]V
}

func NuevoDicc[K comparable, V any]() *Dicc[K, V] {
    return &Dicc[K, V]{datos: make(map[K]V)}
}

func (d *Dicc[K, V]) Set(clave K, valor V) {
    d.datos[clave] = valor
}

func (d *Dicc[K, V]) Get(clave K) (V, bool) {
    v, ok := d.datos[clave]
```

```
    return v, ok
}
```

Resumen

Concepto	Sintaxis	Ejemplo
Parámetro de tipo	`[T any]`	`func Imprimir[T any](arr []T)`
Constraint `comparable`	`[T comparable]`	`func Contiene[T comparable](arr []T, elem T) bool`
Constraint personalizado	`type X interface { ~int | ~string }`	`func Maximo[T Ordenable](arr []T) T`
Función comparadora	Parámetro `func(T, T) bool`	`func Ordenar[T any](arr []T, menor func(T, T) bool)`
Struct genérico	`type Nombre[T any] struct { ... }`	`type Caja[T any] struct { valor T }`
Múltiples parámetros	`[K comparable, V any]`	`type Dicc[K comparable, V any] struct { ... }`

Ejercicios

Los ejercicios de este capítulo están en `12-genericos/ejercicios/` del repositorio taller-go. Cada directorio contiene un `README.md` con el enunciado y los esqueletos para resolverlo.

3. Análisis de Algoritmos

Como ya mencionamos antes, podemos entender a un algoritmo como un método para resolver un problema dado en una computadora. Un algoritmo describe los pasos que se deben seguir para alcanzar el resultado buscado.

Para un problema cualquiera, puede existir más de un algoritmo que lo resuelva. Un algoritmo puede consumir más o menos recursos que otro que resuelve el mismo problema, por lo tanto una de las principales variables de análisis generalmente es la **eficiencia** en el uso de los recursos.

Nota

Los recursos que consume un algoritmo son tiempo y espacio (memoria). El mismo problema se puede resolver, en la misma computadora, en minutos o en años, de acuerdo al algoritmo que se implemente.

Otro aspecto a tener en cuenta al analizar algoritmos es la **simplicidad**. Un algoritmo simple es más fácil de entender, adaptar y mantener, por eso a veces prima la simplicidad a la hora de elegir un algoritmo.

La **simplicidad** y la **eficiencia** no son conceptos antagónicos. Es decir, un algoritmo puede ser simple y eficiente a la vez. Estas métricas sirven principalmente para comparar algoritmos entre sí y así poder elegir el más adecuado para una situación concreta.

3.1 Complejidad

Definición

La **complejidad** es una propiedad de un algoritmo que nos indica como escala la cantidad de recursos que se necesitan a medida que aumenta el volumen de los datos.

En esta definición de **complejidad** entra en juego una variable más, la cantidad o el tamaño de los datos, por ejemplo si tenemos que ordenar pocos datos, y rara vez se agregan nuevos datos o se modifican los existentes, quizás prime la simplicidad del algoritmo a la hora de elegir alguno de los algoritmos de ordenamiento, pero si tenemos que ordenar millones de datos, donde además se agregan cientos o miles de datos y se modifican otros muy frecuentemente, es probable que nos interese analizar cuánto tiempo va a demorar y cuánta memoria o espacio

en disco vamos a necesitar y más aún, cuánto más necesitaremos a medida que aumente el volumen de datos, es decir, es probable que prime la **eficiencia** a la hora de elegir.

Informalmente hablando, cuanto mayor sea la **complejidad** de un algoritmo, será menos **eficiente** en el uso de los recursos.

La **complejidad** es independiente del *hardware* o la máquina donde se ejecuta el algoritmo. No debemos confundir **complejidad** con **rendimiento**.

Nota

El **rendimiento** es la capacidad de una computadora para realizar una determinada tarea en un tiempo dado

El **rendimiento** sí depende del *hardware*. Por ejemplo en la medida que aumenta la velocidad del procesador, se necesitará menos tiempo para completar una tarea dada. En cambio la **complejidad** nos indica cuanto más tiempo o memoria va a requerir el algoritmo en función del tamaño de los datos.

El mito de la computadora todopoderosa

Muchas veces se cae en la tentación de pensar que a medida que aumente la capacidad de procesamiento de las computadoras se podrá completar cualquier tarea en un tiempo aceptable. Esta afirmación es una falacia. Existen problemas muy bien conocidos que resultan intratables para las computadoras, es decir, que no importa que tanto aumente la capacidad de procesamiento, un pequeño aumento en el tamaño de los datos implica que el tiempo de ejecución aumente desmesuradamente, hasta el punto de hacer inviable el cálculo.

En esta primera aproximación al estudio del análisis de algoritmos, nos vamos a enfocar en la **complejidad temporal**. Como la complejidad temporal es una métrica independiente del *hardware* donde se ejecuta el programa, normalmente se estima contando la cantidad de operaciones elementales que realiza el algoritmo bajo análisis para completar el cálculo, teniendo en cuenta que cada unidad elemental requiere una cantidad fija de tiempo, que por simplicidad se asume como una unidad de tiempo.

Como la cantidad de operaciones elementales que debe realizar depende de los datos, se toma siempre el peor caso, para poder obtener una cota confiable. Por ejemplo si hay que buscar un elemento en un arreglo desordenado, no queda otra que buscar el elemento en todas las posiciones del arreglo. En el peor caso se deberá recorrer todo el arreglo para encontrar el elemento en la última posición escrutada o poder concluir que el elemento no se encuentra, es probable que si ejecutamos nuestro algoritmo muchas veces, algunas veces lo encuentre antes de recorrer todo el arreglo, pero como buscamos una cota, **se toma siempre el peor caso**.

3.2 Cota Superior Asintótica (O grande)

La cota superior asintótica, O grande o *Big-O* en inglés, es parte de la familia de notaciones asintóticas, también conocidas como notaciones de Bachmann-Landau. En ciencias de la computación, la O grande, permite clasificar a los algoritmos de acuerdo a como aumenta la cantidad de recursos que necesita en la medida que aumenta el tamaño de la entrada, es decir nos permite clasificar algoritmos en el límite, cuando el tamaño de la entrada tiende a infinito. Sea $T(n)$ la función que indica cuánto tiempo va a tardar un algoritmo en función del tamaño de la entrada n, donde n es un número natural, entonces:

Definición

$$T(n) = O(f(n)) \text{ si } \exists\ c,\ n_0 \text{ talque } T(n) \leq c \cdot f(n),\ \forall n > n_0 \tag{3.1}$$

donde c es una constante positiva y n_0 un número natural.

Esto quiere decir que la tasa de crecimiento de $T(n)$ es menor o igual a la tasa de crecimiento de $f(n)$. Es decir la tasa de crecimiento de $T(n)$ está acotada por arriba por $f(n)$, $\forall n > n_0$.

Por ejemplo, si tenemos la función $T(n) = 4n + 1$, se verifica que es de la familia de $O(n)$, es decir la tasa de crecimiento de $T(n)$ está acotada por $f(n) = n$, ya que:

$$\exists\ c = 5,\ n_0 = 1 \tag{3.2}$$

tal que:

$$5n \geq 4n + 1,\ \forall n \geq 1 \tag{3.3}$$

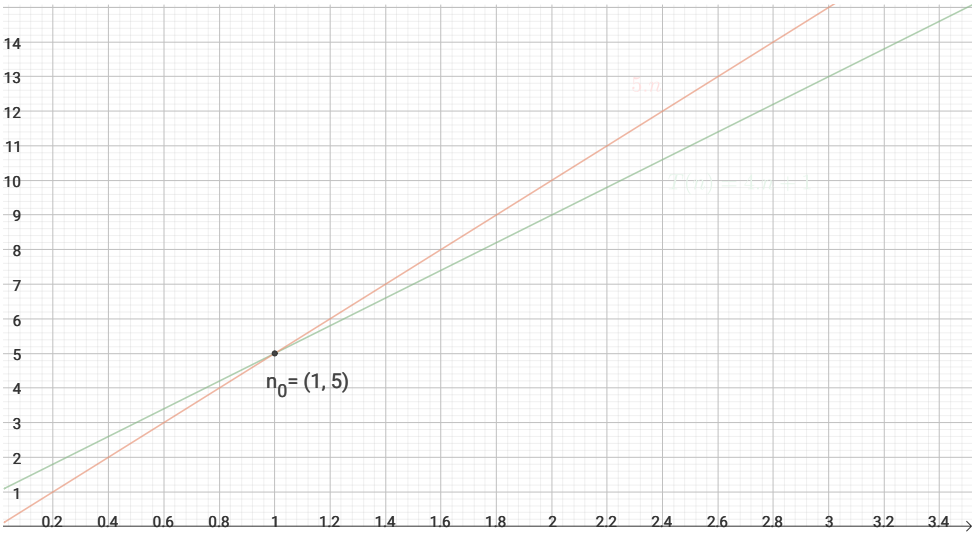


Figura 3.1: Función Acotada: $T(n) \subset O(n)$

A continuación se muestran algunas tasas de crecimiento de las funciones que comúnmente se encuentran al calcular la O grande.

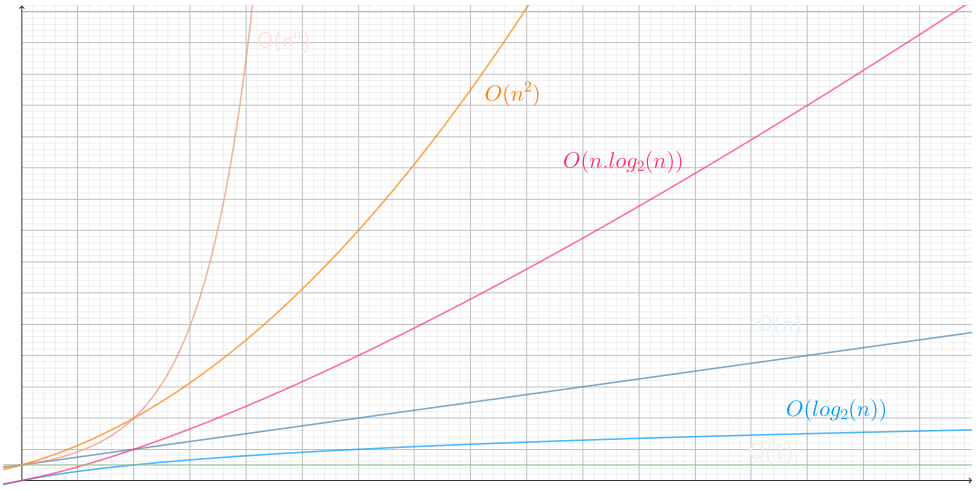


Figura 3.2: Comparación de tasas de crecimiento

En la siguiente tabla se muestran algunas de las funciones más comunes y sus nombres

Orden	Nombre
$O(1)$	constante
$O(log_2(n))$	logarítmica
$O(n)$	lineal
$O(n\ log_2(n))$	casi lineal
$O(n^2)$	cuadrática
$O(k^n)$	exponencial

Propiedades de la O grande

Propiedad transitiva

Si,

$$T(n) = O(f(n)) \land f(n) = O(g(n)) \tag{3.4}$$

entonces:

$$T(n) = O(g(n)) \tag{3.5}$$

Multiplicación por una constante

Si,

$$T(n) = O(f(n)) \land k \neq 0 \tag{3.6}$$

entonces:

$$O(k\,T(n)) = k \cdot O(T(n)) = O(f(n)) \tag{3.7}$$

Propiedad de la suma

Si,

$$T_1(n) = O(f(n)) \land T_2(n) = O(g(n)) \tag{3.8}$$

entonces:

$$T_1(n) + T_2(n) = \max(O(f(n)), O(g(n))) \tag{3.9}$$

Propiedad del producto

Si,

$$T_1(n) = O(f(n)) \land T_2(n) = O(g(n)) \tag{3.10}$$

entonces:

$$T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n)) \tag{3.11}$$

Propiedad de los polinomios

Si,

$$T(n) = c_k\, n^k + c_{k-1}\, n^{k-1} + ... + c_1\, n + c_0 \tag{3.12}$$

es decir $T(n)$ es un polinomio de grado k, entonces:

$$T(n) = O\big(n^k\big) \tag{3.13}$$

Ejemplo de aplicación de las propiedades de la O grande

$T_1(n) = 5n^2 + 3 = O(n^2)$

$T_2(n) = 4n + 1 = O(n)$

$O(T_1(n) + T_2(n)) = \max(O(n^2), O(n)) = O(n^2)$

$O(T_1(n) \cdot T_2(n)) = O(n^2 \cdot n) = O(n^3)$

3.3 Cálculo de la O grande

Como ya se mencionó, para poder calcular el orden de un algoritmo se necesita contar cuantas operaciones elementales realiza.

Operaciones elementales (OE)

Nota

Un valor simple, es un valor que se puede representar en un registro de la computadora y por lo tanto se puede leer y escribir en una única operación, por ejemplo un número entero de una longitud finita, un número en punto flotante, un bit, etc. No son valores simples los

enteros de longitud indefinida o valores compuestos, por ejemplo un objeto compuesto por otros objetos.

Las operaciones elementales son lo que tienen un costo de 1 es decir cuyo tiempo de ejecución es 1 (una unidad genérica de tiempo).

Las siguientes son OE:

- Asignación o consulta de una variable simple.
- Operaciones aritméticas elementales de valores simples (suma, resta, multiplicación, división y resto).
- Comparaciones (mayor, mayor igual, igual, distinto, menor y menor igual).
- Operaciones lógicas (*and*, *or* y *not*).
- Acceso indexado a un elemento simple de un arreglo.
- Desreferenciar un puntero.

Operaciones complejas

Por operaciones complejas entendemos a los condicionales, los ciclos y la ejecución de funciones. Estas operaciones en general pueden anidar otras operaciones.

Condicionales

$$"if"\ \langle C \rangle\ "then"\ \langle S_1 \rangle\ "else"\ \langle S_2 \rangle \tag{3.14}$$

entonces:

$$T(n) = T(C) + \max(T(S_1), T(S_2)) \tag{3.15}$$

Ciclos

$$"for"\ \langle C \rangle\ \{\ \langle S \rangle\ \} \tag{3.16}$$

entonces:

$$T(n) = T(C) + \langle iteraciones \rangle \cdot (T(C) + T(S)) \tag{3.17}$$

Ejecución de Funciones

$$F(P_1, P_2, P_3, ..., P_n)\ \{\ \langle S \rangle\ \} \tag{3.18}$$

entonces:

$$T(n) = 1 + T(P_1) + T(P_2) + T(P_3) + ... + T(P_n) + T(S) \tag{3.19}$$

Ejemplos de cálculo

3.3.3.1 Búsqueda Lineal

En este ejemplo vamos a implementar el algoritmo de búsqueda lineal para buscar un elemento dentro de un arreglo. El algoritmo se puede enunciar como:

Algoritmo: Búsqueda Lineal

Dado un arreglo de elementos (por simplicidad solo números enteros) y un elemento a buscar, se debe recorrer todo el arreglo desde la primera posición hasta la última, hasta encontrar el elemento buscado o determinar que no se encuentra. Si el elemento se encuentra en el arreglo, se devuelve la posición del elemento o −1 en caso contrario.

A continuación una implementación en Go:

Go

```go
func busquedaLineal(arreglo []int, objetivo int) int {
    for i := 0; i < len(arreglo); i++ {
        if arreglo[i] == objetivo {
            return i
        }
    }
    return -1
}
```

Para calcular el tiempo de ejecución de un algoritmo, conviene empezar por las operaciones elementales que se encuentran más anidadas. En este ejemplo la línea 4 dentro del condicional es $O(1)$.

Evaluar la condición `arreglo[i] == objetivo` (línea 3) también es $O(1)$ ya que se trata de accesos a valores y una comparación, por lo tanto:

$$T(\langle condicional \rangle) = O(1) \tag{3.20}$$

Entonces dentro del ciclo `for` tenemos un condicional de $O(1)$ y la actualización del índice (`i++`) que también es una operación simple y por lo tanto es $O(1)$. Podemos concluir que todo el cuerpo del ciclo es $O(1)$.

$$T(\langle ciclo \rangle) = O(1) + n\,O(1) = O(n) \tag{3.21}$$

ya que evaluar la condición del ciclo, `i < len(arreglo)`, también es $O(1)$ (donde n es la longitud del arreglo).

La línea 7 también es $O(1)$.

$$T(n) = O(1) + O(n) + O(1) = O(n) \tag{3.22}$$

3.3.3.2 Búsqueda Binaria

Algoritmo: Búsqueda Binaria

Dado un arreglo de elementos (por simplicidad solo números enteros) que están **ordenados** y un elemento a buscar, se compara el elemento buscado con el elemento que se encuentra en el medio del arreglo. Si el elemento del medio del arreglo es menor que elemento buscado, se descarta la primera mitad del arreglo, en cambio si el elemento del medio es mayor que el elemento buscado, se descarta la mitad superior del arreglo. Si no es menor ni mayor, entonces es igual y encontramos el elemento.

La operación se repite hasta que se encuentra el elemento o ya no se puede seguir partiendo al medio y por lo tanto no se encuentra.

Implementación en Go:

Go

```go
func busquedaBinaria(lista []int, elemento int) int {
    L := 0
    R := len(lista) - 1
    for L <= R {
        M := (L + R) / 2
        if lista[M] < elemento {
            L = M + 1
            continue
        }
        if lista[M] > elemento {
            R = M - 1
            continue
        }
        return M // Se encontró el elemento
    }
    return -1 // No se encontró
}
```

Para analizar la búsqueda binaria, la primera observación que podemos realizar es que los condicionales son $O(1)$. Siguiendo el mismo razonamiento, todas las instrucciones que se realizan fuera del ciclo `for` también son OE. Por lo tanto podemos plantear la siguiente ecuación de recurrencia:

$$T(n) = T\left(\frac{n}{2}\right) + c \tag{3.23}$$

Donde $T(\frac{n}{2})$ representa que en cada vuelta del ciclo mientras se descarta la mitad del arreglo. La constante c representa todas las operaciones $O(1)$ que se realizan en cada vuelta del ciclo. Para resolverla podemos suponer que `n` es una potencia de 2. Es decir:

$$n = 2^k \tag{3.24}$$

Reemplazando obtenemos:

Primera vuelta del ciclo

$$T(n) = T\left(\frac{2^k}{2}\right) + c = T\left(2^{k-1}\right) + c \tag{3.25}$$

Segunda vuelta del ciclo

$$T(n) = T\left(2^{k-2}\right) + 2\,c \tag{3.26}$$

$$\ldots \tag{3.27}$$

Vuelta i del ciclo

$$T(n) = T\left(2^{k-i}\right) + i\,c \tag{3.28}$$

entonces para $i = k$

Vuelta k del ciclo

$$T(n) = T(1) + k\,c \tag{3.29}$$

$T(1)$ es cuanto cuesta si el arreglo tiene tamaño 1, es decir `L == R`, y por lo tanto solo ejecuta una iteración del bucle para determinar si el elemento es el buscado o ajustar los índices para salir. Entonces $T(1) = O(1)$

podemos despejar k de la ecuación $n = 2^k$ y nos queda que $k = log_2(n)$. Finalmente queda:

$$T(n) = O(1) + c\ log_2(n) \tag{3.30}$$

$$T(n) = O(log_2(n)) \tag{3.31}$$

Es decir al tener el arreglo ordenado, la búsqueda binaria necesita realizar mucho menos operaciones que la búsqueda lineal para encontrar el valor buscado o determinar que no existe. La clave está en descartar la mitad del arreglo en cada vuelta del ciclo **`for`**.

3.4 Ejercicios

1. **Máximo entre dos números.** Dado el siguiente pseudocódigo, calculá la cantidad de operaciones elementales que ejecuta y determiná su O grande:

text

```
FUNCION maximo(x, y)
    SI x >= y ENTONCES
        RETORNAR x
    SINO
        RETORNAR y
    FIN SI
FIN FUNCION
```

Algoritmo máximo

1. **Potencia.** Dado el siguiente pseudocódigo, calculá la cantidad de operaciones elementales y determiná su O grande:

text

```
FUNCION potencia(x, n)
    resultado ← 1.0
    PARA i ← 0 HASTA n-1 HACER
        resultado ← resultado * x
    FIN PARA
    RETORNAR resultado
FIN FUNCION
```

Algoritmo potencia

1. **Recorrer un arreglo.** Dado el siguiente pseudocódigo, calculá la cantidad de operaciones elementales y determiná su O grande:

text

```
FUNCION listarAlumnos(a)
    i ← 0
    MIENTRAS i < LONGITUD(a) HACER
        ESCRIBIR a[i].nombre + " " + a[i].apellido
        i ← i + 1
    FIN MIENTRAS
    RETORNAR
FIN FUNCION
```

Algoritmo listar alumnos

1. **Fragmentos de código.** Para cada uno de los siguientes fragmentos, determiná el tiempo de ejecución en notación O grande:

text

```
PARA i ← 0 HASTA n-1 HACER
    sum ← sum + 1
FIN PARA
```

Fragmento 1

text

```
PARA i ← 0 HASTA n-1 PASO 2 HACER
    sum ← sum + 1
FIN PARA
```

Fragmento 2

text

```
PARA i ← 0 HASTA n-1 HACER
    PARA j ← 0 HASTA n-1 HACER
        sum ← sum + 1
    FIN PARA
FIN PARA
```

Fragmento 3

text

```
PARA i ← 0 HASTA n-1 HACER
    sum ← sum + 1
FIN PARA
PARA j ← 0 HASTA n-1 HACER
```

```
            sum ← sum + 1
        FIN PARA
```

Fragmento 4

text

```
PARA i ← 0 HASTA n-1 HACER
    PARA j ← 0 HASTA n*n-1 HACER
        sum ← sum + 1
    FIN PARA
FIN PARA
```

Fragmento 5

text

```
 PARA i ← 0 HASTA n-1 HACER
     PARA j ← 0 HASTA i-1 HACER
         sum ← sum + 1
     FIN PARA
 FIN PARA
```

Fragmento 6

text

```
PARA i ← 0 HASTA n-1 HACER
    PARA j ← 0 HASTA n*n-1 HACER
        PARA k ← 0 HASTA j-1 HACER
            sum ← sum + 1
        FIN PARA
    FIN PARA
FIN PARA
```

Fragmento 7

4. Gestión de memoria

En general la memoria de una computadora se puede clasificar en:

Memoria primaria (o central) Constituida por memoria volátil, más rápida y costosa que otros medios de almacenamiento. Es la memoria de trabajo del procesador, la RAM (*Random Access Memory*) de acceso rápido, donde se encuentran los programas en ejecución y sus datos. La información que almacena la RAM se pierde al interrumpirse el suministro eléctrico.

Memoria secundaria Conformada por el conjunto de memorias no volátiles. Es la memoria de persistencia de información. Ejemplos: discos rígidos, discos ópticos, memorias USB, etc. Conserva la información almacenada al interrumpirse el suministro de corriente eléctrica. Es más lenta y barata que la RAM.

4.1 Organización de la memoria primaria

Tanto los programas como el resto de la información que guarda la computadora se encuentran almacenados en la **memoria secundaria**.

Al lanzar la ejecución de un programa, las instrucciones y los datos iniciales se copian a la **memoria primaria**. Un programa en ejecución se denomina **proceso**.

En tiempo de ejecución, un proceso está asociado a una porción de la memoria primaria que se divide lógicamente en 4 segmentos:

Segmento de código (*code segment*) Es la porción donde se localizan las instrucciones que componen nuestro programa. Su tamaño se determina al comenzar la ejecución. Asociado a este segmento se encuentra un **puntero** que indica la próxima instrucción a ejecutar.

Segmento de datos (*data segment*) Almacena las variables globales y estáticas. Su tamaño también queda determinado al comenzar la ejecución.

Pila (*stack segment*) Almacenará el contenido de las variables locales en cada invocación de una función. Cada entrada en la pila constituye un **marco de datos** (*stack frame*) que representa el contexto de una función en ejecución e incluye variables locales, parámetros y valores de retorno. Se asigna como un bloque de memoria contigua.En Go, el tamaño inicial del *stack* es pequeño (generalmente 2KB) pero no es de tamaño fijo como en C, sino que tiene tamaño dinámico (el runtime lo agranda si hace falta).

Memoria dinámica (*heap*) Es el espacio de memoria que se utiliza para la asignación dinámica de memoria. Su tamaño no está determinado al comenzar la ejecución y se va ajustando a

medida que el programa solicita más memoria para almacenar datos. No se asigna como un único bloque contiguo, sino que puede fragmentarse.

4.2 Ejecución de programas

Estos son algunos de los aspectos a tener en cuenta al momento de ejecutar un programa:

1. **Compilación y almacenamiento.** Los programas se almacenan inicialmente en la memoria secundaria (por ejemplo, en un disco duro o SSD). En el caso de Go como se trata de un lenguaje compilado, los archivos fuentes se compilan y se genera un archivo ejecutable que se almacena en el disco.
2. **Asignación de memoria central** Al iniciar la ejecución, el **sistema operativo** carga el programa en la memoria central (RAM). Esto incluye las instrucciones del programa y los datos iniciales necesarios para su ejecución. La memoria asignada al programa se divide en segmentos específicos para el código, los datos, el *stack* y el *heap*. El programa en ejecución se denomina **proceso**.
3. **Ejecución del programa** El procesador ejecuta una a una las instrucciones del programa desde el **segmento de código**. Cada vez que ejecuta una instrucción, avanza el puntero de instrucción a la siguiente instrucción. Cuando ejecuta una llamada a una función, se crea un nuevo *frame* en el *stack* para almacenar las variables locales y los parámetros de la función. Al terminar la función, el *stack frame* se elimina y el valor de retorno se transfiere al marco anterior desde donde se llamó a la función. Si durante la ejecución de una función se solicita memoria dinámica, se asigna en el *heap*.En Go, durante la ejecución del programa, el recolector de basura (*garbage collector*) se encarga de liberar la memoria no utilizada en el *heap*, evitando así las fugas de memoria (*memory leaks*).
4. **Interacción con el sistema operativo** El sistema operativo supervisa y gestiona la memoria asignada al programa. Si el programa necesita más memoria, puede solicitarla al sistema operativo, que ajustará el tamaño del *heap* o el *stack* según sea necesario.
5. **Liberación de memoria** Al finalizar la ejecución, el sistema operativo libera toda la memoria asignada al programa, incluyendo los segmentos de código, datos, pila y heap.

En la siguiente figura se muestra un esquema de la memoria de un proceso en ejecución. Cada segmento de memoria tiene un tamaño y una función específica en el programa. La figura es solo a modo didáctico y no representa la organización real de la memoria en Go, que es más compleja.

En el diagrama el *stack* se ubica en la parte superior de la memoria y crece hacia abajo; cuando no puede crecer más se produce un error de desbordamiento de pila (*stack overflow*). El *heap* se ubica en la parte inferior de la memoria, sobre los segmentos de código y datos y crece hacia arriba.

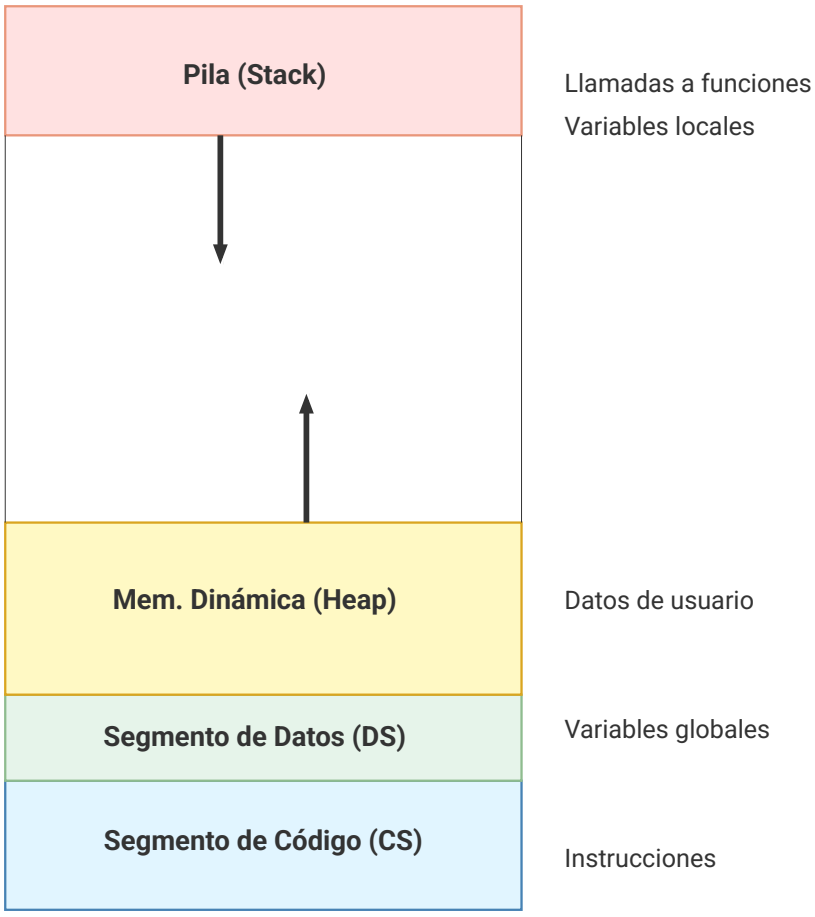


Figura 4.1: Segmentos de memoria de un proceso en ejecución

4.3 Gestión de memoria dinámica en Go

La gestión de memoria es un aspecto clave en cualquier lenguaje de programación, ya que impacta en el rendimiento, la eficiencia y la estabilidad del software.

En Go, las variables se almacenan en el *stack* o en el *heap* dependiendo de su alcance, duración y cómo se utilizan.

El compilador de Go decide automáticamente si una variable debe almacenarse en el *stack* o en el *heap*. Esto se conoce como *Escape Analysis*. Si una variable **"escapa"** del alcance de la función, se almacena en el *heap* en lugar del *stack*.

Consideraciones de rendimiento

El acceso a las variables que se encuentran en el *stack* es más directo y más rápido, mientras que el acceso a los datos en el *heap* es más lento ya que se deben referenciar desde el *stack*, pero permite estructuras de datos más grandes y persistentes.

Veamos un ejemplo, dado el siguiente fragmento de código:

Go

```
type Direccion struct {
    calle, ciudad, provincia string
    numero                   uint
}

type Persona struct {
    nombre, apellido string
    edad             uint
    direccion        Direccion
}

var num int = 5
var p1 = Persona{"Marcelo", "Díaz", 27, Direccion{"Mariano Acosta",
                 "Gonzalez Catán", "Buenos Aires", 6420}}

func main() {
    p2 := Persona{nombre: "Pepe", edad: 23}
    p3 := &p2
    p4 := Persona{"Juan", "Gonzalez", 34, Direccion{"Valentín Gómez",
                 "Caseros", "Buenos Aires", 742}}

    p3.direccion := Direccion{"Av. Corrientes", "CABA", "Buenos Aires", 1050}
    p3.apellido = "Martinez"
    p4.direccion = Direccion{"Mariano Acosta", "Gonzalez Catán",
                             "Buenos Aires", 6420}
}
```

El **Stack**, el **Heap** y el **Segmento de Datos** presentarán el siguiente estado:

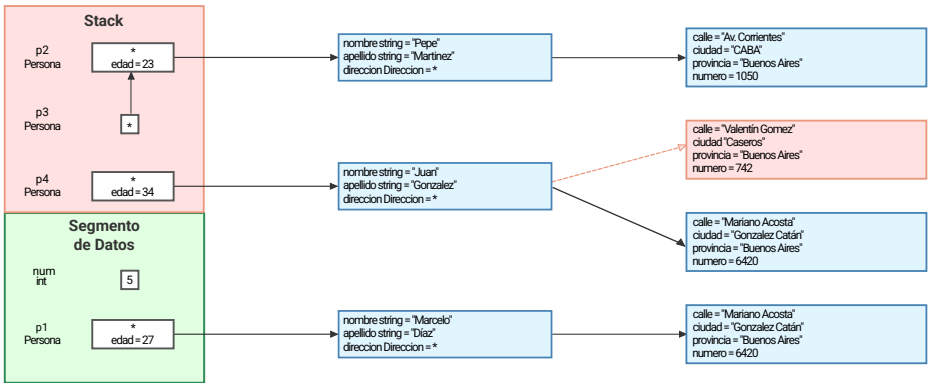


Figura 4.2: Mapa de Memoria

Vamos a analizar cada variable en el código y justificar dónde se almacena:

- **`num`**

Go

```
var num int = 5
```

- **Almacenamiento: Segmento de Datos**

- **Justificación:** `num` es una variable global (declarada a nivel de paquete). Al no ser una variable local, no se ubica en el *stack* de ninguna función, sino en el segmento de datos estáticos definido al inicio del proceso.

- **p1**

Go

```
var p1 = Persona{"Marcelo", "Díaz", 27, Direccion{"Mariano Acosta",
                "Gonzalez Catán", "Buenos Aires", 6420}}
```

- **Almacenamiento: Segmento de Datos** (estructura) y **Heap** (cadenas)
- **Justificación:** `p1` es una variable global, que se encuentra en el segmento de datos, es de tipo `Persona` y ocupa un espacio de tamaño fijo para almacenar toda la estructura. Sin embargo, los valores de los campos de tipo `string` se almacenan en el *heap*, ya que en Go los *strings* son cabeceras que apuntan a una secuencia de bytes en memoria dinámica, mientras que `edad`, que es del tipo `uint` se almacena también en el **Segmento de Datos**.

- **p2**

Go

```
p2 := Persona{nombre: "Pepe", edad: 23}
```

- **Almacenamiento: Stack** (estructura) y **Heap** (cadenas)
- **Justificación:** `p2` es una variable local definida dentro de `main`, por lo que se crea en el marco de pila (*stack frame*) de dicha función. Al igual que con `p1`, el contenido de los campos de tipo `string` (“Pepe”) se almacena en el *heap*.

- **p3**

Go

```
p3 := &p2
```

- **Almacenamiento: Stack**
- **Justificación:** `p3` es un puntero a `p2`, definido como variable local en `main`. Se almacena en el *stack* y apunta a la misma estructura `Persona` que `p2`. A través de `p3` se modifican `p2.apellido` a `"Martinez"` y `p2.direccion` a `Direccion{"Av. Corrientes", "CABA", "Buenos Aires", 1050}`.

- **p4**

Go

```
p4 := Persona{"Juan", "Gonzalez", 34, Direccion{"Valentín Gómez",
              "Caseros", "Buenos Aires", 742}}
```

- **Almacenamiento: Stack** (estructura) y **Heap** (cadenas)
- **Justificación:** `p4` es una variable local de tipo `Persona` que se almacena en el *stack*. Los valores de tipo `string` (`"Juan"`, `"Gonzalez"`, etc.) se almacenan en el *heap*. Posteriormente, se reasigna su campo `direccion` con `Direccion{"Mariano Acosta", "Gonzalez Catán", "Buenos Aires", 6420}`.

Resumen general:

Variable	Ubicación	Justificación
`num`	Segmento de Datos	Variable global simple.
`p1`	Segmento de Datos (header) / *Heap* (str)	Estructura global, contenido de cadenas en heap.
`p2`	*Stack* (estructura) / *Heap* (cadenas)	Variable local, contenido de cadenas en heap.
`p3`	*Stack*	Puntero local a `p2`.
`p4`	*Stack* (estructura) / *Heap* (cadenas)	Variable local, contenido de cadenas en heap.

En Go, el **compilador** y el **recolector de basura** (GC) optimizan el uso del *stack* y el *heap*. Las estructuras simples y de corta duración suelen estar en el stack, mientras que los datos más complejos o de mayor duración (como cadenas) se almacenan en el heap.

Go utiliza un **recolector de basura concurrente** para liberar memoria automáticamente. Es concurrente porque la mayor parte del trabajo de limpieza ocurre en paralelo con la ejecución del programa. Esto permite que el programador no tenga que liberar la memoria manualmente, evitando errores comunes como el acceso a memoria inválida.

Cuando existen datos en la memoria dinámica que ya no son accesibles (no están referenciados desde la pila ni el segmento de datos), el GC los marca para su eliminación y eventual liberación.

Para coordinar ciertos pasos del proceso de limpieza, el GC realiza pausas extremadamente breves denominadas *stop-the-world* (generalmente menores a un milisegundo). Go se destaca por optimizar estas pausas para que no interfieran con el rendimiento percibido del sistema:

- Minimiza las pausas para optimizar la latencia.
- Usa múltiples núcleos de CPU para ejecutar la recolección en paralelo.
- La mayor parte del marcado de objetos ocurre mientras el programa sigue corriendo.
- Detecta y elimina referencias a objetos no utilizados de forma eficiente.

Importante

Un GC concurrente mejora el rendimiento y la experiencia del usuario, ya que evita grandes pausas en la ejecución del programa. Esto es fundamental en servidores y sistemas en tiempo real, donde una pausa larga podría afectar la respuesta del sistema.

Ejercicios

1. **Mapa de memoria.** Dado el siguiente código, dibujá un esquema de la memoria indicando en qué segmento (datos, stack o heap) se almacena cada variable y su contenido:

Go

```go
var global int = 42

type Punto struct {
    X, Y int
}

func main() {
    a := 10
    b := &a
    c := Punto{X: 3, Y: 7}
    d := "hola"
}
```

1. **Escape analysis.** Para cada una de las siguientes funciones, indicá si la variable creada escapa al heap o se queda en el stack. Justificá:

Go

```go
func f1() int {
    x := 5
    return x
}

func f2() *int {
    x := 5
    return &x
}

func f3() {
    s := make([]int, 10)
    s[0] = 1
    fmt.Println(s[0])
}
```

1. **Strings en heap.** Explicá por qué el contenido de un `string` en Go siempre se almacena en el heap, incluso cuando la variable que lo contiene es local. ¿Qué parte de la variable queda en el stack?
2. **Stack frames.** Dada la secuencia de llamadas `main()` -> `calcular()` -> `sumar(a, b int)`, describí el contenido del stack en cada paso. Indicá qué datos contiene cada stack frame (parámetros, variables locales, dirección de retorno).
3. **Garbage collector.** Explicá brevemente:

- Qué problema resuelve el garbage collector concurrente de Go.
- Qué significa *stop-the-world* y por qué las pausas son del orden de microsegundos en Go.
- Qué ventaja tiene un GC concurrente frente a uno no concurrente.

5. Estructuras de Datos

5.1 Tipos Abstractos de Datos

Definición

Podemos imaginar un Tipo Abstracto de Datos (TAD) o *Abstract Data Type* (ADT) en inglés, como una **caja negra** que contiene valores y define un conjunto de operaciones para manipularlos. Quien usa el TAD solo conoce **qué** operaciones puede realizar, pero no **cómo** están implementadas internamente.

Definición

Un **TAD** es un modelo matemático caracterizado por:

- **Valores**: los datos que se pueden almacenar y manipular.
- **Operaciones**: las acciones que se pueden realizar sobre esos datos.

En un TAD distinguimos dos capas: la **estructura interna** donde realmente se almacenan los datos, y la **interfaz** que expone hacia afuera las operaciones disponibles. Esta separación es fundamental: quien usa el TAD solo necesita conocer la interfaz, no los detalles internos.

Por ejemplo, un conjunto es un TAD que permite agregar elementos, consultar si un elemento pertenece o eliminarlos. No tenemos acceso directo a la estructura interna que lo implementa, solo podemos usar las operaciones que define su interfaz.

Características de un TAD

5.1.2.1 Abstracción

La abstracción implica definir un conjunto de operaciones que pueden realizarse sobre un tipo de dato, sin revelar cómo se implementan internamente.

5.1.2.2 Ocultamiento de información

Es el mecanismo que permite ocultar los detalles de implementación, encapsulando los datos y las operaciones en una única estructura y exponiendo solo una interfaz bien definida.

La **abstracción** se logra mediante el **ocultamiento de información**.

5.1.2.3 Interfaz

Es la parte visible del TAD, que define cómo se puede interactuar con él. La interfaz es el **contrato** que el TAD ofrece a sus usuarios.

5.1.2.4 Comportamiento

Es el conjunto de operaciones que se pueden realizar sobre un TAD y los efectos que producen.

El **comportamiento** de un TAD se define a través de su **interfaz**.

5.1.2.5 Invariante de representación

El **invariante de representación** (o simplemente *invariante*) es una condición lógica que debe cumplirse para todo estado **válido** del TAD. Es como una regla interna que garantiza la integridad y coherencia de la estructura de datos.

Por ejemplo, en un TAD que modela un `Reloj` de 24 horas, el invariante podría ser que la hora esté siempre entre 0 y 23. Si alguna primitiva dejara la hora en 25, el invariante estaría roto.

> **Primitivas**
>
> Cada operación del TAD se denomina **primitiva**. Las primitivas son las únicas operaciones disponibles para manipular el estado interno del TAD.

5.1.2.5.1 El invariante durante la ejecución de una primitiva

El invariante debe cumplirse **siempre**, con una salvedad importante: **durante la ejecución de una primitiva el invariante puede dejar de cumplirse temporalmente**. Esto es aceptable siempre que la primitiva lo **restablezca** antes de finalizar.

Supongamos un TAD con dos campos `a` y `b`, donde el invariante es `a == b`. Si una primitiva ejecuta primero `a = a + 1` y luego `b = b + 1`, entre ambas operaciones el invariante no se cumple. Esto es válido siempre que al terminar la primitiva la igualdad se cumpla nuevamente.

5.1.2.5.2 Atomicidad de las primitivas

Las primitivas deben ser **atómicas**: deben ejecutarse por completo o no tener ningún efecto. Si una primitiva falla a mitad de camino, el TAD debe quedar en el mismo estado válido en el que estaba antes de ejecutarla. En otras palabras, una primitiva debe dejar el invariante intacto, sin importar si la operación se completó exitosamente o no.

> **Importante**
>
> Un TAD cuyas primitivas no son atómicas puede dejar el invariante en un estado inconsistente. Decimos entonces que el TAD está **roto**.

Ejemplo paso a paso: TAD Contador

Vamos a construir un TAD `Contador` que mantiene un valor entero que puede incrementarse o decrementarse dentro de un rango fijo.

5.1.3.1 1. Definir los valores

El Contador necesita tres valores enteros:

- `minimo`: el valor más chico que puede tomar el contador.

- `maximo`: el valor más grande que puede tomar el contador.
- `actual`: el valor actual del contador.

Además, para analizar el invariante vamos a llevar la cuenta de cuántas veces se modificó el contador exitosamente:

- `cambios`: cantidad total de modificaciones realizadas.

5.1.3.2 2. Definir las operaciones (primitivas)

Primitiva	Descripción
`NuevoContador(min, max)`	Crea un contador con `actual = min`
`Incrementar()`	Aumenta `actual` en 1, si no supera `maximo`
`Decrementar()`	Disminuye `actual` en 1, si no es menor que `minimo`
`Valor()`	Devuelve el valor actual
`Cambios()`	Devuelve la cantidad de modificaciones

5.1.3.3 3. Definir el invariante

El invariante del Contador tiene dos partes:

1. `minimo <= actual <= maximo`: el valor actual siempre está dentro del rango.
2. `cambios` es igual a la cantidad de veces que se ejecutaron `Incrementar` o `Decrementar` con éxito.

5.1.3.4 4. Analizar las transiciones de estado

Analicemos cada primitiva y cómo afecta al invariante.

Constructor `NuevoContador(min, max)`:

```
estado inicial:  (ninguno)
   ↓  validar que min <= max
   ↓  actual = min
   ↓  cambios = 0
estado final:    actual = min, cambios = 0
```

Después del constructor, el invariante se cumple: `min <= actual <= max` y `cambios` es 0 (todavía no hubo modificaciones).

Primitiva `Incrementar()`:

```
estado inicial:  min <= actual <= max, cambios = n
   ↓  ¿actual < maximo?
   │   ├── No → devolver error (el estado no cambia)
   │   └── Sí → actual++              ←  el invariante se mantiene
   │            cambios++             ←  ¡el invariante podria romperse!
   ↓                                  ←  (explicacion a continuacion)
estado final:    min <= actual <= max, cambios = n + 1
```

¿En qué momento podría no cumplirse el invariante? En esta implementación simple, tanto `actual++` como `cambios++` son operaciones elementales y al finalizar la primitiva el invariante se cumple. El punto importante es que el orden de las operaciones importa: si primero incremen-

táramos `cambios` y después `actual`, y la primitiva fallara entre medio, quedaríamos con `cambios` desactualizado. Por eso las primitivas deben diseñarse para que, si algo sale mal, el estado vuelva atrás.

5.1.3.5 5. Implementación en Go

Go

```go
package contador

import "errors"

type Contador struct {
  actual  int
  minimo  int
  maximo  int
  cambios int
}

func NuevoContador(min, max int) (*Contador, error) {
  if min > max {
    return nil, errors.New("minimo no puede ser mayor que maximo")
  }
  return &Contador{
    actual:  min,
    minimo:  min,
    maximo:  max,
    cambios: 0,
  }, nil
}

func (c *Contador) Incrementar() error {
  if c.actual >= c.maximo {
    return errors.New("el contador ya alcanzó el máximo")
  }
  c.actual++
  c.cambios++
  return nil
}

func (c *Contador) Decrementar() error {
  if c.actual <= c.minimo {
    return errors.New("el contador ya alcanzó el mínimo")
  }
  c.actual--
  c.cambios++
  return nil
}

func (c *Contador) Valor() int {
  return c.actual
}

func (c *Contador) Cambios() int {
  return c.cambios
}
```

Código en el repositorio

Este ejemplo completo con tests está disponible en `taller-tad/01-tipos-abstractos-de-datos/ejemplos/contador/`.

5.1.3.6 6. Verificar el invariante en cada primitiva

`NuevoContador`: valida que `min <= max` antes de crear el contador. Si la validación falla, devuelve error y no se crea ningún contador (atomicidad). Si tiene éxito, `actual = min` y `cambios = 0`. Invariante: ✅.

`Incrementar`: verifica que `actual < maximo`. Si no, devuelve error sin modificar nada (atomicidad). Si sí, incrementa `actual` y `cambios`. Al terminar, `actual` sigue siendo `<= maximo` (porque verificamos antes) y `cambios` aumentó en 1. Invariante: ✅.

`Decrementar`: análogo a `Incrementar` pero con `minimo`. Invariante: ✅.

`Valor` y `Cambios`: solo lectura, no modifican el estado. Invariante: ✅.

5.1.3.7 7. Atomicidad en la práctica

Observá el patrón que usan `Incrementar` y `Decrementar`:

1. **Validar** la condición que protege el invariante.
2. Si no se cumple -> **error** sin modificar nada.
3. Si se cumple -> **modificar** el estado.

Este patrón garantiza atomicidad: la primitiva se ejecuta por completo (modifica el estado) o no tiene efecto (devuelve error). Nunca queda un estado intermedio donde el invariante esté roto.

Nota

En Go, el constructor devuelve un puntero `*Contador` y un `error`. Esta convención permite que el constructor valide los parámetros y, si algo falla, devuelva un error sin haber creado ningún estado inconsistente.

Go: *struct* e *interface* para definir TADs

Go cuenta con *struct* e *interface* para definir nuevos tipos abstractos de datos. Las estructuras nos permiten definir un conjunto de valores y operaciones asociadas. Algunas operaciones pueden ser públicas (forman parte de la interfaz del TAD) y otras privadas (para uso interno). Las interfaces permiten que varios tipos que presentan el mismo comportamiento puedan manipularse como un único tipo.

En el ejemplo del Contador, `Contador` es un *struct* con campos en minúscula (`actual`, `minimo`, `maximo`, `cambios`). Al estar en minúscula son privados: nadie fuera del paquete `contador` puede accederlos directamente. Las funciones y métodos con mayúscula (`NuevoContador`, `Incrementar`, `Valor`) son públicos: forman la interfaz del TAD.

Repositorio del taller de TAD

Los ejemplos y ejercicios de este capítulo están en el repositorio `untref-ayp2/taller-tad`. Está organizado por capítulo (`01-tipos-abstractos-de-datos/`, `02-pilas-colas/`, etc.) y dentro de cada uno hay dos carpetas:

- **`ejemplos/`**: programas completos que ilustran los conceptos del apunte.
- **`ejercicios/`**: esqueletos de código con partes incompletas para completar y tests automatizados para validar la solución.

Para ejecutar los tests de un ejercicio:

Shell

```
cd taller-tad/01-tipos-abstractos-de-datos/ejercicios/01-fraccion
go test -v
```

Ejercicios

Los ejercicios de este capítulo están en `01-tipos-abstractos-de-datos/ejercicios/` del repositorio taller-tad.

Este repositorio incluye las estructuras definidas en data-structures como un subdirectorio. Antes de resolver los ejercicios, asegurate de tener implementadas las estructuras necesarias en `data-structures/`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.2 Pilas y Colas

Los primeros TAD que vamos a estudiar son **Pilas** y **Colas**.

Pilas y Colas son estructuras de datos **dinámicas** que mantienen el orden de los elementos y donde los elementos que se agregan y remueven tienen una ubicación específica.

> **TAD dinámico**
>
> Un TAD dinámico es un TAD que puede modificar su tamaño, es decir, el espacio en memoria que ocupa, en función de la cantidad de datos que almacena.

Pila

Es una estructura del tipo **LIFO** (*Last In First Out*) es decir el último elemento que ingresó en la pila será el primer elemento en salir. Por ejemplo, en una pila de libros, si queremos agregar un nuevo libro debemos colocarlo encima de la pila, sobre el último libro. A su vez el último libro de la pila es el único al que le podemos ver su portada y sacar de la pila. Por lo tanto para sacar el libro de abajo de la pila, primero hay que sacar uno por uno todos los libros que están apilados sobre él.

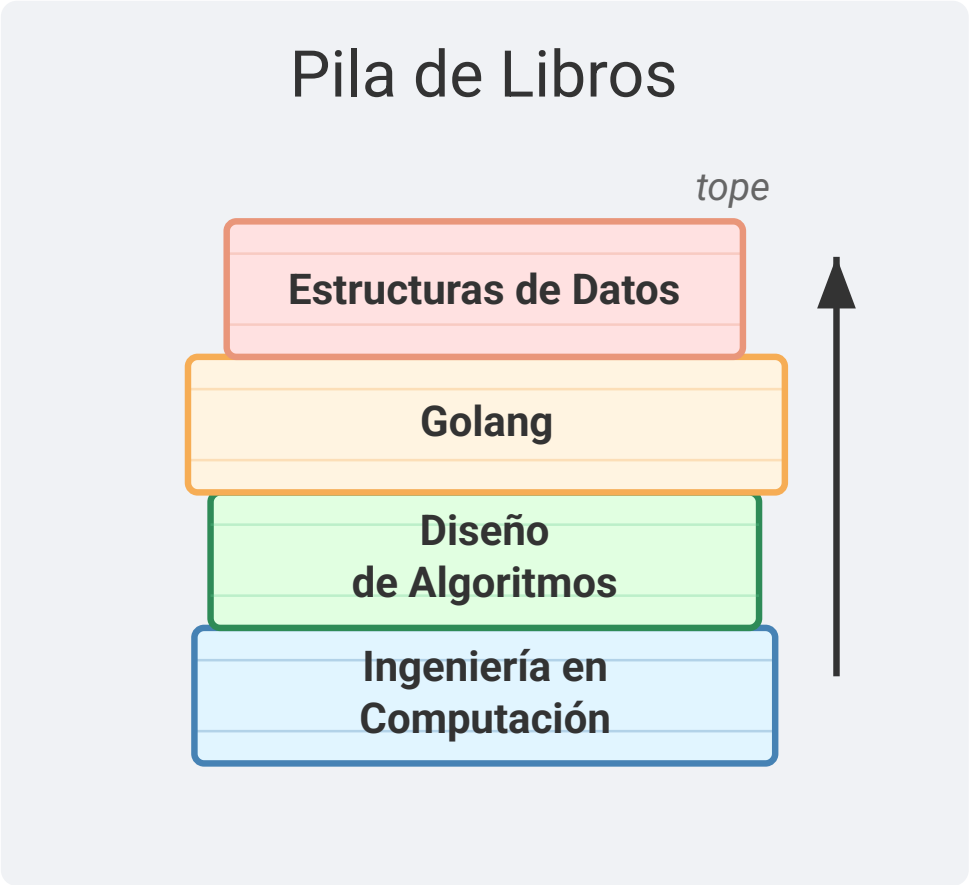


Figura 5.1: Pila de Libros

A partir de la descripción del comportamiento de la pila de libros, podemos intuir el comportamiento de la estructura Pila o Stack:

`Push` Permite insertar un elemento en la pila (siempre en el tope de la misma)

`Pop` Permite extraer un elemento de la pila (siempre el que está en el tope). Si se intenta hacer un Pop de una pila vacía se debe indicar un error

`Top` Permite ver el último elemento que ingresó en la pila, sin sacarlo. Si se intenta hacer un Top de una pila vacía se debe indicar un error

`IsEmpty` Verifica si la pila está vacía. Devuelve `true` si la pila está vacía o `false` en caso contrario

Todas las operaciones deben ser $O(1)$. Es decir de tiempo constante, independiente del tamaño de la entrada.

Este comportamiento define la **interfaz** del tipo Pila, es decir las operaciones válidas que se pueden realizar sobre la misma. En código:

Go

```go
type Stack[T any] interface {
    Push(val T)
    Pop() (T, error)
    Top() (T, error)
    IsEmpty() bool
}
```

El parámetro de tipo `[T any]` hace que la interfaz sea **genérica**: al crear una pila concreta se elige un tipo fijo (`Stack[int]`, `Stack[string]`, etc.) y esa pila solo aceptará elementos de ese tipo. No se pueden mezclar tipos distintos en una misma pila, como sí se podía con el tipo `any` sin parámetro.

Encapsulamiento

Es fundamental ocultar la implementación para que no se pueda manipular el contenedor de datos con operaciones no permitidas. Por ejemplo, la definición:

Go

```go
// Forma incorrecta de definir una pila,
// porque no encapsula el contenedor de datos
type Stack[T any] []T
```

deja expuesto el contenedor de datos y se podría manipular con operaciones propias de slices.

Go

```go
var p Stack[int]
p = append(p, 1)
p = append(p, 2)
p[0] = 99
```

En este fragmento se agregan elementos a la pila usando la función `append` de Go y luego se modifica un elemento arbitrariamente.

El siguiente seudocódigo muestra una implementación del TAD Pila sobre un arreglo dinámico. El contenedor de datos `data` está encapsulado dentro del struct y su tipo depende del parámetro genérico `T`. Las operaciones `Push`, `Pop` y `Top` trabajan siempre sobre el extremo del arreglo, garantizando costo $O(1)$.

text

```
TIPO Stack[T] struct
    data ← ARREGLO de T
FIN TIPO

FUNCION NewStack[T]() → *Stack[T]
    RETORNAR &Stack[T]{}
FIN FUNCION

FUNCION (s *Stack[T]) IsEmpty() → bool
    RETORNAR LONGITUD(s.data) = 0
FIN FUNCION

FUNCION (s *Stack[T]) Push(x T)
    s.data ← AGREGAR_AL_FINAL(s.data, x)
FIN FUNCION

FUNCION (s *Stack[T]) Pop() → (T, error)
    SI s.IsEmpty() ENTONCES
        RETORNAR valorCero, error("pila vacía")
    FIN SI
    x ← s.data[LONGITUD(s.data) - 1]
    s.data ← s.data[0 : LONGITUD(s.data) - 1]
    RETORNAR x, nil
FIN FUNCION

FUNCION (s *Stack[T]) Top() → (T, error)
    SI s.IsEmpty() ENTONCES
        RETORNAR valorCero, error("pila vacía")
    FIN SI
    RETORNAR s.data[LONGITUD(s.data) - 1], nil
FIN FUNCION
```

`NewStack` Crea una pila vacía. Reserva espacio en memoria para almacenar la pila y devuelve la dirección de memoria correspondiente.

`IsEmpty` Verifica si la cantidad de elementos del contenedor de datos es igual a 0.

`Push` Agrega el elemento recibido al final del arreglo.

Pop Si la pila está vacía devuelve el valor cero de `T` y un error. Caso contrario devuelve el elemento del tope y lo elimina de la pila, reduciendo el arreglo en una posición.

Top Similar a `Pop` pero no elimina el tope. Si la pila está vacía devuelve error.

A continuación un ejemplo de uso con `Stack[int]`:

Go

```go
import "fmt"

func main() {
    // Crear una pila de enteros
    s := NewStack[int]()

    // Agregar elementos
    s.Push(10)
    s.Push(20)
    s.Push(30)

    // Consultar el tope sin extraerlo
    top, _ := s.Top()
    fmt.Println("Tope:", top)

    // Extraer todos los elementos
    for !s.IsEmpty() {
        val, _ := s.Pop()
        fmt.Println(val)
    }
}
```

Y el mismo `Stack[T]` puede usarse con `string`:

Go

```go
func main() {
    pila := NewStack[string]()
    pila.Push("uno")
    pila.Push("dos")
    pila.Push("tres")

    for !pila.IsEmpty() {
        val, _ := pila.Pop()
        fmt.Println(val)
    }
}
```

Cola

Es una estructura del tipo **FIFO (First In First Out)** es decir el primer elemento en ingresar en la cola es el primero en salir. Un ejemplo clásico del uso de esta estructura es para modelar una

cola de personas, por ejemplo en la caja de un supermercado. La última persona que llega se coloca al final de la cola y espera su turno para ser atendida.

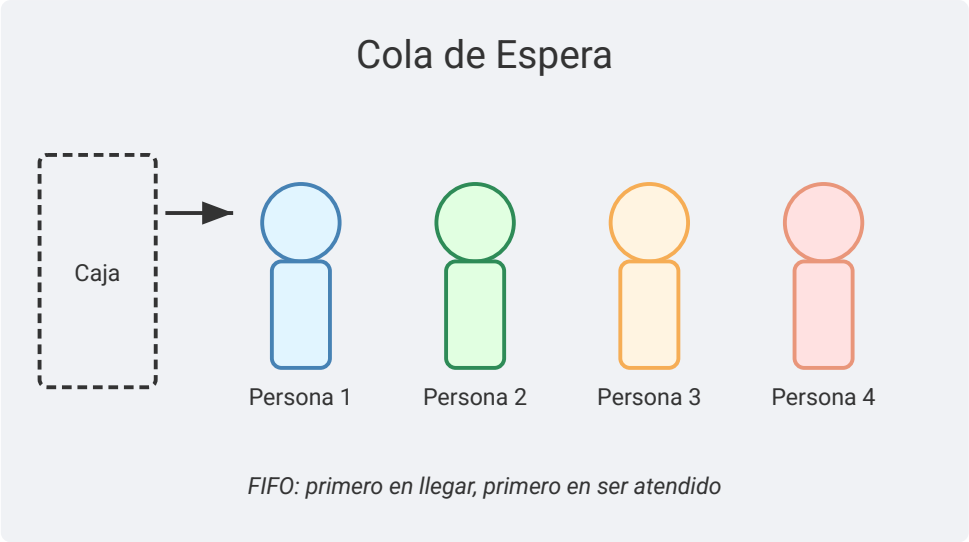


Figura 5.2: Cola de Espera

El comportamiento de la cola queda definido por las siguientes operaciones:

`Enqueue` Permite insertar un elemento en la cola (siempre en la última posición)

`Dequeue` Permite extraer un elemento de la cola (siempre el primero). Si se intenta hacer un Dequeue de una cola vacía se debe indicar un error

`Front` Permite ver el primer elemento de la cola sin extraerlo. Si se intenta hacer un Front de una cola vacía se debe indicar un error

`IsEmpty` Verifica si la Cola está vacía. Devuelve `true` si la cola está vacía o `false` en caso contrario

Todas las operaciones deben ser $O(1)$.

Su interfaz en código es:

Go

```go
type Queue[T any] interface {
    Enqueue(val T)
    Dequeue() (T, error)
    Front() (T, error)
    IsEmpty() bool
}
```

El siguiente seudocódigo muestra una implementación sobre un arreglo dinámico:

text

```
TIPO Queue[T] struct
    data ← ARREGLO de T
FIN TIPO
```

```
FUNCION NewQueue[T]() → *Queue[T]
    RETORNAR &Queue[T]{}
FIN FUNCION

FUNCION (q *Queue[T]) Enqueue(x T)
    q.data ← AGREGAR_AL_FINAL(q.data, x)
FIN FUNCION

FUNCION (q *Queue[T]) Dequeue() → (T, error)
    SI q.IsEmpty() ENTONCES
        RETORNAR valorCero, error("cola vacía")
    FIN SI
    x ← q.data[0]
    q.data ← q.data[1 : LONGITUD(q.data)]
    RETORNAR x, nil
FIN FUNCION

FUNCION (q *Queue[T]) Front() → (T, error)
    SI q.IsEmpty() ENTONCES
        RETORNAR valorCero, error("cola vacía")
    FIN SI
    RETORNAR q.data[0], nil
FIN FUNCION

FUNCION (q *Queue[T]) IsEmpty() → bool
    RETORNAR LONGITUD(q.data) = 0
FIN FUNCION
```

Ejemplo de uso:

Go

```go
func main() {
    cola := NewQueue[string]()

    cola.Enqueue("primero")
    cola.Enqueue("segundo")
    cola.Enqueue("tercero")

    front, _ := cola.Front()
    fmt.Println(front)  // primero

    for !cola.IsEmpty() {
        val, _ := cola.Dequeue()
        fmt.Println(val)
    }
}
```

Ejercicios

Los ejercicios de este capítulo están en `02-pilas-colas/ejercicios/` del repositorio taller-tad.

Antes de resolverlos hay que tener implementados `SliceStack[T]` y `SliceQueue[T]` en data-structures, que está dentro del repositorio `taller-tad` y contiene las interfaces y los esqueletos. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.3 Listas Enlazadas

Las listas enlazadas son estructuras de datos que permiten almacenar una colección de elementos en posiciones de memoria **no necesariamente contiguas**. Cada elemento se guarda en un nodo que contiene un campo de dato y uno o dos punteros a otros nodos. Los nodos se enlazan entre sí para formar la secuencia.

Son estructuras **dinámicas**: pueden crecer a medida que se agregan datos y reducirse cuando se eliminan. A diferencia de los arreglos, no requieren bloques contiguos de memoria ni reasignaciones costosas al cambiar de tamaño.

Algunos usos frecuentes:

Algoritmos de manipulación de texto Las listas enlazadas son útiles en procesamiento de texto, donde la inserción y eliminación de caracteres o líneas se realizan con frecuencia. Cada línea del documento puede ser un nodo; insertar o borrar una línea solo requiere reenlazar punteros, sin desplazar el resto del contenido.

Undo y redo en aplicaciones Las listas enlazadas dobles permiten navegar hacia adelante y atrás en el historial de acciones. Cada acción es un nodo; la doble vinculación permite moverse en ambas direcciones en tiempo constante.

Listas de reproducción Una lista circular permite reproducir canciones en ciclo infinito. Al llegar al final, se vuelve al principio sin necesidad de reiniciar manualmente. Algunas implementaciones combinan listas dobles (para avanzar/retroceder) con comportamiento circular (para repetición continua).

Existen cuatro variantes principales de listas enlazadas, que se diferencian en la cantidad de punteros por nodo, cómo se marcan los extremos y cómo se recorren. Cada variante tiene fortalezas y debilidades según la operación que se quiera optimizar.

Lista Enlazada Simple

Definición

Una lista enlazada simple es una estructura lineal donde cada nodo tiene un **sucesor**, salvo el último cuyo puntero es `nil`. La lista vacía no contiene nodos y su tamaño es 0.

El nodo almacena:

- **Dato**: el valor del elemento.
- **Siguiente**: puntero al próximo nodo de la secuencia.

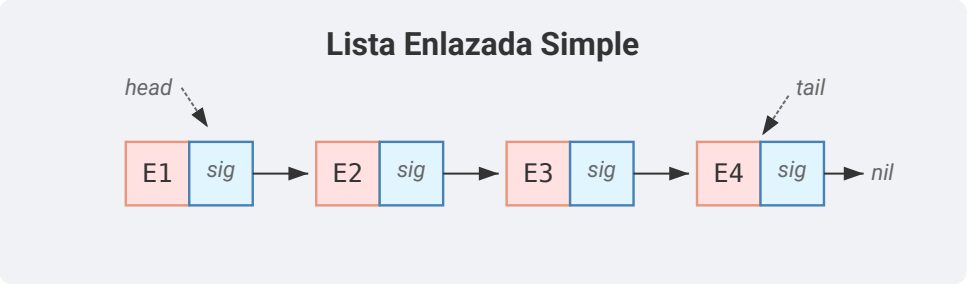


Figura 5.3: Lista Enlazada Simple

Go

```go
type node[T any] struct {
    data T
    next *node[T]
}
```

Go

```go
type List[T comparable] struct {
    head *node[T]
    tail *node[T]
    size int
}
```

Notar que el nodo se parametriza con `[T any]` porque solo almacena y enlaza datos, sin necesidad de compararlos. La lista, en cambio, usa `[T comparable]`, ya que, como veremos a continuación, algunas operaciones necesitan comparar elementos con `==` para encontrar un dato buscado.

Cuando una operación como `Head()` o `Tail()` se ejecuta sobre una lista vacía, debe devolver el **valor cero** del tipo `T` (no `nil`, que solo es válido para punteros, *slices*, *maps* y canales). En Go, el valor cero se obtiene con `var zero T`:

Go

```go
var zero T
/*
    0       para int
    0.0     para float
    ""      para string
    false   para bool
    nil     para punteros, slices, maps y canales
*/
```

Lista Enlazada Doble

Definición

Una lista enlazada doble es una estructura lineal donde cada nodo tiene un **sucesor** y un **predecesor**, salvo el primero (sin predecesor) y el último (sin sucesor).

El nodo almacena:

- **Dato**: el valor del elemento.
- **Siguiente**: puntero al próximo nodo.
- **Previo / Anterior**: puntero al nodo predecesor.

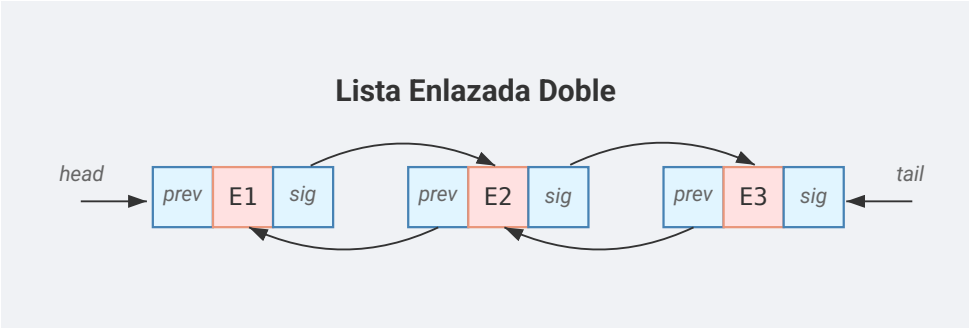


Figura 5.4: Lista Enlazada Doble

Go

```go
type node[T any] struct {
    data T
    next *node[T]
    prev *node[T]
}
```

Go

```go
type List[T comparable] struct {
    head *node[T]
    tail *node[T]
    size int
}
```

Lista Enlazada Circular

Definición

Una lista enlazada circular es una estructura donde el último nodo se enlaza al primero, formando un ciclo. Puede implementarse con enlaces simples o dobles.

En una lista circular doble:

- El sucesor del último nodo es el primero.
- El predecesor del primero es el último.

- No hay marcador de fin: el recorrido puede continuar indefinidamente.

El nodo es el mismo que el de la lista doble (`node[T]` con `next` y `prev`).

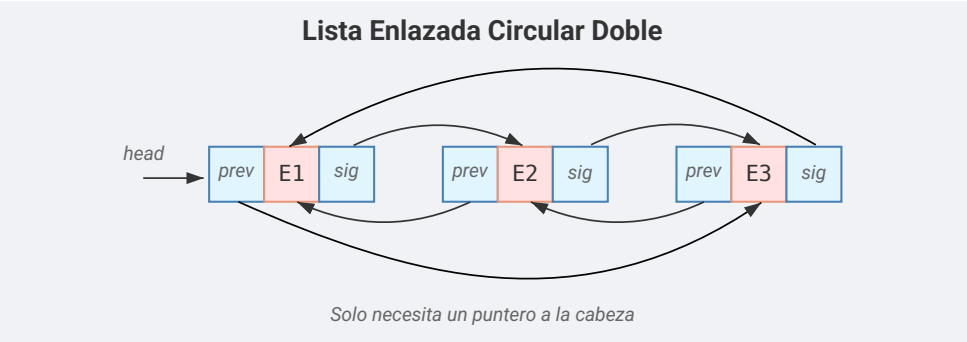


Figura 5.5: Lista Enlazada Circular Doble

Al ser cíclica, basta con mantener un único puntero a la cabeza: la cola se obtiene como `head.prev`. Esto ahorra un campo en la estructura.

Go

```go
type List[T comparable] struct {
    head *node[T]  // único puntero necesario
    size int
}
```

Interfaz común

Si bien las listas son versátiles y no existe un único comportamiento estándar para todas, vamos a definir una interfaz común para los tipos de lista que veremos a continuación. El objetivo es didáctico: acordar un conjunto de operaciones típicas que nos permita comparar implementaciones.

Go

```go
type List[T comparable] interface {
    // Consulta
    Size() int
    IsEmpty() bool
    Contains(data T) bool
    Head() (T, bool)
    Tail() (T, bool)

    // Inserción
    Prepend(data T)
    Append(data T)
    InsertAfter(target, data T) bool
    InsertBefore(target, data T) bool
```

```
    // Eliminación
    RemoveFirst() bool
    RemoveLast() bool
    Remove(data T) bool

    // Recorrido
    Values() []T

    // Utilidad
    Clear()
    String() string
}
```

La mayoría de las operaciones de inserción, eliminación y búsqueda dependen de un método interno `find` que recorre la lista en busca de un elemento. Este método es **privado** (en Go, minúscula inicial) porque devuelve un puntero a un nodo interno, y no debería exponerse fuera de la lista. A continuación se muestra su implementación en cada variante; las operaciones del resto de la sección lo referencian.

Simple

text

```
find(buscado):
    actual := head
    mientras actual != nil:
        si actual.dato == buscado:
            retornar actual
        actual = actual.siguiente
    retornar nil
```

find — Lista Simple

Doble

text

```
find(buscado):
    actual := head
    mientras actual != nil:
        si actual.dato == buscado:
            retornar actual
        actual = actual.siguiente
    retornar nil
```

find — Lista Doble

Circular

text

```
find(buscado):
    si IsEmpty():
        retornar nil
    actual := head
    para i := 0; i < Size(); i++:
        si actual.dato == buscado:
            retornar actual
        actual = actual.siguiente
    retornar nil
```

find — Lista Circular

5.3.4.1 Consulta

Size() Devuelve la cantidad de nodos de la lista.

Simple

text

```
Size():
    retornar size
```

Size — Lista Simple

Doble

text

```
Size():
    retornar size
```

Size — Lista Doble

Circular

text

```
Size():
    retornar size
```

Size — Lista Circular

IsEmpty() Devuelve `true` si la lista no tiene elementos.

Simple

```text
IsEmpty():
    retornar size == 0
```

IsEmpty — Lista Simple

Doble

```text
IsEmpty():
    retornar size == 0
```

IsEmpty — Lista Doble

Circular

```text
IsEmpty():
    retornar size == 0
```

IsEmpty — Lista Circular

Contains(data T) Devuelve `true` si el elemento está presente (solo la primera ocurrencia).

Simple

```text
Contains(buscado):
    retornar find(buscado) != nil
```

Contains — Lista Simple

Doble

```text
Contains(buscado):
    retornar find(buscado) != nil
```

Contains — Lista Doble

Circular

```text
Contains(buscado):
    retornar find(buscado) != nil
```

Contains — Lista Circular

Head() Devuelve el dato del primer nodo. Si la lista está vacía devuelve el valor cero y `false`.

Simple

```text
Head():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay primer dato
    retornar head.dato, verdadero
```

Head — Lista Simple

Doble

```text
Head():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay primer dato
    retornar head.dato, verdadero
```

Head — Lista Doble

Circular

```text
Head():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay primer dato
    retornar head.dato, verdadero
```

Head — Lista Circular

Tail() Devuelve el dato del último nodo. En la lista circular se obtiene desde `head.prev`.

Simple

text

```
Tail():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay último dato
    retornar tail.dato, verdadero
```

Tail — Lista Simple

Doble

text

```
Tail():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay último dato
    retornar tail.dato, verdadero
```

Tail — Lista Doble

Circular

text

```
Tail():
    si IsEmpty():
        retornar zero, falso  // lista vacía: no hay último dato
    retornar head.prev.dato, verdadero  // en la circular la cola precede a head
```

Tail — Lista Circular

5.3.4.2 Inserción

Prepend(data T) Agrega un nodo con el dato al inicio de la lista.

Simple

text

```
Prepend(dato):
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = head  // apunta al que era el primer nodo
    head = nuevo            // el nuevo pasa a ser el primer nodo
    si IsEmpty():
        tail = nuevo  // si estaba vacía, el nuevo también es la cola
    tamaño++
```

Prepend — Lista Simple

Doble

```text
Prepend(dato):
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = head  // apunta al que era el primer nodo
    si IsEmpty():
        tail = nuevo  // si estaba vacía, el nuevo también es la cola
    sino:
        head.prev = nuevo  // el nodo anterior enlaza su prev al nuevo
    head = nuevo  // el nuevo pasa a ser el primer nodo
    tamaño++
```

Prepend — Lista Doble

Circular

```text
Prepend(dato):
    nuevo := nuevoNodo(dato)
    si IsEmpty():
        nuevo.siguiente = nuevo  // lista de un solo nodo: se apunta a sí mismo
        nuevo.prev = nuevo
    sino:
        cola := head.prev
        nuevo.siguiente = head
        nuevo.prev = cola
        head.prev = nuevo
        cola.siguiente = nuevo  // cierra el ciclo entre cola y el nuevo head
    head = nuevo
    tamaño++
```

Prepend — Lista Circular

La lista simple solo actualiza `head` y `tail` en caso de lista vacía. La doble además debe enlazar `prev` del `head` anterior. La circular requiere mantener el ciclo, enlazando el nuevo nodo con `head` y con la cola (`head.prev`).

Append(data T) Agrega un nodo con el dato al final de la lista.

Simple

```text
Append(dato):
    nuevo := nuevoNodo(dato)
    si IsEmpty():
```

```
            head = nuevo  // el primer nodo de la lista
        sino:
            tail.siguiente = nuevo  // se encadena tras la cola actual
        tail = nuevo  // en ambos casos el nuevo pasa a ser la cola
        tamaño++
```

Append — Lista Simple

Doble

```text
  Append(dato):
      nuevo := nuevoNodo(dato)
      nuevo.prev = tail  // apunta al que era la cola
      si IsEmpty():
          head = nuevo  // el primer nodo de la lista
      sino:
          tail.siguiente = nuevo  // se encadena tras la cola actual
      tail = nuevo  // en ambos casos el nuevo pasa a ser la cola
      tamaño++
```

Append — Lista Doble

Circular

```text
Append(dato):
    si IsEmpty():
        Prepend(dato)  // reutiliza el caso de lista con un solo nodo
        retornar
    cola := head.prev
    nuevo := nuevoNodo(dato)
    nuevo.prev = cola  // el nuevo va entre la cola y head
    nuevo.siguiente = head
    cola.siguiente = nuevo
    head.prev = nuevo  // queda enlazado en ambos extremos del ciclo
    tamaño++
```

Append — Lista Circular

InsertAfter(target, data T) Busca `target` e inserta un nodo con `data` a continuación. Devuelve `false` si no encuentra `target`.

Simple

```text
```

```
InsertAfter(buscado, dato):
    actual := find(buscado)
    si actual == nil:
        retornar falso  // target no encontrado
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = actual.siguiente  // el nuevo apunta al sucesor de actual
    actual.siguiente = nuevo  // actual pasa a apuntar al nuevo
    si actual == tail:
        tail = nuevo  // si inserté al final, el nuevo es la nueva cola
    tamaño++
    retornar verdadero
```

InsertAfter — Lista Simple

Doble

text

```
InsertAfter(buscado, dato):
    actual := find(buscado)
    si actual == nil:
        retornar falso  // target no encontrado
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = actual.siguiente
    nuevo.prev = actual
    si actual == tail:
        tail = nuevo  // si inserté al final, el nuevo es la nueva cola
    sino:
        nuevo.siguiente.prev = nuevo  // el sucesor enlaza su prev al nuevo
    actual.siguiente = nuevo  // actual pasa a apuntar al nuevo
    tamaño++
    retornar verdadero
```

InsertAfter — Lista Doble

Circular

text

```
InsertAfter(buscado, dato):
    actual := find(buscado)
    si actual == nil:
        retornar falso  // target no encontrado
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = actual.siguiente
    nuevo.prev = actual
    actual.siguiente.prev = nuevo  // el sucesor enlaza su prev al nuevo
    actual.siguiente = nuevo  // actual pasa a apuntar al nuevo
    tamaño++
    retornar verdadero
```

InsertAfter — Lista Circular

InsertBefore(target, data T) Busca `target` e inserta un nodo con `data` antes. Devuelve `false` si no encuentra `target`.

Simple

text

```
InsertBefore(buscado, dato):
    si IsEmpty():
        retornar falso  // no hay nada ante qué insertar
    si head.dato == buscado:
        Prepend(dato)  // insertar antes del primer nodo es un prepend
        retornar verdadero
    actual := head
    mientras actual.siguiente != nil:
        si actual.siguiente.dato == buscado:
            nuevo := nuevoNodo(dato)
            nuevo.siguiente = actual.siguiente  // el nuevo apunta al target
            actual.siguiente = nuevo  // actual pasa a apuntar al nuevo
            tamaño++
            retornar verdadero
        actual = actual.siguiente  // recorre buscando al predecesor del target
    retornar falso
```

InsertBefore — Lista Simple

Doble

text

```
  InsertBefore(buscado, dato):
      actual := find(buscado)
      si actual == nil:
          retornar falso  // target no encontrado
      nuevo := nuevoNodo(dato)
      nuevo.prev = actual.prev
      nuevo.siguiente = actual
      si actual == head:
          head = nuevo  // si inserté antes del primer nodo, el nuevo es head
      sino:
          // el predecesor enlaza su siguiente al nuevo
          nuevo.prev.siguiente = nuevo
      actual.prev = nuevo  // target pasa a apuntar al nuevo
      tamaño++
      retornar verdadero
```

InsertBefore — Lista Doble

Circular

text

```
InsertBefore(buscado, dato):
    actual := find(buscado)
    si actual == nil:
        retornar falso  // target no encontrado
    nuevo := nuevoNodo(dato)
    nuevo.prev = actual.prev
    nuevo.siguiente = actual
    actual.prev.siguiente = nuevo  // el predecesor enlaza su siguiente al nuevo
    actual.prev = nuevo  // target pasa a apuntar al nuevo
    si actual == head:
        head = nuevo  // si inserté antes del primer nodo, el nuevo es head
    tamaño++
    retornar verdadero
```

InsertBefore — Lista Circular

En la lista simple `InsertBefore` requiere recorrer la lista buscando al predecesor porque no hay puntero `prev`. En la lista doble y circular, una vez encontrado el nodo, el reenlace es $O(1)$ gracias al puntero `prev`.

5.3.4.3 Eliminación

RemoveFirst() Elimina el primer nodo. Devuelve `false` si la lista está vacía.

Simple

text

```
RemoveFirst():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
        Clear()  // el único nodo: se vacía toda la lista
        retornar verdadero
    head = head.siguiente  // salta el primer nodo
    tamaño--
    retornar verdadero
```

RemoveFirst — Lista Simple

Doble

text

```
RemoveFirst():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
```

```
            Clear()  // el único nodo: se vacía toda la lista
            retornar verdadero
        head = head.siguiente  // salta el primer nodo
        head.prev = nil  // el nuevo head ya no tiene predecesor
        tamaño--
        retornar verdadero
```

RemoveFirst — Lista Doble

Circular

text

```
RemoveFirst():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
        Clear()  // el único nodo: se vacía toda la lista
        retornar verdadero
    cola := head.prev  // guarda la cola antes de mover head
    head = head.siguiente  // salta el primer nodo
    head.prev = cola
    cola.siguiente = head  // vuelve a cerrar el ciclo
    tamaño--
    retornar verdadero
```

RemoveFirst — Lista Circular

RemoveLast() Elimina el último nodo. Devuelve `false` si la lista está vacía.

Simple

text

```
RemoveLast():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
        Clear()  // el único nodo: se vacía toda la lista
        retornar verdadero
    actual := head
    mientras actual.siguiente != tail:
        actual = actual.siguiente  // avanza hasta el anteúltimo nodo
    actual.siguiente = nil  // desconecta el último nodo
    tail = actual  // el anteúltimo pasa a ser la cola
    tamaño--
    retornar verdadero
```

RemoveLast — Lista Simple

Doble

text

```
RemoveLast():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
        Clear()  // el único nodo: se vacía toda la lista
        retornar verdadero
    tail = tail.prev  // retrocede al anteúltimo nodo
    tail.siguiente = nil  // desconecta el último nodo
    tamaño--
    retornar verdadero
```

RemoveLast — Lista Doble

Circular

text

```
RemoveLast():
    si IsEmpty():
        retornar falso  // nada que eliminar
    si Size() == 1:
        Clear()  // el único nodo: se vacía toda la lista
        retornar verdadero
    cola := head.prev  // cola actual
    anteultimo := cola.prev
    anteultimo.siguiente = head  // la nueva cola apunta a head
    head.prev = anteultimo  // head queda enlazado con la nueva cola
    tamaño--
    retornar verdadero
```

RemoveLast — Lista Circular

`RemoveLast` es $O(n)$ en la lista simple porque debe recorrer hasta el anteúltimo nodo, mientras que en doble y circular es $O(1)$ gracias al puntero `prev`.

Remove(data T) Busca y elimina la **primera** ocurrencia del elemento. Si hay elementos duplicados, solo se elimina el primero que se encuentra. Devuelve `false` si no lo encuentra.

Simple

text

```
Remove(dato):
    si IsEmpty():
        retornar falso  // nada que eliminar
    si head.dato == dato:
        RemoveFirst()  // el target es la cabeza: reutiliza el caso simple
        retornar verdadero
```

```
        actual := head
        mientras actual.siguiente != nil:
            si actual.siguiente.dato == dato:
                // salta el nodo a eliminar
                actual.siguiente = actual.siguiente.siguiente
                si actual.siguiente == nil:
                    // si eliminé el último, actual pasa a ser la cola
                    tail = actual
                tamaño--
                retornar verdadero
            actual = actual.siguiente
        retornar falso
```

Remove — Lista Simple

Doble

text

```
           Remove(dato):
               actual := find(dato)
               si actual == nil:
                   retornar falso  // dato no encontrado
               si actual == head:
                   head = actual.siguiente  // el target era la cabeza
               sino:
                   // salta hacia adelante desde el predecesor
                   actual.prev.siguiente = actual.siguiente
               si actual == tail:
                   tail = actual.prev  // el target era la cola
               sino:
                   // salta hacia atrás desde el sucesor
                   actual.siguiente.prev = actual.prev
               tamaño--
               retornar verdadero
```

Remove — Lista Doble

Circular

text

```
    Remove(dato):
        si IsEmpty():
            retornar falso  // nada que eliminar
        si Size() == 1:
            si head.dato == dato:
                Clear()  // era el único nodo
                retornar verdadero
            retornar falso
```

```
        actual := head
        para i := 0; i < Size(); i++:
            si actual.dato == dato:
                // salta el nodo en el ciclo
                actual.prev.siguiente = actual.siguiente
                actual.siguiente.prev = actual.prev
                si actual == head:
                    head = actual.siguiente  // si eliminé la cabeza, avanza
                tamaño--
                retornar verdadero
            actual = actual.siguiente
        retornar falso
```

Remove — Lista Circular

La lista simple debe tratar como caso especial la eliminación de la cabeza (no hay predecesor) y la actualización de `tail`. En doble y circular, el puntero `prev` permite reenlazar simétricamente, aunque en doble aún se verifican extremos.

5.3.4.4 Recorrido

Values() Devuelve un slice con los datos en el orden de la lista.

Simple

text

```
        Values():
            resultado := []
            actual := head
            mientras actual != nil:
                resultado.agregar(actual.dato)
                actual = actual.siguiente  // avanza al siguiente nodo
            retornar resultado
```

Values — Lista Simple

Doble

text

```
        Values():
            resultado := []
            actual := head
            mientras actual != nil:
                resultado.agregar(actual.dato)
                actual = actual.siguiente  // avanza al siguiente nodo
            retornar resultado
```

Values — Lista Doble

Circular

```text
Values():
    si IsEmpty():
        retornar []  // sin nodos, no hay nada que recorrer
    resultado := []
    actual := head
    // nunca llega a nil: recorre una vuelta completa
    para i := 0; i < Size(); i++:
        resultado.agregar(actual.dato)
        actual = actual.siguiente
    retornar resultado
```

Values — Lista Circular

5.3.4.5 Utilidad

Clear() Elimina todos los nodos y deja la lista vacía.

Simple

```text
Clear():
    head = nil
    tail = nil
    tamaño = 0
```

Clear — Lista Simple

Doble

```text
Clear():
    head = nil
    tail = nil
    tamaño = 0
```

Clear — Lista Doble

Circular

```text
```

```
Clear():
    head = nil  // en la circular solo existe head: no hay tail que resetear
    tamaño = 0
```

Clear — Lista Circular

String() Devuelve una representación textual de la lista.

Simple

```text
String():
    elementos := Values()
    retornar "[" + elementos.join(", ") + "]"
```

String — Lista Simple

Doble

```text
String():
    elementos := Values()
    retornar "[" + elementos.join(", ") + "]"
```

String — Lista Doble

Circular

```text
String():
    elementos := Values()
    retornar "[" + elementos.join(", ") + "]"
```

String — Lista Circular

Lista con Centinelas

Definición

Los centinelas son nodos ficticios que se colocan al principio y al final de la lista. No contienen datos útiles y su propósito es eliminar los casos especiales en las operaciones de inserción y eliminación.

En una lista con centinelas:

- La cabeza real es el sucesor del centinela frontal.
- La cola real es el predecesor del centinela trasero.
- Una lista vacía tiene los dos centinelas apuntándose entre sí.

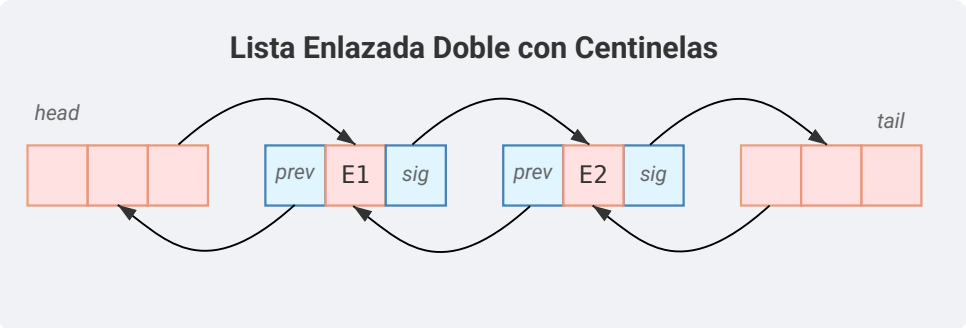


Figura 5.54: Lista Enlazada Doble con Centinelas

Go

```go
type List[T comparable] struct {
    head *node[T]  // centinela frontal
    tail *node[T]  // centinela trasero
    size int
}
```

Los centinelas eliminan los casos borde de lista vacía, cabeza y cola. Al inicializar la lista, `head.siguiente = tail` y `tail.prev = head`. Como los centinelas nunca son `nil`, toda inserción o eliminación ocurre siempre entre dos nodos reales o ficticios, y las operaciones se vuelven uniformes.

find Al no haber `nil`, el recorrido va desde `head.siguiente` hasta llegar al centinela `tail`.

text

```
find(buscado):
    // el primer nodo real está tras el centinela frontal
    actual := head.siguiente
    // el centinela trasero marca el fin: nunca se llega a nil
    mientras actual != tail:
        si actual.dato == buscado:
            retornar actual
        actual = actual.siguiente
    retornar nil
```

find — Lista con Centinelas

IsEmpty La lista vacía se detecta cuando `size == 0` o los centinelas se apuntan entre sí.

text

```
IsEmpty():
    // vacía cuando los centinelas se apuntan entre sí
    retornar head.siguiente == tail
```

IsEmpty — Lista con Centinelas

Head / Tail El primer dato está en `head.siguiente` y el último en `tail.prev`.

text

```
Head():
    si IsEmpty():
        retornar zero, falso  // lista vacía
    // el primer nodo real está tras el centinela frontal
    retornar head.siguiente.dato, verdadero

Tail():
    si IsEmpty():
        retornar zero, falso  // lista vacía
    // el último nodo real está antes del centinela trasero
    retornar tail.prev.dato, verdadero
```

Head / Tail — Lista con Centinelas

InsertBefore No necesita verificar si `actual` es la cabeza real porque el centinela `head` garantiza que `actual.prev` nunca es `nil`.

text

```
InsertBefore(buscado, dato):
    actual := find(buscado)
    si actual == nil:
        retornar falso  // buscado no encontrado
    nuevo := nuevoNodo(dato)
    // si actual es el primer nodo real, su prev es el centinela head
    nuevo.prev = actual.prev
    nuevo.siguiente = actual
    actual.prev.siguiente = nuevo  // el predecesor enlaza su siguiente al nuevo
    actual.prev = nuevo  // actual pasa a apuntar al nuevo
    tamaño++
    retornar verdadero
```

InsertBefore — Lista con Centinelas

InsertAfter Como `actual.siguiente` nunca es `nil` (puede ser `tail`), no hay caso especial de cola.

text

```
InsertAfter(buscado, dato):
    actual := find(buscado)
    si actual == nil:
```

```
            retornar falso  // buscado no encontrado
        nuevo := nuevoNodo(dato)
        // el sucesor puede ser el centinela tail: nunca es nil
        nuevo.siguiente = actual.siguiente
        nuevo.prev = actual
        actual.siguiente.prev = nuevo  // el sucesor enlaza su prev al nuevo
        actual.siguiente = nuevo  // actual pasa a apuntar al nuevo
        tamaño++
        retornar verdadero
```

InsertAfter — Lista con Centinelas

Prepend Inserta entre el centinela `head` y el primer nodo real. No puede delegar en `InsertBefore` porque busca por valor y fallaría con datos duplicados.

text

```
Prepend(dato):
    nuevo := nuevoNodo(dato)
    // si la lista está vacía, head.siguiente es tail: igual funciona
    nuevo.siguiente = head.siguiente
    nuevo.prev = head
    // el primer nodo real (o el centinela tail) enlaza su prev al nuevo
    head.siguiente.prev = nuevo
    head.siguiente = nuevo  // el centinela head apunta al nuevo
    tamaño++
```

Prepend — Lista con Centinelas

Append Inserta entre el último nodo real y el centinela `tail`.

text

```
Append(dato):
    nuevo := nuevoNodo(dato)
    nuevo.siguiente = tail  // apunta al centinela final
    // si la lista está vacía, tail.prev es head: igual funciona
    nuevo.prev = tail.prev
    // el último nodo real (o el centinela head) enlaza su siguiente al nuevo
    tail.prev.siguiente = nuevo
    tail.prev = nuevo  // el centinela tail apunta al nuevo
    tamaño++
```

Append — Lista con Centinelas

Remove No necesita verificar si el nodo es `head` o `tail` real. Los centinelas aseguran que `actual.prev` y `actual.siguiente` siempre existen.

text

```
Remove(dato):
    actual := find(dato)
```

```
    si actual == nil:
        retornar falso  // dato no encontrado
    // sin casos de cabeza o cola: si actual es el primer o último nodo real,
    // el vecino del otro lado es un centinela y se reenlaza igual
    actual.prev.siguiente = actual.siguiente  // el predecesor salta al sucesor
    actual.siguiente.prev = actual.prev  // el sucesor salta al predecesor
    tamaño--
    retornar verdadero
```

Remove — Lista con Centinelas

RemoveFirst Reenlaza el centinela `head` con el segundo nodo real. No puede delegar en `Remove` por el mismo problema de los duplicados.

text

```
RemoveFirst():
    si IsEmpty():
        retornar falso  // nada que eliminar
    head.siguiente = head.siguiente.siguiente  // salta el primer nodo real
    // el nuevo head.siguiente apunta al centinela
    // (puede ser tail si era el único nodo)
    head.siguiente.prev = head
    tamaño--
    retornar verdadero
```

RemoveFirst — Lista con Centinelas

RemoveLast Reenlaza el centinela `tail` con el anteúltimo nodo real.

text

```
    RemoveLast():
        si IsEmpty():
            retornar falso  // nada que eliminar
        tail.prev = tail.prev.prev  // retrocede al anteúltimo nodo real
        // el nuevo tail.prev apunta al centinela
        // (puede ser head si era el único nodo)
        tail.prev.siguiente = tail
        tamaño--
        retornar verdadero
```

RemoveLast — Lista con Centinelas

Clear Solo reenlaza los centinelas entre sí.

text

```
        Clear():
            head.siguiente = tail  // reenlaza los centinelas entre sí
            tail.prev = head
            tamaño = 0
```

Clear — Lista con Centinelas

Ventajas frente a las versiones sin centinelas:

- `InsertBefore` e `InsertAfter` sin verificar extremos.
- `Prepend` y `Append` con código simétrico, sin casos de lista vacía.
- `Remove` sin verificar `head`/`tail`.
- `RemoveFirst` y `RemoveLast` con código simétrico.
- `Clear` solo reenlaza los centinelas.
- `IsEmpty` es simplemente comparar punteros.

La principal desventaja es el costo de memoria de dos nodos adicionales, que es despreciable en la mayoría de los escenarios.

Operación	Simple	Doble	Circular	Centinelas
`Head`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Tail`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Prepend`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Append`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`InsertAfter`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`InsertBefore`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`RemoveFirst`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`RemoveLast`	$O(n)$	$O(1)$	$O(1)$	$O(1)$
`Remove`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`find`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Clear`	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 5.66: Comparación de algunas de las operaciones según la variante

La complejidad asintótica de las operaciones que requieren búsqueda es $O(n)$ en todas las variantes. La diferencia está en las constantes y en la cantidad de casos especiales que debe manejar el código: la lista simple requiere verificaciones constantes de `nil` en los extremos, mientras que los centinelas las eliminan por completo.

Ejercicios

Los ejercicios de este capítulo están en `03-listas/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.4 Mapas de Bits

Los mapas de bits son estructuras de datos que permiten registrar la presencia o ausencia de un conjunto de elementos. Cada posición del mapa de bits representa un elemento, y el valor de cada posición indica si el elemento está presente (1) o ausente (0).

Una de las implementaciones más eficientes de mapas de bits es el uso de números enteros sin signo, aprovechando su representación binaria. Cada bit de un número entero puede representar la presencia o ausencia de un elemento. Por ejemplo, si tenemos un conjunto de elementos que van del 0 al 7, podemos usar un `uint8` de 8 bits para representar la presencia o ausencia de cada elemento. Esta implementación es eficiente en términos de espacio y tiempo, ya que permite realizar operaciones de búsqueda, inserción y eliminación en tiempo constante $O(1)$.

Los mapas de bits se utilizan en diversas aplicaciones, como la representación de conjuntos, la compresión de datos y la implementación de algoritmos de búsqueda. Son especialmente útiles en situaciones donde se requiere un acceso rápido a los elementos y se tiene un conjunto limitado de posibles valores.

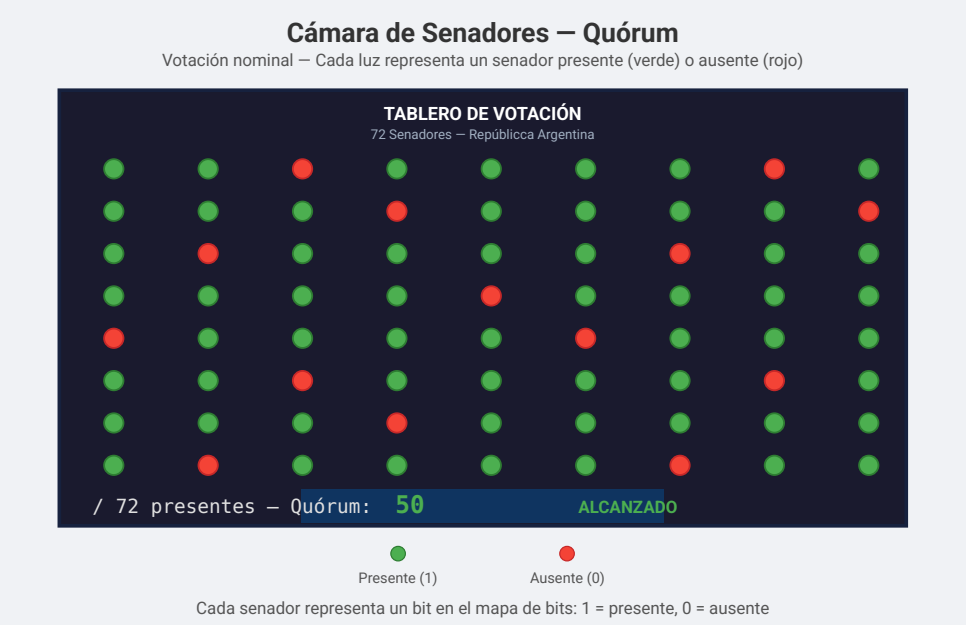


Figura 5.67: Quórum en una sesión del Congreso.

Implementación con números enteros

Supongamos que tenemos un conjunto de elementos que van del 0 al 7. Podemos usar un `uint8` de 8 bits para representar la presencia o ausencia de cada elemento. Por ejemplo, si el número entero es 15, cuya representación en binario con 8 bits es `00001111`, esto indica que los elementos 0, 1, 2 y 3 están presentes, mientras que los demás elementos están ausentes.

El bit menos significativo (a la derecha) representa el elemento 0, el siguiente bit representa el elemento 1, y así sucesivamente.

5.4.1.1 Operaciones sobre bits

Los mapas de bits permiten realizar diversas operaciones sobre los bits, como la inserción, eliminación y búsqueda de elementos. Estas operaciones se pueden implementar utilizando operadores binarios que implementan las operaciones lógicas booleanas.

Las operaciones lógicas booleanas son:

***AND* (Producto Lógico)** Operación que devuelve 1 si ambos bits son 1, y 0 en caso contrario. El operador *AND* se representa con el símbolo `&`.

***OR* (Suma Lógica)** Operación que devuelve 1 si al menos uno de los bits es 1, y 0 en caso contrario. El operador *OR* se representa con el símbolo `|`.

***XOR* (OR Exclusivo)** Operación que devuelve 1 si los bits son diferentes (uno es 1 y el otro es 0), y 0 en caso contrario. El operador *XOR* se representa con el símbolo `^`.

***NOT* (Negación)** Operación que invierte el valor de un bit (0 se convierte en 1 y 1 se convierte en 0). El operador *NOT* se representa con el símbolo `^`.

A continuación se presenta la tabla de verdad de las operaciones lógicas booleanas sobre bits:

A	B	(A & B)	(A | B)	(A ^ B)	(^A)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 5.68: Tabla de verdad de las operaciones sobre bits

5.4.1.2 Operadores binarios en Go

Go soporta 6 operaciones binarias y una operación unaria que se pueden utilizar para manipular los bits de un número entero. Estas operaciones son:

AND El operador *AND* (`&`) compara dos números bit a bit y devuelve un nuevo número donde cada bit es 1 si ambos bits de entrada son 1, y 0 en caso contrario. Por ejemplo:

Go

```go
var a uint8 = 0b00001100 // 12
var b uint8 = 0b00001010 // 10
c := a & b               // 0b00001000 (8)
```

Esto significa que el resultado `c` tiene un 1 en la posición 3 (de derecha a izquierda) porque tanto `a` como `b` tienen un 1 en esa posición.

OR El operador *OR* (`|`) compara dos números bit a bit y devuelve un nuevo número donde cada bit es 1 si al menos uno de los bits de entrada es 1. Por ejemplo:

Go

```
var a uint8 = 0b00001100 // 12
var b uint8 = 0b00001010 // 10
c := a | b               // 0b00001110 (14)
```

Esto significa que el resultado `c` tiene un 1 en las posiciones 1, 2 y 3 porque al menos uno de los bits de entrada es 1 en esas posiciones.

XOR El operador *OR Exclusivo*, *XOR* (^) compara dos números bit a bit y devuelve un nuevo número donde cada bit es 1 si los bits de entrada son diferentes (uno es 1 y el otro es 0). Por ejemplo:

Go

```
var a uint8 = 0b00001100 // 12
var b uint8 = 0b00001010 // 10
c := a ^ b               // 0b00000110 (6)
```

Esto significa que el resultado `c` tiene un 1 en las posiciones 1 y 2 porque los bits de entrada son diferentes en esas posiciones.

Desplazamiento a Izquierda o *Left Shift* El operador de desplazamiento a la izquierda (<<) desplaza todos los bits de un número hacia la izquierda, llenando los bits vacíos con ceros. Por ejemplo:

Go

```
var a uint8 = 0b00000001 // 1
b := a << 2              // 0b00000100 (4)
```

Esto significa que el resultado `b` tiene un 1 en la posición 2 porque se ha desplazado el bit 1, originalmente en la posición 0, dos posiciones a la izquierda.

Desplazamiento a Derecha o *Right Shift* El operador de desplazamiento a la derecha (>>) desplaza todos los bits de un número hacia la derecha, llenando los bits vacíos con ceros. Por ejemplo:

Go

```
var a uint8 = 0b00001000 // 8
b := a >> 2              // 0b00000010 (2)
```

Esto significa que el resultado `b` tiene un 1 en la posición 1 porque se ha desplazado el bit 1, originalmente en la posición 3, dos posiciones a la derecha.

NOT La única operación unaria es la negación o complemento (los 0 se cambian por 1 y viceversa). Esta operación se representa con el símbolo ^. Por ejemplo:

Go

```go
var a uint8 = 0b00001100 // 12
b := ^a                  // ^a invierte los 8 bits: 0b11110011 (243)
```

El NOT invierte **todos** los bits del `uint8`. El resultado es `0b11110011`, que en decimal es `243`. Como `uint8` no tiene signo, el bit más significativo se interpreta como parte del valor, no como signo.

Es por esto que los tipos sin signo (`uint8`, `uint16`, `uint32`, `uint64`) son los más convenientes para implementar mapas de bits: todas las operaciones bit a bit se comportan de forma predecible sobre el rango completo de bits, sin interferencias por la interpretación de signo que introduciría un `int`.

AND NOT* o *bit clear El operador *bit clear* (`&^`) es una combinación de los operadores *AND* y *NOT*. Este operador se utiliza para borrar bits específicos en un número. Es como realizar la operación AND entre el primer operando y el complemento del segundo. Por ejemplo:

Go

```go
var a uint8 = 0b00001100 // 12
var b uint8 = 0b00001010 // 10
// ^b invierte los 8 bits (0b11110101), pero al hacer
// AND con a (0b00001100) solo importan los 4 bits bajos
c := a &^ b              // 0b00000100 (4)
```

El operador `&^` (`a &^ b`) equivale a `a & (^b)`. Podemos pensarlo como "apagar" en `a` los bits que están encendidos en `b`. El resultado `c` tiene un 1 en la posición 2 porque `a` tiene un 1 ahí y `b` tiene un 0; en cambio, los bits 1 y 3 de `a` se apagan porque `b` tiene 1 en esas posiciones.

Operación	Símbolo	Ejemplo
AND	`&`	`a & b`
OR	`|`	`a | b`
XOR	`^`	`a ^ b`
Desplazamiento a Izquierda	`<<`	`a << 2`
Desplazamiento a Derecha	`>>`	`a >> 2`
NOT	`^`	`^a`
AND NOT	`&^`	`a &^ b`

Table 5.69: Operaciones sobre bits en Go

Go también soporta los operadores de asignación compuesta, que combinan una operación binaria con una asignación. Estos operadores son:

Símbolo	Operación	Ejemplo
`&=`	`a = a & b`	`a &= b`
`|=`	`a = a | b`	`a |= b`
`^=`	`a = a ^ b`	`a ^= b`

`<<=`	`a = a << b`	`a <<= b`
`>>=`	`a = a >> b`	`a >>= b`
`&^=`	`a = a &^ b`	`a &^= b`

Table 5.70: Operadores de asignación compuesta en Go

Ejercicios

Los ejercicios de este capítulo están en `04-mapa-de-bits/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.5 Tablas de hash

Las tablas de *hash* son estructuras de datos eficientes para almacenar y recuperar información. Emplean una función *hash* para convertir claves en índices dentro de un arreglo, permitiendo un acceso a los datos en tiempo promedio constante ($O(1)$[5]). Esto las hace ideales para implementar diccionarios, conjuntos y otras estructuras que demandan búsquedas rápidas. Una tabla de *hash* extiende la idea de un arreglo tradicional, posibilitando el uso de claves de cualquier tipo para acceder a sus posiciones, gracias a la función *hash* que asigna cada clave a un índice específico en el arreglo. La eficacia de esta estructura depende de una función *hash* que sea rápida y distribuya las claves de manera uniforme en el arreglo. La siguiente imagen ilustra los componentes de una tabla de *hash*.

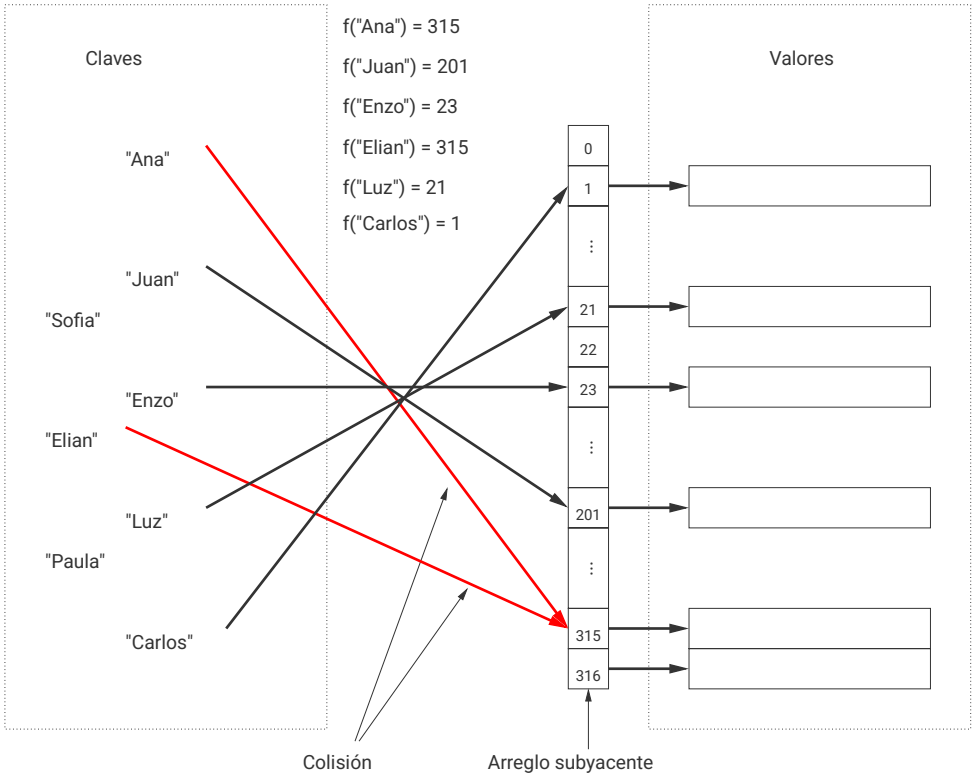


Figura 5.71: Esquema de una tabla de *hash*.

Claves El universo de claves puede ser muy grande, o infinito. De ese conjunto solo un subconjunto se almacenará en la tabla de *hash*.

Valores El valor asociado a una clave. Puede ser cualquier tipo de dato, incluyendo estructuras de datos complejas.

Función de *hash* Una función que toma una clave y devuelve un índice en el arreglo.

Arreglo La estructura de datos subyacente que almacena los elementos, es un arreglo de pares `(clave, valor)`. El tamaño del arreglo debe ser lo suficientemente grande para evitar colisiones, pero no tan grande como para desperdiciar espacio. Se debe almacenar

[5]En este caso se toma el tiempo promedio, bajo ciertas condiciones. Cuando se evalúa un TAD donde se insertan, buscan y remueven elementos muy frecuentemente, se acostumbra tomar el tiempo promedio de las operaciones. Sin embargo, en el peor de los casos, algunas operaciones pueden tener un orden lineal.

el par `(clave, valor)` para poder distinguir entre claves diferentes que colisionan en el mismo índice.

Colisiones Ocurren cuando dos claves diferentes se mapean al mismo índice en el arreglo. Las colisiones deben manejarse de manera eficiente para garantizar un rendimiento óptimo de la tabla de *hash*.

Claves

Las claves deben ser **inmutables**, es decir, no pueden modificarse una vez que se han creado. Esto es importante porque la función de *hash* se basa en el valor de la clave para calcular el índice en el arreglo. Si la clave cambia, el índice también cambiará, lo que puede provocar errores al intentar acceder a los elementos en la tabla de *hash*.

Las claves pueden ser de cualquier tipo de dato, incluyendo enteros, cadenas de texto, estructuras de datos complejas, etc. Es importante que la función de *hash* sea capaz de manejar todos los tipos de claves que se utilicen, otra opción es utilizar distintas funciones de *hash* para distintos tipos de claves.

Colisiones

Un desafío frecuente en las tablas de *hash* son las colisiones. Estas se producen cuando dos claves distintas son convertidas por la función *hash* al mismo índice dentro del arreglo. Esto puede deberse a una distribución no uniforme de las claves por la función *hash* o a una capacidad limitada del arreglo para contener todos los datos. Para resolver este problema, se han desarrollado diversas estrategias, entre las cuales se encuentran:

- *Hashing* abierto
- *Hashing* cerrado

hashing cerrado con búsqueda lineal

Esta técnica, también conocida como direccionamiento cerrado con sondeo o búsqueda lineal, aborda las colisiones buscando secuencialmente la siguiente ubicación disponible en el arreglo. Cuando una función *hash* asigna un índice ya ocupado a una nueva clave, en lugar de sobrescribir el elemento existente, se examinan las posiciones subsiguientes del arreglo de forma lineal (incrementando el índice de uno en uno). Este proceso continúa hasta que se encuentra una celda vacía donde se puede almacenar el nuevo elemento. Posteriormente, al buscar un elemento, se aplica la misma función *hash* a la clave buscada y si la posición inicial está ocupada por un elemento diferente, se sigue la misma secuencia lineal de búsqueda hasta encontrar el elemento deseado o una posición vacía (lo que indicaría que el elemento no está presente).

Si bien esta técnica es fácil de implementar, puede llevar a la formación de *clusters* o agrupaciones de elementos consecutivos ocupados, lo que puede degradar el rendimiento de las operaciones, ya que se requerirán más comparaciones para encontrar elementos ubicados en estos *clusters* o posiciones vacías para insertar nuevos elementos. Cuando estos *clusters*

crecen demasiado se degrada el rendimiento de la tabla. Para mitigar este problema se pueden utilizar diferentes técnicas como el sondeo cuadrático que en lugar de incrementar el índice de uno en uno, lo hace de acuerdo a una función cuadrática predeterminada o el doble *hashing* que utiliza una segunda función de *hash* para determinar el incremento del índice.

Esta técnica requiere poder distinguir entre posiciones vacías en la tabla y posiciones donde se han borrado elementos. Para ello se debe marcar las posiciones eliminadas con algún marcador especial. Esto permite que la búsqueda continúe sin problemas.

5.5.3.1 Inserción

En la siguiente figura se grafica la inserción de un elemento en una tabla de *hash* cerrada.

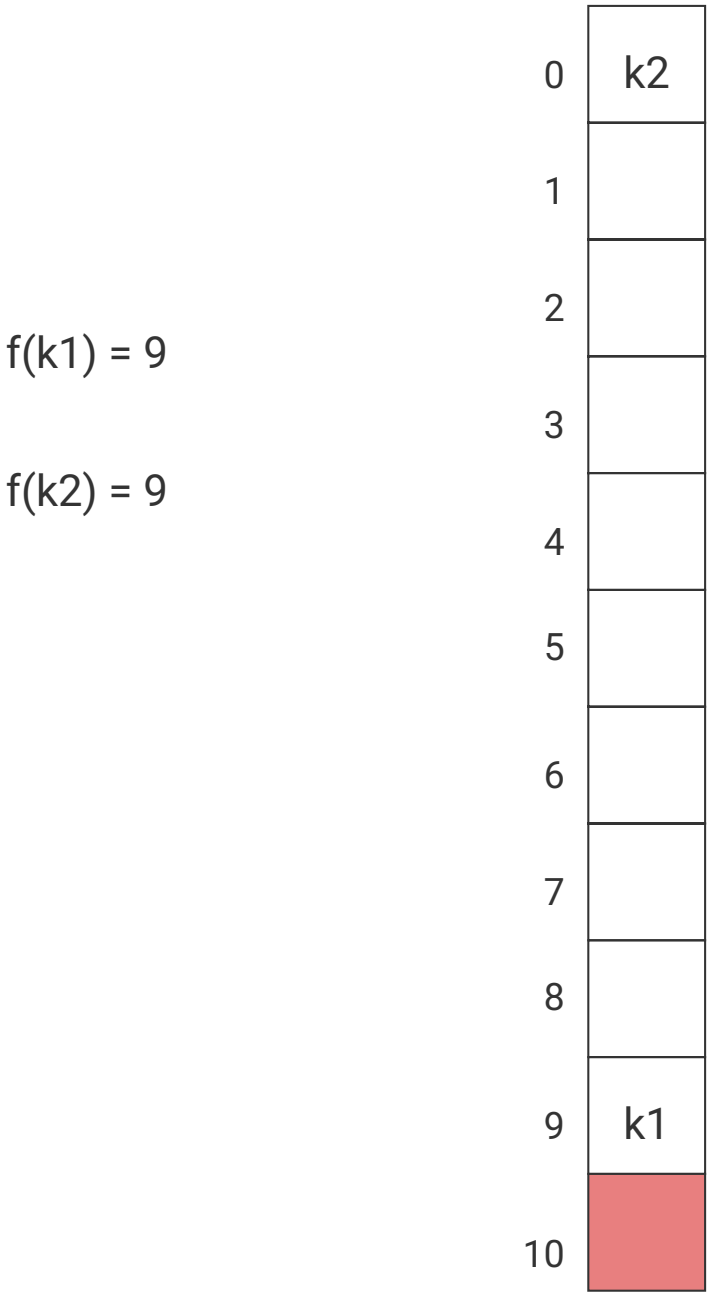


Figura 5.72: Inserción en una tabla de *hash* cerrada.

Supongamos que vamos a insertar las claves k_1 y k_2, en ese orden, y que la posición 10 del arreglo ya se encontraba ocupada. En este caso, $f(k_1)$ devuelve el índice 9, como inicialmente la posición se encuentra vacía, se puede asociar la clave a ese índice. Luego $f(k_2)$ también devuelve 9, como la posición 9 se encuentra ocupada, intenta en la próxima, como la posición 10 ya está ocupada incrementa en forma circular el índice y finalmente puede asociar k_2 a la posición 0 del arreglo.

5.5.3.2 Eliminación

La eliminación de un elemento en una tabla de *hash* cerrada es un poco más complicada que la inserción. Cuando se elimina un elemento, se debe marcar la posición como eliminada, pero no se puede dejar la posición vacía, ya que esto podría causar problemas al buscar otros elementos en el futuro. En su lugar, se debe utilizar un marcador especial para indicar que la posición ha sido eliminada.

Esto se conoce como *marcador de eliminación* y permite que la búsqueda continúe sin problemas. Sin embargo, esto puede aumentar el tiempo de búsqueda si hay muchos elementos eliminados en la tabla de *hash*.

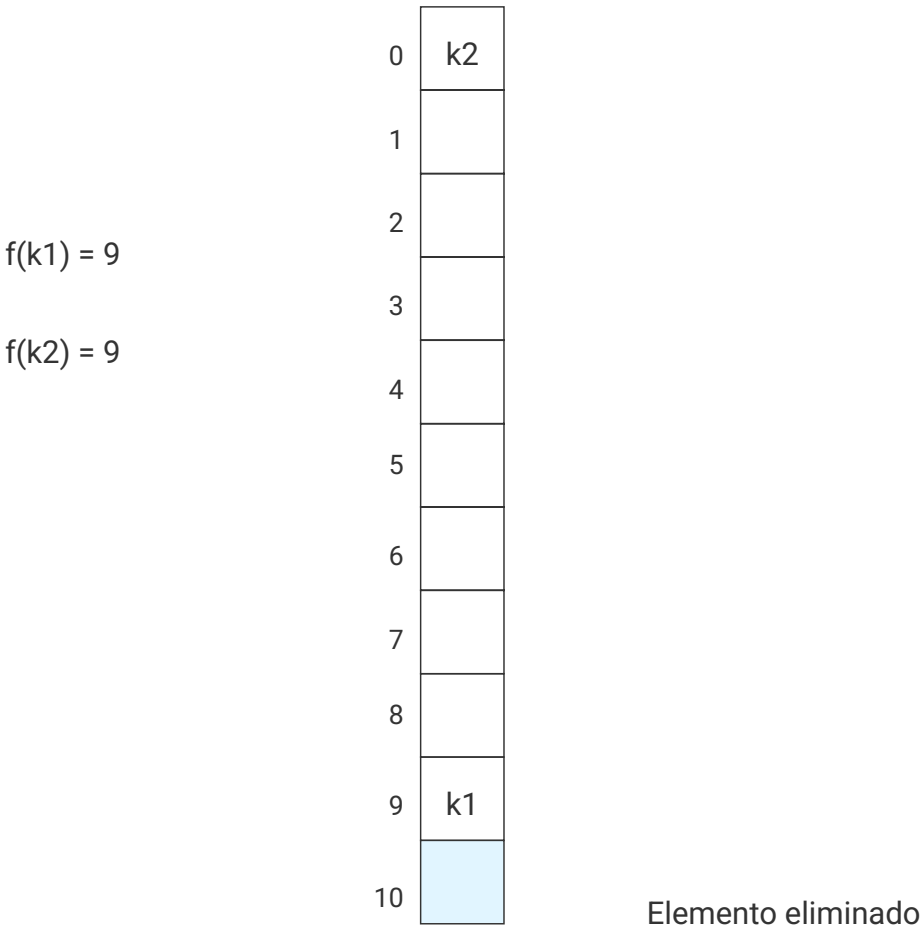


Figura 5.73: Eliminación en una tabla de *hash* cerrada.

En la figura se observa que al eliminar el elemento en la posición 10, se marca la posición como eliminada, pero no se deja vacía. Esto permite que la búsqueda de otros elementos continúe sin problemas. En resumen, una posición de la tabla podrá estar en algunos de los tres siguientes estados:

- Vacía
- Ocupada
- Eliminada

5.5.3.3 Búsqueda

La búsqueda de un elemento en una tabla de *hash* cerrada es similar a la inserción. Se utiliza la función de *hash* para calcular el índice y luego se busca el elemento en esa posición. Si el elemento no se encuentra en la posición calculada, se incrementa el índice en 1 hasta encontrar el elemento o una posición vacía. Si se encuentra una posición eliminada, la búsqueda continua.

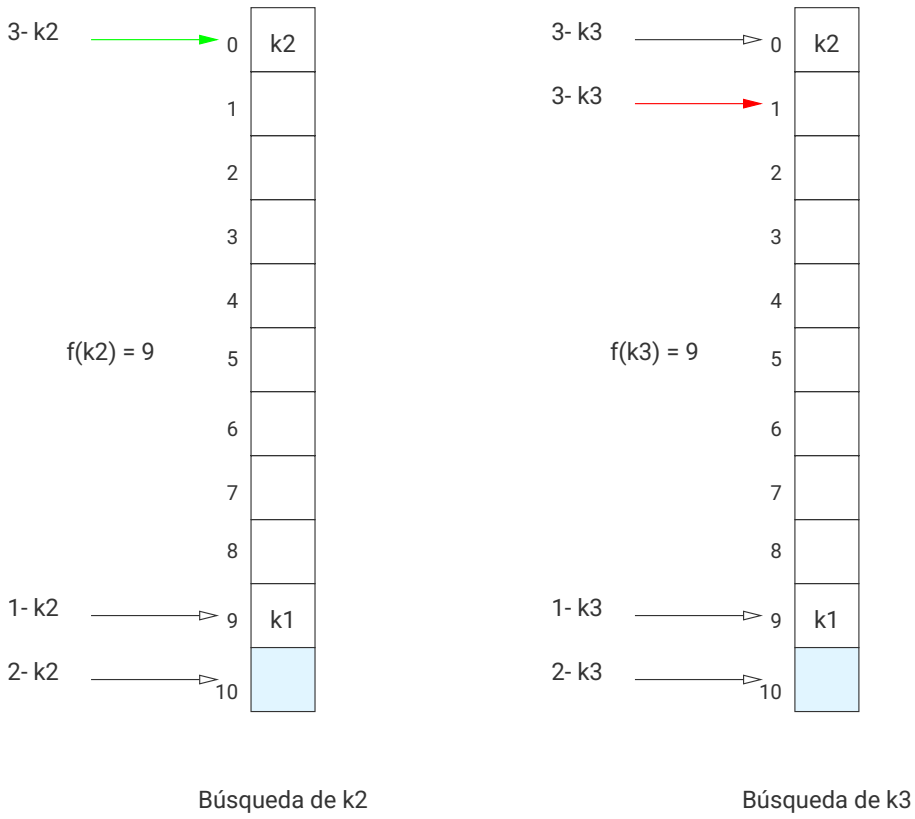


Figura 5.74: Búsqueda en una tabla de *hash* cerrada.

En la figura se observa que al buscar la clave k_2, se intenta con la posición 9, como la clave asociada a esa posición es k_1, la búsqueda continua, luego la posición 10 está eliminada, por lo que se avanza a la posición 0 donde finalmente se encuentra la clave buscada.

Supongamos que se busca una clave k_3 cuyo valor de *hash* también es 9. En este caso, la búsqueda comenzaría en la posición 9, pero como la clave asociada a esa posición es k_1, la búsqueda continuaría hasta encontrar la clave o una posición vacía. En este caso en la posición 1 se encuentra una posición vacía, por lo que se concluye que la clave k_3 no está presente en la tabla de *hash*.

5.5.3.4 Aritmética Modular

Es la aritmética que deriva del estudio del resto de la división de enteros. Es una forma de realizar cálculos en un conjunto finito de números. En el caso del *hashing* cerrado, se utiliza para calcular el índice en el arreglo. La operación más común en la aritmética modular es el operador módulo, que devuelve el resto de la división de dos números.

La operación de módulo se denota como $amodb$, donde a es el número que se va a dividir y b es el divisor. El resultado de la operación es el resto de la división de a entre b. Por ejemplo, $7mod3 = 1$, ya que al dividir 7 entre 3, el cociente es 2 y el resto es 1.

Algunas propiedades útiles de la aritmética modular son:

- $(a + b)modn = (amodn + bmodn)modn$
- $(a - b)modn = (amodn - bmodn)modn$
- $(a \cdot b)modn = (amodn \cdot bmodn)modn$
- $(a^b)modn = (amodn)^b modn$

Hashing abierto

El *hashing* abierto es una técnica fundamental para el manejo de colisiones en tablas de *hash*. En contraste con el direccionamiento cerrado, donde todos los elementos se almacenan directamente dentro del arreglo subyacente, el *hashing* abierto utiliza una estructura de datos adicional (generalmente una lista enlazada). En cada posición del arreglo se encuentra una lista enlazada para almacenar los elementos que colisionan en ese índice. Cuando dos o más claves diferentes se "mapean" a la misma posición en la tabla, en lugar de buscar otra posición en el arreglo, simplemente se añaden los nuevos elementos a la lista existente en esa posición. Esto significa que múltiples elementos pueden residir en la misma posición del arreglo, encadenados en una lista.

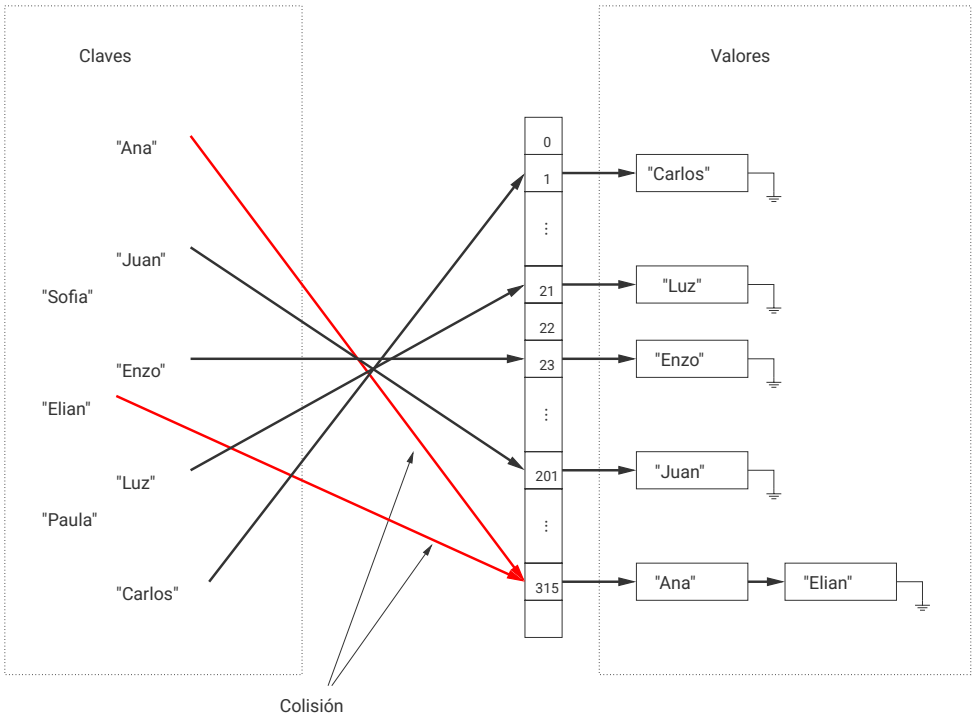


Figura 5.75: Tabla de *hash* abierta.

Consideraciones de diseño

Al diseñar una tabla de *hash* es muy importante evitar colisiones y al mismo tiempo ocupar el menor espacio posible. Para ello se deben considerar los siguientes aspectos:

5.5.5.1 Capacidad y tamaño

En una tabla de *hash* es importante distinguir dos conceptos relacionados pero distintos:

Capacidad (m): es la cantidad total de *buckets* del arreglo subyacente. Representa el espacio máximo disponible antes de redimensionar (aunque en la práctica se redimensiona antes por el factor de carga). En el código es el campo `capacity`.

Tamaño (n): es la cantidad de elementos efectivamente almacenados en la tabla. En el código es el campo `size`.

La relación entre ambos está dada por el factor de carga $\alpha = n/m$, que indica qué tan llena está la tabla.

5.5.5.2 Factor de carga

El factor de carga (α) es una métrica crucial para evaluar y predecir el rendimiento de una tabla de *hash*. Se define formalmente como la relación entre el número de elementos (n) actualmente almacenados en la tabla y el tamaño (m) del arreglo subyacente:

$$\alpha = \frac{n}{m} \tag{5.1}$$

Este valor proporciona una indicación de cuán "llena" está la tabla de *hash*. La intuición detrás del factor de carga es que a medida que la tabla se llena (es decir, a medida que α se acerca o supera a 1), la probabilidad de que nuevas claves colisionen con posiciones ya ocupadas aumenta significativamente.

Se recomienda mantener el factor de carga por debajo de 0.75 como una guía general basada en el compromiso entre el rendimiento y la utilización del espacio. Este umbral suele ser un buen punto de equilibrio para muchas aplicaciones que utilizan técnicas de *hashing* cerrado. Sin embargo, el factor de carga óptimo puede variar dependiendo de:

La técnica de resolución de colisiones utilizada El *hashing* abierto puede tolerar factores de carga más altos (incluso mayores que 1) antes de que el rendimiento se degrade significativamente en comparación con el direccionamiento cerrado. En el *hashing* abierto, un factor de carga α significa que en promedio, cada lista tendrá α elementos.

Los requisitos específicos de la aplicación Si la eficiencia de las operaciones es primordial, se podría optar por un factor de carga más bajo. Si el espacio es una limitación crítica, se podría tolerar un factor de carga más alto a expensas de un posible impacto en el rendimiento.

La calidad de la función hash Una función hash que distribuye las claves de manera más uniforme permitirá un rendimiento aceptable incluso con factores de carga ligeramente más altos, ya que habrá menos colisiones para el mismo número de elementos.

5.5.5.3 Tamaño del arreglo

Se recomienda que el tamaño del arreglo (m) en una tabla de *hash* sea un **número primo**. Esto se fundamenta en principios matemáticos relacionados con la distribución uniforme de las claves al aplicar la función *hash* y la aritmética modular. Si bien no es una regla absoluta y existen implementaciones exitosas con tamaños no primos, utilizar un tamaño de arreglo primo puede mitigar ciertos patrones de colisión y mejorar la dispersión de las claves, especialmente cuando se combinan con ciertas funciones *hash*.

Si la posición de una clave en el arreglo se calcula como:

$$indice = \text{hash}(clave) mod m \tag{5.2}$$

Si m es un número primo, la función hash tiende a distribuir las claves de manera más uniforme en el arreglo. Esto se debe a que los números primos no tienen divisores comunes con otros números, lo que reduce la probabilidad de que diferentes claves generen el mismo índice.

Otras tablas de *hash* que no utilizan la aritmética modular, no necesitan que el tamaño del arreglo sea primo. En estos casos, el tamaño del arreglo puede ser cualquier número entero positivo.

5.5.5.4 Redimensionamiento (*resizing*)

A medida que se insertan elementos, el factor de carga $\alpha = n/m$ aumenta. Cuando α supera un umbral predefinido (típicamente 0.75), se redimensiona la tabla para mantener un rendimiento aceptable. La estrategia de redimensionamiento es:

1. Se duplica la capacidad actual y se busca el siguiente número primo mayor o igual a ese valor.
2. Se crea un nuevo arreglo con esa capacidad.
3. Se reubican todos los elementos activos (no eliminados) en el nuevo arreglo, recalculando sus índices con la nueva capacidad.
4. Se actualiza el umbral para la nueva capacidad usando el mismo factor de carga.

Este proceso se conoce como *rehashing* y tiene un costo de $O(n)$, donde n es la cantidad de elementos en la tabla. Si bien es una operación costosa, al duplicar la capacidad ocurre con poca frecuencia: se realizan n inserciones de costo $O(1)$ antes de cada redimensionamiento de $O(n)$, lo que da un costo amortizado de $O(1)$ por inserción.

En el código de ejemplo, el redimensionamiento se activa automáticamente en el método `Put` cuando la cantidad de elementos alcanza el umbral (`threshold`), calculado como `capacidad × factorDeCarga`.

5.5.5.5 Funciones de *hash*

La función de *hash* es el corazón de cualquier tabla de *hash*. Su propósito fundamental es tomar una clave de entrada (que puede ser de cualquier tipo de dato: números, cadenas de texto, objetos complejos, etc.) y transformarla en un índice entero dentro del rango válido del arreglo subyacente de la tabla de *hash* (típicamente de 0 a m−1, donde m es el tamaño del arreglo).

Las dos características primordiales que una buena función de *hash* debe poseer son la eficiencia y la distribución uniforme:

Eficiencia Es crucial que su cálculo sea rápido y se pueda realizar en $O(1)$ con respecto al tamaño de la clave. Las funciones de *hash* se construyen utilizando operaciones aritméticas y lógicas básicas que son inherentemente rápidas a nivel de hardware como sumas, restas, multiplicaciones, divisiones enteras, módulos y operaciones bit a bit como `AND`, `OR`, `XOR`, y desplazamientos de bits.

Distribución Uniforme El objetivo ideal de una función de *hash* es asignar cada clave posible a un índice único dentro del arreglo. Sin embargo, dado que el espacio de posibles claves suele ser mucho mayor que el tamaño del arreglo, las colisiones son inevitables. Una buena función de *hash* minimiza la probabilidad de colisiones al distribuir las claves de entrada de la manera más uniforme posible a lo largo de los índices del arreglo. Esto significa que para un conjunto de claves dado, la función hash debería intentar dispersarlas

equitativamente entre las m posiciones del arreglo, evitando la concentración de muchas claves en unas pocas posiciones.

Por ejemplo, para cadenas de caracteres se puede usar la **Multiplicación Polinómica** que trata la cadena como un polinomio donde los caracteres son los coeficientes de un polinomio. (Más efectivo para evitar colisiones con anagramas).

Sea la cadena $S = c_1 c_2 ... c_n$ donde c_i es el i-ésimo carácter de la cadena. Cada carácter se puede "mapear" a un valor numérico utilizando su código ASCII. Sea a una constante elegida (a menudo un número primo) y m el tamaño del arreglo de la tabla de *hash* (generalmente un número primo). La función de *hash* se puede definir como:

$$h(S) = (c_1 \cdot a^{n-1} + c_2 \cdot a^{n-2} + ... + c_n \cdot a^0) mod m \tag{5.3}$$

El módulo se aplica al resultado final para asegurar que el valor hash esté dentro del rango válido de índices del arreglo $[0, m-1]$.

Ejemplo de implementación de *hashing* cerrado

5.5.6.1 Interfaz `HashTable`

Ver código completo (`interface.go`)

Go

```go
package hashtable

// HashTable define el contrato que deben implementar las tablas de hash.
//
// K debe ser un tipo comparable (puede usarse como clave en un map de Go).
// V puede ser cualquier tipo.
type HashTable[K comparable, V any] interface {
  // Put agrega un nuevo par clave-valor. Si la clave ya existe, actualiza
  // el valor asociado.
  Put(key K, value V)
  // Get devuelve el valor asociado a la clave.
  // Error si la clave no existe.
  Get(key K) (V, error)
  // Delete elimina el par clave-valor asociado a la clave.
  // Error si la clave no existe.
  Delete(key K) error
  // Contains devuelve true si la clave existe en la tabla.
  Contains(key K) bool
  // Size devuelve la cantidad de elementos en la tabla.
  Size() int
  // IsEmpty devuelve true si la tabla no tiene elementos.
  IsEmpty() bool
  // Keys devuelve una lista con todas las claves.
  Keys() []K
  // Values devuelve una lista con todos los valores.
  Values() []V
  // Clear elimina todos los elementos de la tabla.
  Clear()
  // String devuelve una representación en cadena estilo Python dict.
```

```
  // Formato: {clave1: valor1, clave2: valor2}
  String() string
}
```

5.5.6.2 Tipos y operaciones públicas

Ver código completo (`hashtable.go`)

Go

```go
package hashtable

import (
  "errors"
  "fmt"
)

// hashTableEntry representa una entrada en la tabla hash, que contiene una
// clave, su valor asociado y un marcador de eliminación.
//
// El campo deleted (tombstone) evita perder la posición al eliminar: en lugar
// de compactar el arreglo, se marca la entrada como eliminada para que las
// búsquedas lineales no se interrumpan.
type hashTableEntry[K comparable, V any] struct {
  key     K
  value   V
  deleted bool
}

// HashTableOpenAddressing es una implementación de hashing cerrado con sondeo
// lineal que utiliza un arreglo para almacenar pares clave-valor.
type HashTableOpenAddressing[K comparable, V any] struct {
  // arreglo de entradas de la tabla hash.
  buckets []*hashTableEntry[K, V]
  // size es el número de elementos en la tabla.
  size int
  // capacity es la capacidad de la tabla.
  capacity int
  // loadFactor es el factor de carga de la tabla.
  loadFactor float32
  // threshold es el umbral de carga para redimensionar la tabla.
  threshold int
}

// NewHashTableOpenAddressing crea una nueva tabla de hash cerrada con la
// capacidad y el factor de carga especificados.
//
// - Si la capacidad es igual a 0, se establece en 17.
//
// - Si el factor de carga es menor o igual a 0 o mayor que 1, se establece en
// 0.75.
//
// - Si la capacidad no es un número primo, se redimensiona a la siguiente
// capacidad primo mayor o igual a la capacidad especificada.
func NewHashTableOpenAddressing[K comparable, V any](capacity int, loadFactor
```

```
float32) *HashTableOpenAddressing[K, V] {
  if capacity == 0 {
    capacity = 17
  }
  if loadFactor <= 0 || loadFactor > 1 {
    loadFactor = 0.75
  }
  if !isPrime(capacity) {
    capacity = nextPrime(capacity)
  }
  return &HashTableOpenAddressing[K, V]{
    buckets:    make([]*hashTableEntry[K, V], capacity),
    size:       0,
    capacity:   capacity,
    loadFactor: loadFactor,
    threshold:  int(float32(capacity) * loadFactor),
  }
}

// Put agrega un nuevo par clave-valor a la tabla de hash. Si la clave ya
// existe, actualiza el valor asociado a la clave.
func (ht *HashTableOpenAddressing[K, V]) Put(key K, value V) {
  if index, esta := ht.getIndex(key); esta {
    ht.buckets[index].value = value
    return
  }

  if ht.size >= ht.threshold {
    ht.resize()
  }

  index := ht.hash(key) % ht.capacity
  for {
    if ht.buckets[index] == nil || ht.buckets[index].deleted {
      ht.buckets[index] = &hashTableEntry[K, V]{key: key, value: value}
      ht.size++
      return
    }
    index = (index + 1) % ht.capacity
  }
}

// Get devuelve el valor asociado a la clave y nil. Si la clave no existe,
// devuelve el zero value de V y un error.
func (ht *HashTableOpenAddressing[K, V]) Get(key K) (V, error) {
  index, exists := ht.getIndex(key)
  if !exists {
    var zero V
    return zero, errors.New("key not found")
  }
  return ht.buckets[index].value, nil
}

// Delete elimina el par clave-valor asociado a la clave.
// Devuelve error si la clave no existe.
//
```

```go
// Usa eliminación perezosa (tombstone): marca la entrada como deleted en lugar
// de eliminarla del arreglo. Esto mantiene la continuidad de las sondas
// lineales: si se eliminara la entrada, las búsquedas posteriores podrían no
// encontrar elementos que están más allá en la secuencia de sondeo.
func (ht *HashTableOpenAddressing[K, V]) Delete(key K) error {
  index, exists := ht.getIndex(key)
  if !exists {
    return errors.New("key not found")
  }
  ht.buckets[index].deleted = true
  var zero V
  ht.buckets[index].value = zero
  ht.size--
  return nil
}

// Contains devuelve true si la clave existe en la tabla.
func (ht *HashTableOpenAddressing[K, V]) Contains(key K) bool {
  _, exists := ht.getIndex(key)
  return exists
}

// Keys devuelve una lista de todas las claves en la tabla de hash.
//
// Pre-asigna el slice con capacidad ht.size para evitar realocaciones durante
// los append. Solo incluye entradas activas (no eliminadas).
func (ht *HashTableOpenAddressing[K, V]) Keys() []K {
  keys := make([]K, 0, ht.size)
  for _, node := range ht.buckets {
    if node != nil && !node.deleted {
      keys = append(keys, node.key)
    }
  }
  return keys
}

// Values devuelve una lista de todos los valores en la tabla de hash.
//
// Pre-asigna el slice con capacidad ht.size para evitar realocaciones. Solo
// incluye valores de entradas activas (no eliminadas).
func (ht *HashTableOpenAddressing[K, V]) Values() []V {
  values := make([]V, 0, ht.size)
  for _, node := range ht.buckets {
    if node != nil && !node.deleted {
      values = append(values, node.value)
    }
  }
  return values
}

// Size devuelve el número de elementos en la tabla de hash.
func (ht *HashTableOpenAddressing[K, V]) Size() int {
  return ht.size
}

// IsEmpty devuelve true si la tabla de hash está vacía, false en caso contrario.
```

```go
func (ht *HashTableOpenAddressing[K, V]) IsEmpty() bool {
  return ht.size == 0
}

// Clear elimina todos los elementos de la tabla de hash.
func (ht *HashTableOpenAddressing[K, V]) Clear() {
  ht.buckets = make([]*hashTableEntry[K, V], ht.capacity)
  ht.size = 0
}

// String devuelve una representación en cadena de la tabla de hash.
//
// Formato: {clave1: valor1, clave2: valor2}. Tras recorrer todas las entradas
// activas se elimina la última coma y espacio para mantener el formato limpio.
func (ht *HashTableOpenAddressing[K, V]) String() string {
  result := "{"
  for _, node := range ht.buckets {
    if node != nil && !node.deleted {
      result += fmt.Sprintf("%v: %v", node.key, node.value) + ", "
    }
  }
  if len(result) > 1 {
    result = result[:len(result)-2]
  }
  result += "}"
  return result
}
```

5.5.6.3 Funciones internas

Ver código completo (`internal.go`)

Go

```go
package hashtable

import "fmt"

// a es una constante utilizada para calcular el hash de un string
const a int = 11

// hash calcula el índice del bucket para una clave dada.
//
// Se utiliza la técnica de Multiplicación Polinómica con el método de Horner:
// evalúa el polinomio c1*a^{n-1} + c2*a^{n-2} + ... + cn*a^0 aplicando
// la propiedad distributiva para evitar el cálculo costoso de potencias.
//
// La constante a=11 es un número primo pequeño que ofrece buena dispersión
// para cadenas de texto.
//
// Como K es cualquier tipo comparable, se convierte la clave a string con
// fmt.Sprintf para aplicar el algoritmo polinómico sobre su representación.
//
// El módulo por la capacidad NO se aplica acá sino en el llamante, para que
// este mismo valor pueda reusarse al buscar en distintas capacidades (ej.
```

```
// durante el redimensionamiento).
func (ht *HashTableOpenAddressing[K, V]) hash(key K) int {
  s := fmt.Sprintf("%v", key)
  var hash int = 0
  for _, c := range s {
    hash = hash*a + int(c)
  }
  return hash
}

// getIndex devuelve el índice del bucket para una clave dada y un booleano que
// indica si la clave existe.
//
// Realiza una sonda lineal (linear probing): calcula el índice inicial con
// hash(key) % capacity y, si el bucket está ocupado por otra clave, avanza
// secuencialmente hasta encontrar la clave buscada o un bucket vacío (nil).
//
// La condición i < ht.capacity limita la búsqueda para evitar un bucle
// infinito si la tabla estuviera completamente llena (sin buckets nil).
// Al encontrar un bucket nil detiene la búsqueda porque en hashing cerrado
// con sondeo lineal los elementos no saltan buckets vacíos.
func (ht *HashTableOpenAddressing[K, V]) getIndex(key K) (int, bool) {
  for i, index := 0, ht.hash(key)%ht.capacity; i < ht.capacity &&
ht.buckets[index] != nil; i, index = i+1, (index+1)%ht.capacity {
    if !ht.buckets[index].deleted && ht.buckets[index].key == key {
      return index, true
    }
  }
  return 0, false
}

// resize redimensiona la tabla de hash y reubica todos los elementos en la
// nueva tabla.
//
// El nuevo tamaño es el siguiente número primo mayor o igual al doble de la
// capacidad actual.
//
// Se itera sobre todos los buckets del arreglo viejo y se reinsertan solo las
// entradas activas (no eliminadas). Las entradas marcadas como deleted se
// descartan, liberando así la memoria que ocupaban.
//
// Tras la reubicación se actualizan capacity y threshold. El factor de carga
// loadFactor se mantiene igual. Esto asegura que la próxima vez que se alcance
// el umbral se vuelva a redimensionar.
func (ht *HashTableOpenAddressing[K, V]) resize() {
  newCapacity := nextPrime(ht.capacity * 2)
  newBuckets := make([]*hashTableEntry[K, V], newCapacity)

  for _, node := range ht.buckets {
    if node != nil && !node.deleted {
      index := ht.hash(node.key) % newCapacity
      for newBuckets[index] != nil {
        index = (index + 1) % newCapacity
      }
      newBuckets[index] = node
    }
```

```
  }

  ht.buckets = newBuckets
  ht.capacity = newCapacity
  ht.threshold = int(float32(newCapacity) * ht.loadFactor)
}

// nextPrime devuelve el siguiente número primo mayor o igual a n.
//
// La búsqueda es secuencial: prueba cada número a partir de n hacia arriba
// hasta encontrar un primo. Como la frecuencia de primos es aproximadamente
// n / ln(n), rara vez recorre muchos números.
func nextPrime(n int) int {
  if n <= 1 {
    return 2
  }
  for i := n; ; i++ {
    if isPrime(i) {
      return i
    }
  }
}

// isPrime devuelve true si n es un número primo, false en caso contrario.
//
// Optimización 6k ± 1: todo primo mayor a 3 puede expresarse como 6k ± 1.
// Esto permite verificar divisibilidad solo con números de esa forma,
// reduciendo las iteraciones a ~1/3 de una verificación ingenua.
func isPrime(n int) bool {
  if n <= 1 {
    return false
  }
  if n <= 3 {
    return true
  }
  if n%2 == 0 || n%3 == 0 {
    return false
  }
  for i := 5; i*i <= n; i += 6 {
    if n%i == 0 || n%(i+2) == 0 {
      return false
    }
  }
  return true
}
```

La implementación concreta `HashTableOpenAddressing` satisface la interfaz `HashTable` para cualquier tipo de clave `K comparable` y valor `V any`. Al usar `K comparable`, la tabla acepta cualquier tipo que pueda usarse como clave en un `map` de Go (strings, números, structs sin slices, etc.). La función de *hash* convierte la clave a string con `fmt.Sprintf` para aplicar el algoritmo polinómico sobre su representación.

Ejercicios

1. **Implementar tabla de hash con encadenamiento** — `HashTableOpenAddressing` (sondeo lineal) ya está implementado en `data-structures/hashtable/` como referencia. Completar el esqueleto de `HashTableChaining` (encadenamiento separado) en el mismo paquete. Al ser `K comparable`, deben implementar una función de *hash* que funcione para cualquier tipo comparable; pueden usar el paquete `hash/maphash` de la biblioteca estándar de Go.
2. **Resolver ejercicios de aplicación** — Los ejercicios de este capítulo están en `05-hashing/ejercicios/` del repositorio `taller-tad`.

5.6 Conjuntos

Los conjuntos son estructuras de datos que permiten almacenar elementos de forma **desordenada** y **sin repetición**. Son similares a los conjuntos matemáticos: dos conjuntos son iguales si tienen los mismos elementos, sin importar el orden en que se encuentren.

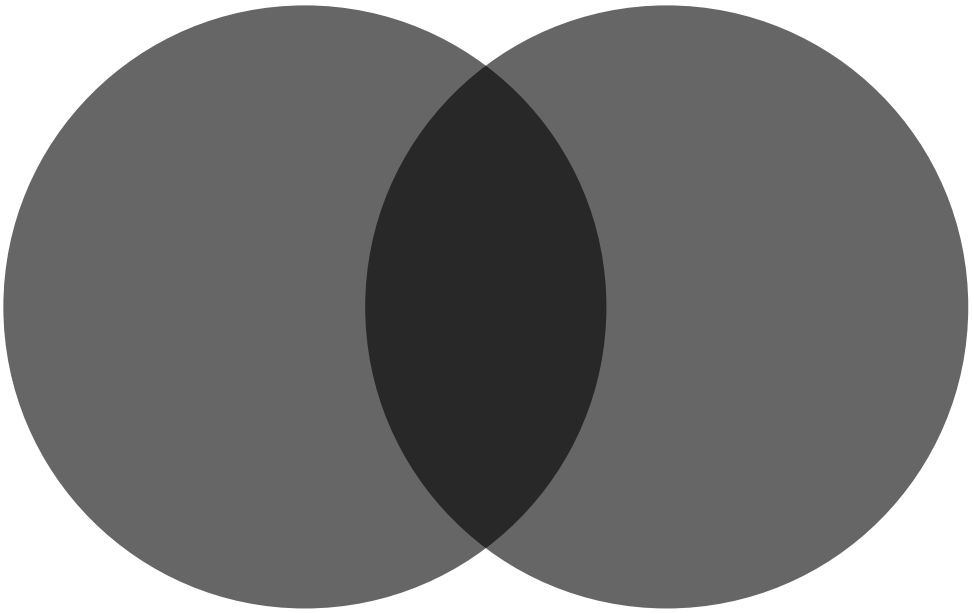


Figura 5.76: Diagrama de Venn: dos conjuntos con intersección no vacía

Las operaciones básicas que se pueden realizar con conjuntos se dividen en dos categorías: operaciones sobre los elementos y operaciones entre conjuntos.

Operaciones sobre los elementos

`Contains(e T) bool` Verifica si el elemento `e` pertenece al conjunto. Debe ser $O(1)$ promedio.

`Add(e T)` Agrega el elemento `e` al conjunto. Si ya existe, no tiene efecto. Debe ser $O(1)$ promedio.

`Remove(e T)` Elimina el elemento `e` del conjunto. Si no existe, no tiene efecto. Debe ser $O(1)$ promedio.

`Size() int` Devuelve la cantidad de elementos del conjunto. Debe ser $O(1)$.

`Values() []T` Devuelve un slice con todos los elementos del conjunto, sin un orden definido. Debe ser $O(n)$.

`String() string` Devuelve una representación textual del conjunto, por ejemplo `{1, 2, 3}`. Debe ser $O(n)$.

Operaciones entre conjuntos

`Union(other Set[T]) Set[T]` Devuelve un nuevo conjunto con los elementos de ambos conjuntos. Debe ser $O(n + m)$.

`Intersection(other Set[T]) Set[T]` Devuelve un nuevo conjunto con los elementos comunes a ambos conjuntos. Debe ser $O(\min(n, m))$.

`Difference(other Set[T]) Set[T]` Devuelve un nuevo conjunto con los elementos que están en el receptor pero no en `other`. Debe ser $O(n)$.

`SymmetricDifference(other Set[T]) Set[T]` Devuelve un nuevo conjunto con los elementos que están en uno de los conjuntos pero no en ambos. Debe ser $O(n + m)$.

`Subset(other Set[T]) bool` Verifica si el receptor es subconjunto de `other`. Debe ser $O(n)$.

`Superset(other Set[T]) bool` Verifica si el receptor es superconjunto de `other`, es decir, si `other` es subconjunto del receptor. Debe ser $O(m)$.

Interfaz

Todas estas operaciones definen la interfaz del TAD Conjunto:

Go

```go
type Set[T comparable] interface {
    // Operaciones sobre elementos
    Contains(element T) bool
    Add(element T)
    Remove(element T)
    Size() int
    Values() []T
    String() string

    // Operaciones entre conjuntos
    Union(other Set[T]) Set[T]
    Intersection(other Set[T]) Set[T]
    Difference(other Set[T]) Set[T]
    SymmetricDifference(other Set[T]) Set[T]
    Subset(other Set[T]) bool
    Superset(other Set[T]) bool
}
```

Estrategias de implementación

No existe un tipo nativo `Set` en Go, pero existen varias formas de implementarlo. La elección depende del balance entre eficiencia y simplicidad.

5.6.4.1 Sobre `map[T]struct{}`

La forma más directa es usar un `map` nativo de Go con valores de tipo `struct{}`. Como las claves del mapa son únicas y `struct{}` no ocupa memoria adicional (es un tipo de tamaño cero), esta combinación es el *idiom* recomendado en Go para representar conjuntos:

Go

```go
type setMap[T comparable] struct {
    elements map[T]struct{}
}
```

Las operaciones básicas se traducen naturalmente:

- **Agregar**: `s.elements[elem] = struct{}{}`
- **Pertenencia**: `_, ok := s.elements[elem]`
- **Eliminar**: `delete(s.elements, elem)`
- **Cantidad**: `len(s.elements)`

Todas estas operaciones son $O(1)$ promedio gracias a la tabla de *hash* interna de `map`. Las operaciones que recorren todos los elementos (`Values`, `String`, `Union`, etc.) son $O(n)$.

Esta implementación es la más simple y eficiente para la mayoría de los casos, pero tiene una limitación: el tipo `T` debe ser `comparable` (como exige Go para las claves de un mapa).

5.6.4.2 Sobre una tabla de *hash*

Un conjunto puede verse como una tabla de *hash* donde solo importan las claves y el valor asociado es irrelevante. El módulo `data-structures` (dentro del repositorio `taller-tad`) provee una implementación genérica de `HashTable[K comparable, V any]` que puede usarse como base:

Go

```go
type setHash[T comparable] struct {
    table hashtable.HashTable[T, struct{}]
}
```

El comportamiento es análogo al de `map[T]struct{}` con la misma complejidad $O(1)$ promedio para las operaciones básicas. La ventaja es que al implementar la tabla de *hash* uno controla la función de *hash*, la política de colisiones y el redimensionamiento, lo que permite adaptar el comportamiento a necesidades específicas.

5.6.4.3 Otras alternativas

También es posible implementar un conjunto sobre un arreglo dinámico (*slice*) o sobre una lista enlazada. En estos casos las operaciones de búsqueda pasan a ser $O(n)$, lo que hace que `Contains`, `Add` (que debe verificar si el elemento ya existe) y `Remove` sean lineales. Solo conviene usarlas para conjuntos muy pequeños o cuando la eficiencia no es crítica.

Conjuntos ordenados

Un conjunto ordenado es una variante del TAD Conjunto que, además de no permitir duplicados, mantiene los elementos organizados según un criterio de orden. Esto habilita operaciones que la versión con *hash* no puede realizar de forma eficiente.

5.6.5.1 Para qué se usan

Los conjuntos ordenados son útiles cuando se necesita:

- **Obtener el mínimo o el máximo** elemento del conjunto.
- **Recorrer los elementos en orden** ascendente o descendente.
- **Buscar por rango**, por ejemplo obtener todos los elementos entre dos valores dados.
- **Combinar conjuntos ordenados** mediante un *merge* en tiempo $O(n + m)$, más eficiente que la alternativa con tablas de *hash*.

Algunos ejemplos concretos: mantener una lista de palabras en orden alfabético, un ranking de puntajes ordenados, o un calendario de eventos donde se necesita consultar el próximo evento después de una fecha dada.

5.6.5.2 Estrategias de implementación

Lista enlazada ordenada: es la implementación más sencilla. La lista se mantiene ordenada en todo momento: al agregar un elemento se recorre hasta encontrar la posición correcta según el criterio de orden. Esto hace que `Add`, `Remove` y `Contains` sean $O(n)$, ya que en el peor caso hay que recorrer toda la lista. La `List` del módulo `data-structures` (dentro de `taller-tad`) puede usarse como base.

Arreglo ordenado (*slice*): los elementos se mantienen en un slice ordenado. `Contains` puede usar búsqueda binaria ($O(\log n)$), pero `Add` y `Remove` requieren desplazar elementos ($O(n)$). Es adecuado cuando el conjunto es mayormente estático (muchas consultas, pocas modificaciones).

Árbol binario de búsqueda balanceado: es la opción más eficiente pero también la más compleja. Un árbol AVL (que se estudia en el capítulo Árboles Binarios de Búsqueda Balanceados) mantiene los elementos ordenados y ofrece $O(\log n)$ en inserción, eliminación y búsqueda.

5.6.5.3 Orden de las operaciones

La siguiente tabla compara las complejidades del conjunto ordenado según la estructura subyacente. Se incluye también el conjunto con *hash* como referencia:

Operación	Conjunto con *hash*	Lista enlazada	Slice ordenado	Árbol AVL
`Add`	$O(1)$ prom.	$O(n)$	$O(n)$	$O(\log n)$
`Remove`	$O(1)$ prom.	$O(n)$	$O(n)$	$O(\log n)$
`Contains`	$O(1)$ prom.	$O(n)$	$O(\log n)$	$O(\log n)$
`Size`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Values`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Union`	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n + m)$
`Intersection`	$O(\min(n, m))$	$O(n + m)$	$O(n + m)$	$O(n + m)$

Las operaciones binarias (`Union`, `Intersection`, etc.) se benefician del orden: pueden resolverse recorriendo ambas estructuras en simultáneo (*merge*), lo que da $O(n + m)$ independientemente de la implementación interna. En el conjunto con *hash*, en cambio, `Intersection` puede aprovechar la búsqueda $O(1)$ del conjunto más chico para lograr $O(\min(n, m))$.

La elección de la estructura depende del uso. Para los ejercicios de este capítulo se trabajará con listas enlazadas, que son la opción más simple de implementar y permiten concentrarse en la lógica de las operaciones entre conjuntos ordenados.

5.6.5.4 Implementación con listas enlazadas

Para que la lista se mantenga ordenada es necesario poder comparar elementos. En Go, los tipos nativos ordenados (enteros, flotantes, strings) pueden usar el constraint `cmp.Ordered` (disponible desde Go 1.21):

Go

```go
import "cmp"

type SortedSetList[T cmp.Ordered] struct {
    list list.LinkedList[T]
}
```

Para tipos que no implementan `cmp.Ordered` se pasa una función de comparación:

Go

```go
type SortedSetListFunc[T any] struct {
    list list.LinkedList[T]
    less func(a, b T) bool
}
```

Invariante de representación

Como se vio en el capítulo Tipos Abstractos de Datos, todo TAD debe cumplir un **invariante de representación**. En el `SortedSetList` el invariante tiene dos partes:

1. La lista interna está **ordenada** según el criterio definido por la función `less`.
2. La lista **no contiene elementos repetidos**.

Ambas condiciones deben cumplirse siempre, antes y después de cada operación, y son responsabilidad de las primitivas del TAD mantenerlas.

`Add`: recorre la lista hasta encontrar la posición donde el nuevo elemento es menor o igual al actual. Si ya existe un elemento igual, no se agrega. Si se llega al final, se agrega al final.

`Contains`: recorre la lista comparando cada elemento. Si se pasa de la posición donde debería estar (el elemento actual es mayor que el buscado), se puede detener la búsqueda: el elemento no está.

`Remove`: recorre la lista hasta encontrar el elemento y lo elimina.

`Union`: recorre ambas listas en simultáneo. En cada paso se toma el menor de los dos elementos actuales, se agrega al resultado y se avanza en la lista correspondiente. Si son iguales, se agrega uno solo y se avanza en ambas.

`Intersection`: similar al *merge* pero solo se agregan los elementos que aparecen en ambas listas.

`Difference`: recorre ambas listas; los elementos que están en la primera pero no en la segunda se agregan al resultado.

Este patrón de *merge* es posible únicamente cuando ambas colecciones están ordenadas, y es la razón principal por la que los conjuntos ordenados son útiles.

Ejercicios

Los ejercicios de este capítulo están en `06-conjuntos/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.7 Diccionarios

Un **diccionario** (también llamado mapa o arreglo asociativo) es un Tipo Abstracto de Datos (TAD) que permite almacenar colecciones de pares de elementos `(clave, valor)`. La clave es un identificador único que nos permite almacenar, recuperar y eliminar de manera eficiente el valor asociado a ella.

La analogía clásica para entender un diccionario es la de una agenda o guía telefónica. En ella, el nombre de la persona actúa como la **clave** y el número de teléfono es el **valor** asociado. Utilizando un índice alfabético (por ejemplo, buscando la letra "M"), podemos localizar rápidamente el número de teléfono de "Maria" sin necesidad de recorrer secuencialmente toda la guía.

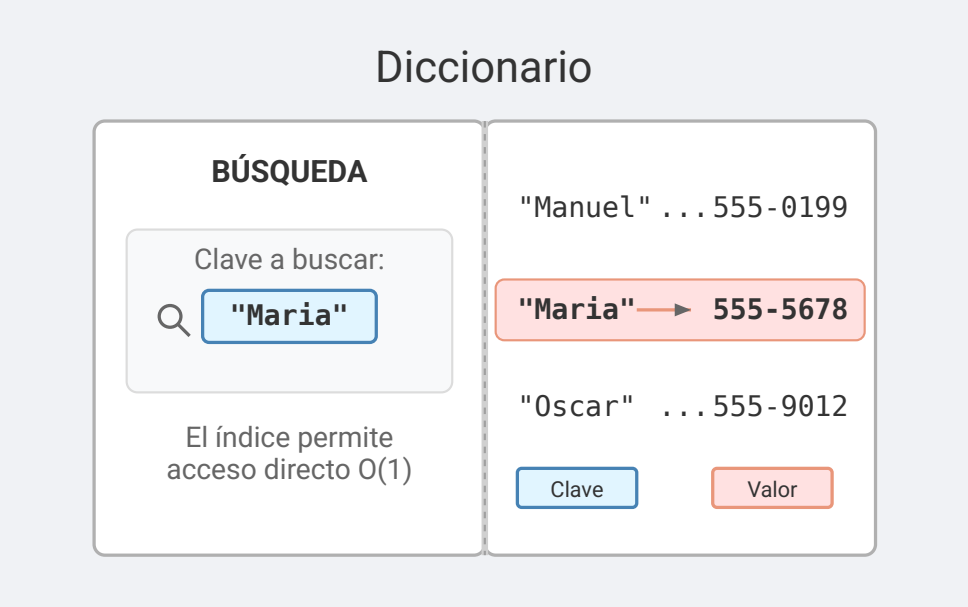


Figura 5.77: Concepto de diccionario como una agenda telefónica indexada.

Relación con las tablas de *hash*

En la práctica, la forma estándar y más eficiente de implementar el TAD Diccionario es mediante una **tabla de *hash*** (estructura que ya estudiamos en detalle en el capítulo Tablas de hash).

Como vimos, una tabla de *hash* utiliza una función de dispersión para mapear las claves a posiciones específicas en un arreglo subyacente. Gracias a esto, un diccionario implementado sobre una tabla de *hash* nos garantiza un costo de tiempo promedio constante ($O(1)$) para las operaciones de inserción, búsqueda y eliminación:

Operación	Complejidad Promedio
Insertar/Actualizar (`Set`)	$O(1)$

Obtener (`Get`)	$O(1)$
Eliminar (`Delete`)	$O(1)$
Verificar existencia (`Contains`)	$O(1)$
Tamaño (`Size`)	$O(1)$
Listar Claves (`Keys`)	$O(n)$
Listar Valores (`Values`)	$O(n)$

Interfaz en Go

Para formalizar este TAD dentro de nuestra biblioteca de estructuras de datos, definimos la *interface* genérica `Dictionary` en Go. El contrato exige que las claves pertenezcan al constraint `comparable` de Go (ya que deben poder compararse por igualdad) y los valores pueden ser de cualquier tipo:

Go

```go
package dictionary

// Dictionary define las operaciones abstractas de un diccionario clave-valor.
type Dictionary[K comparable, V any] interface {
    // Set almacena un par clave-valor. Si la clave ya existe, actualiza su valor.
    Set(key K, value V)

    // Get devuelve el valor asociado a la clave.
    // El booleano indica si la clave existe (sigue el idiom coma-ok de Go).
    Get(key K) (V, bool)

    // Delete elimina la clave y su valor del diccionario. Retorna error si no
existe.
    Delete(key K) error

    // Contains devuelve true si la clave está presente en el diccionario.
    Contains(key K) bool

    // Size devuelve la cantidad total de pares clave-valor almacenados.
    Size() int

    // Keys devuelve un slice con todas las claves presentes en el diccionario.
    Keys() []K

    // Values devuelve un slice con todos los valores presentes en el diccionario.
    Values() []V

    // String devuelve una representación en cadena del diccionario.
    String() string
}
```

El tipo `map` nativo de Go

Go no cuenta con un tipo de datos `Set` o `Stack` nativo, pero sí provee soporte integrado para diccionarios mediante el tipo `map`. Un `map` en Go es una implementación interna y altamente optimizada de una tabla de *hash* abierta.

La sintaxis básica para declarar un mapa asociando claves de tipo `K` con valores de tipo `V` es:

Go

```
var m map[TipoClave]TipoValor
```

5.7.3.1 Inicialización y uso básico

Para poder operar con un mapa en Go es necesario inicializarlo previamente. La forma más común es utilizando la función interna `make()`:

Go

```
// Declaración e inicialización
edades := make(map[string]int)

// Inserción y actualización de elementos
edades["Juan"] = 25
edades["Maria"] = 30

// Recuperación de un valor
edadJuan := edades["Juan"] // edadJuan valdrá 25

// Eliminación de un par clave-valor
delete(edades, "Juan")
```

5.7.3.2 Verificación de existencia (El *idiom* de la coma-ok)

En Go, si intentamos acceder a una clave que no existe en el mapa, el lenguaje no genera un pánico ni un error: nos devuelve el valor cero del tipo del valor (por ejemplo, `0` si el tipo de valor es `int`, o `""` si es `string`).

Para poder distinguir entre una clave que realmente tiene asociado el valor cero y una clave que no existe en absoluto, Go provee la sintaxis especial conocida como el *idiom* de la *comma-ok*:

Go

```
if edad, ok := edades["Juan"]; ok {
    fmt.Printf("Juan tiene %d años\n", edad)
} else {
    fmt.Println("Juan no está en la agenda")
}
```

Aquí, `ok` es un booleano que vale `true` si la clave existe en el mapa y `false` en caso contrario.

5.7.3.3 Recorrido del mapa

Para iterar sobre los pares de un mapa, utilizamos un bucle `for` combinado con la estructura `range`. Es muy importante recordar que **el recorrido de un mapa en Go es no determinista y desordenado** (el orden de los elementos cambia entre ejecuciones consecutivas para evitar que los programas dependan del orden interno de la tabla de *hash*):

Go

```
for nombre, edad := range edades {
    fmt.Printf("%s tiene %d años\n", nombre, edad)
}
```

Estrategias de implementación del TAD

Al momento de construir estructuras que satisfagan nuestra interfaz `Dictionary` en la cursada, existen dos aproximaciones principales:

1. **Envoltura de `map` de Go**: Se puede implementar la interfaz construyendo un *struct* que encapsule un mapa nativo de Go. Esta es la forma más rápida y directa, delegando la lógica de dispersión y colisiones al compilador.
2. **Uso de la `HashTable` de la cursada**: Se puede implementar la interfaz encapsulando la estructura genérica `HashTable` que implementamos en el capítulo 3-5 (ya sea la versión con sondeo lineal o la de encadenamiento separado). Al hacerlo, el diccionario simplemente actúa como una capa de abstracción sobre los métodos ya testeados de la tabla de *hash*.

Inyección de la estructura base por constructor

En el esquema del módulo `data-structures` (dentro de `taller-tad`), la implementación `HashMapDictionary` no crea la `HashTable` internamente, sino que la recibe ya construida a través del constructor:

Go

```
func NewHashMapDictionary[K comparable, V any](
    table hashtable.HashTable[K, V],
) *HashMapDictionary[K, V] {
    return &HashMapDictionary[K, V]{table: table}
}
```

Esta técnica se conoce como **inyección de dependencias** y es un mecanismo habitual para componer TADs a partir de otros TADs más pequeños. Al recibir la tabla de *hash* por parámetro:

- El diccionario funciona con *cualquier* implementación de `HashTable`. Podemos pasarle una `*HashTableChaining`, una `*HashTableOpenAddressing`, o incluso un *mock* en los tests.

- Separamos la responsabilidad de construir la tabla de la responsabilidad de usarla. Quien crea el diccionario decide qué variante de `HashTable` emplear.
- La delegación es total: cada método del diccionario se limita a invocar el método correspondiente sobre la tabla interna.

La siguiente porción de código muestra cómo usaríamos el diccionario con distintas implementaciones de la tabla de *hash*:

Go

```go
// Diccionario con encadenamiento separado
tablaEncadenada := hashtable.NewHashTableChaining[string, int]()
dict1 := NewHashMapDictionary[string, int](tablaEncadenada)

// Diccionario con sondeo lineal (misma interfaz)
tablaSondeo := hashtable.NewHashTableOpenAddressing[string, int](0, 0)
dict2 := NewHashMapDictionary[string, int](tablaSondeo)
```

Este mismo patrón es el que aplicamos en el capítulo anterior con `NewHashTableSet`: el constructor recibe (o crea internamente) una `HashTable[T, struct{}]` para almacenar los elementos del conjunto. La diferencia es que en `HashMapDictionary` exponemos la tabla en el constructor para que quien use la estructura pueda elegir la implementación más conveniente según el contexto.

Ejercicios

Los ejercicios de este capítulo están en `07-diccionarios/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.8 Árboles

Un árbol de datos es una estructura jerárquica compuesta por nodos interconectados mediante aristas. Se caracteriza por tener un único nodo **raíz**, en la cima de la jerarquía, del cual pueden descender cero o más nodos hijos. Los árboles son herramientas fundamentales para modelar relaciones jerárquicas, como la organización de sistemas de archivos, la estructura de directorios o las dependencias entre objetos.

En todo árbol, la **raíz** se distingue por carecer de nodo padre. El resto de los nodos poseen un único padre y pueden tener múltiples (o ningún) hijo. Aquellos nodos sin descendientes se denominan **hojas**. Los nodos que no son hojas ni la raíz se conocen como **nodos internos**.

Una propiedad esencial de los árboles es la existencia de un único camino que conecta la raíz con cada una de las hojas.

Consideremos el árbol genérico ilustrado: el nodo A representa la **raíz**, mientras que D, C, N y K son las **hojas** y los nodos R y Z son **nodos internos**.

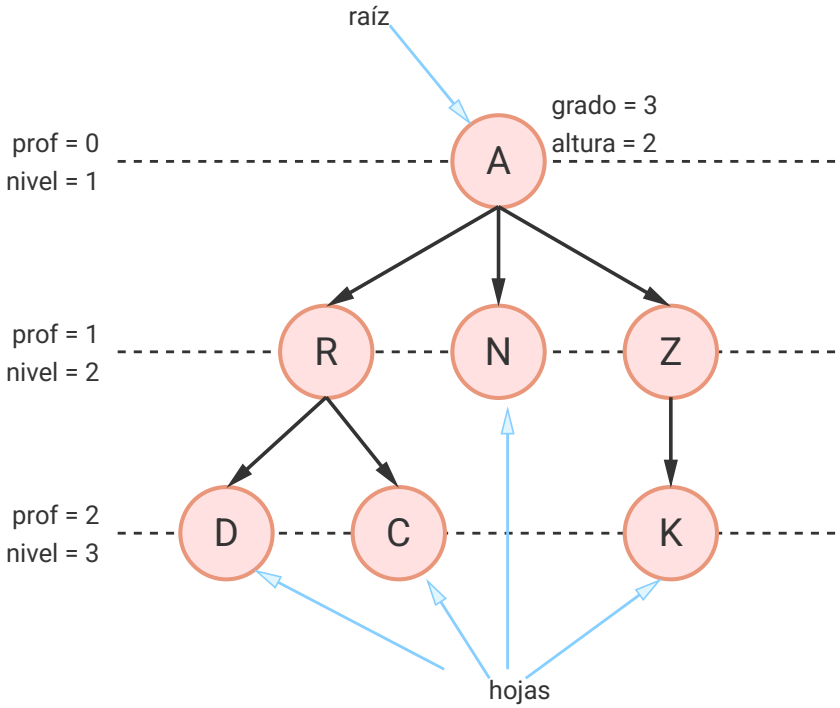


Figura 5.78: Árbol de datos

Propiedades de los árboles

Altura La altura de un árbol es la longitud del camino más largo desde la raíz hasta una hoja. En el caso de un árbol vacío, la altura se define como −1.

Profundidad La profundidad de un nodo es la longitud del camino desde la raíz hasta ese nodo. La profundidad de la raíz es 0.

Nivel El nivel de un nodo es la profundidad del nodo más uno. La raíz se encuentra en el nivel 1, sus hijos en el nivel 2, y así sucesivamente.

Grado El grado de un nodo es el número de hijos que tiene. Un nodo hoja tiene un grado de 0, mientras que la raíz puede tener un grado mayor o igual a 0.

En la figura anterior, el árbol tiene una altura de 2, ya que el camino más largo desde la raíz hasta las hojas C, D o K, que son las más profundas del árbol, tiene una longitud de 2. La raíz A tiene un grado de 3, ya que tiene tres hijos: R, N y Z. Estos nodos son hermanos ya que tienen el mismo padre. El nodo R tiene un grado de 2, ya que tiene dos hijos: D y C. El nodo C es una hoja, por lo que su grado es 0. La profundidad de los nodos R, N y Z es 1, ya que están a un nivel de la raíz. A su vez, N también es una hoja.

Árboles binarios

Los árboles binarios son árboles donde cada nodo tiene a lo sumo dos hijos. Estos hijos se denominan hijo izquierdo e hijo derecho. Los árboles binarios son ampliamente utilizados en informática debido a su simplicidad y eficiencia en diversas operaciones.

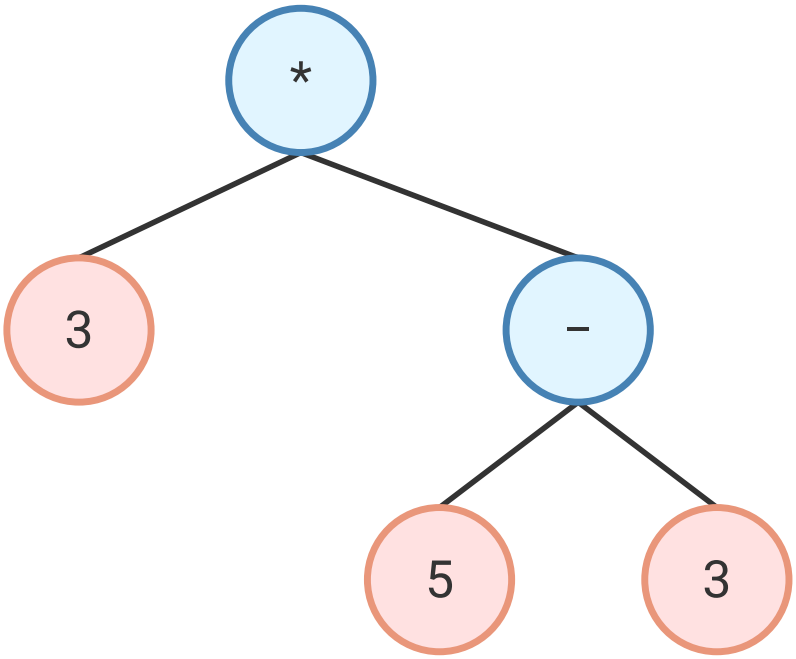


Figura 5.79: Árbol Binario

Definición de Árbol Binario

Un árbol binario es por definición:

- Un árbol vacío, o

- Un nodo raíz con un hijo izquierdo y un hijo derecho, donde cada uno de estos hijos es a su vez un árbol binario.

La definición recursiva de un árbol binario implica que cada subárbol también es un árbol binario. Esto permite construir árboles binarios de manera jerárquica y recursiva, lo que facilita su manipulación y análisis.

5.8.2.1 Recorridos de un árbol binario

Los **recorridos de un árbol binario** son algoritmos fundamentales para visitar y procesar cada uno de los nodos que componen la estructura, siguiendo un orden específico. Existen tres tipos principales de recorridos en profundidad: preorden, inorden y postorden.

La característica distintiva de estos recorridos radica en su naturaleza recursiva. Al aplicar un recorrido a un árbol, el mismo proceso se aplica de manera independiente a sus subárboles izquierdo y derecho. Esto asegura que cada nodo dentro de la estructura sea visitado siguiendo el patrón definido por el tipo de recorrido.

Estos métodos son esenciales en diversas aplicaciones informáticas, incluyendo la búsqueda eficiente de información, la ordenación de datos y la evaluación de expresiones aritméticas o lógicas representadas en forma de árbol. Cada tipo de recorrido ofrece una secuencia de visita única, lo que los hace adecuados para diferentes contextos y tareas.

5.8.2.1.1 Preorden (Raíz - Izquierda - Derecha)

El recorrido preorden comienza visitando la raíz del árbol, seguido por la exploración recursiva del subárbol izquierdo y, finalmente, el subárbol derecho. El orden de procesamiento es, por lo tanto: nodo raíz, subárbol izquierdo, subárbol derecho.

El siguiente seudocódigo ilustra la implementación del recorrido preorden como un método dentro de un nodo de un árbol binario, delegando la lógica del recorrido a cada nodo:

text

```
FUNCION Preorden(raiz)
    SI raiz ES nulo ENTONCES
        retornar
    FIN SI
    Procesar(raiz)           // Visitar el nodo actual
    Preorden(raiz.izquierdo) // Recorrer el subárbol izquierdo
    Preorden(raiz.derecho)   // Recorrer el subárbol derecho
FIN FUNCION
```

`Procesar(raiz)` Representa una operación genérica que se aplica al nodo actual. Esta operación puede variar según la implementación, por ejemplo, podría ser imprimir el valor del nodo o añadirlo a una lista.

Llamadas recursivas Las líneas 6 y 7 realizan las llamadas recursivas para aplicar el recorrido preorden a los subárboles izquierdo y derecho del nodo actual, respectivamente.

5.8.2.1.2 Inorden (Izquierda - Raíz - Derecha)

El recorrido inorden visita primero de forma recursiva todos los nodos del subárbol izquierdo, luego procesa el nodo raíz, y finalmente recorre recursivamente el subárbol derecho. La secuencia de visita es: subárbol izquierdo, nodo raíz, subárbol derecho.

El siguiente seudocódigo muestra la implementación del recorrido inorden:

text

```
FUNCION Inorden (raiz)
    SI raiz ES nulo ENTONCES
        retornar
    FIN SI
    Inorden(raiz.izquierdo) // Visitar el subárbol izquierdo
    Procesar(raiz)          // Visitar el nodo actual
    Inorden(raiz.derecho)   // Visitar el subárbol derecho
FIN FUNCION
```

5.8.2.1.3 Postorden (Izquierda - Derecha - Raíz)

En el recorrido postorden, se exploran recursivamente el subárbol izquierdo, seguido del subárbol derecho, y finalmente se procesa el nodo raíz. El orden de visita es: subárbol izquierdo, subárbol derecho, nodo raíz.

El seudocódigo para el recorrido postorden se presenta a continuación:

text

```
FUNCION Postorden (raiz)
    SI raiz ES nulo ENTONCES
        retornar
    FIN SI
    Postorden(raiz.izquierdo) // Visitar el subárbol izquierdo
    Postorden(raiz.derecho)   // Visitar el subárbol derecho
    Procesar(raiz)            // Visitar el nodo actual
FIN FUNCION
```

5.8.2.2 Implementación

A continuación se presentan las estructuras que definen un árbol binario en Go: el nodo y el árbol que lo contiene. El tipo `T` es genérico, lo que permite almacenar cualquier tipo de valor sin imponer restricciones de orden.

Go

```go
package tree

// TreeNode representa un nodo de un árbol binario.
type TreeNode[T any] struct {
    Value T
    Left  *TreeNode[T]
```

```
    Right *TreeNode[T]
}
```

Go

```go
package tree

// BinaryTree representa un árbol binario genérico.
type BinaryTree[T any] struct {
    Root *TreeNode[T]
}
```

El árbol posee una referencia a la raíz, mientras que los nodos se encargan de enlazar los subárboles. La pregunta al implementar las operaciones (recorridos, altura, cantidad de nodos) es **quién** las ejecuta: el árbol o los nodos.

5.8.2.2.1 Operaciones delegadas en los nodos

En este enfoque, cada nodo es responsable de aplicar la operación sobre sí mismo y sus descendientes. El árbol solo inicia el proceso invocando al nodo raíz.

text

```
FUNCION TreeNode.Inorden(resultado)
    SI this.izquierdo NO es nulo ENTONCES
        this.izquierdo.Inorden(resultado)
    FIN SI
    resultado.agregar(this.valor)
    SI this.derecho NO es nulo ENTONCES
        this.derecho.Inorden(resultado)
    FIN SI
FIN FUNCION

FUNCION BinaryTree.Inorden()
    resultado = []
    SI this.raiz NO es nula ENTONCES
        this.raiz.Inorden(resultado)
    FIN SI
    RETORNAR resultado
FIN FUNCION
```

El árbol actúa como mero punto de entrada: crea el *slice* vacío y lo pasa al nodo raíz, que lo completa recursivamente. Cada nodo visita a sus hijos antes o después de procesarse según el tipo de recorrido.

Ventajas:

- Cada nodo encapsula su propia lógica de recorrido.
- No se necesita pasar el árbol como contexto en las llamadas recursivas.
- El código del nodo es autónomo y reutilizable en otros contextos.

Desventajas:

- El nodo debe conocer el propósito de la operación (ej: acumular en un *slice*), lo que acopla la estructura del nodo a la operación concreta.
- Si se agrega un nuevo recorrido, hay que modificar el nodo.

5.8.2.2.2 Operaciones gestionadas por el árbol

En este enfoque, el árbol implementa la lógica recursiva y el nodo solo expone sus campos. El árbol recorre la estructura comparando valores desde la raíz.

text

```
FUNCION BinaryTree.inordenRecursivo(nodo, resultado)
    SI nodo ES nulo ENTONCES
        RETORNAR
    FIN SI
    este.inordenRecursivo(nodo.izquierdo, resultado)
    resultado.agregar(nodo.valor)
    este.inordenRecursivo(nodo.derecho, resultado)
FIN FUNCION
```

El árbol tiene el control completo del recorrido y usa el nodo solo como dato. La función recursiva es un método privado del árbol.

Ventajas:

- El nodo es una estructura pasiva (POJO/Plain Old Go *struct*): solo almacena datos.
- Las operaciones están centralizadas en el árbol, facilitando su mantenimiento.
- Se puede cambiar la lógica del recorrido sin modificar el nodo.

Desventajas:

- El árbol necesita acceder a los campos internos del nodo (Left, Right), lo que crea un acoplamiento estructural.
- Cada operación duplica la lógica de navegación (izquierda/derecha).

5.8.2.2.3 Comparación

Aspecto	Delegado al nodo	Gestionado por el árbol
Responsabilidad	Cada nodo procesa sus hijos	El árbol dirige el recorrido
Acoplamiento	El nodo conoce la operación	El árbol conoce la estructura del nodo
Modularidad	Alta (nodo autónomo)	Baja (lógica centralizada)
Extensibilidad	Agregar operaciones requiere modificar el nodo	Agregar operaciones solo requiere modificar el árbol
Nodo	Activo (tiene métodos)	Pasivo (solo datos)

Ambos enfoques son válidos. La elección depende del contexto: si se prioriza que el nodo sea una estructura de datos pura, conviene gestionar las operaciones desde el árbol. Si se busca que el nodo sea una entidad activa capaz de manipularse a sí misma, la delegación es más natural.

Nota

> En los ejercicios de este capítulo se opta por el enfoque de **delegación en los nodos**.

Ejercicios

Los ejercicios de este capítulo están en `08-arboles/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.9 Árboles Binarios de Búsqueda

Definición de Árbol Binario de Búsqueda

Un árbol binario de búsqueda es un árbol binario que cumple con las siguientes propiedades para cada nodo N:

- Todos los nodos en el subárbol izquierdo de N tienen valores menores que el valor de N.
- Todos los nodos en el subárbol derecho de N tienen valores mayores que el valor de N.
- Ambos subárboles (izquierdo y derecho) deben ser también árboles binarios de búsqueda.

En la siguiente figura se muestran dos árboles binarios. El de la izquierda es un ABB válido: su raíz es (7), el hijo izquierdo es (3) y el derecho es (9). A su vez, (3) tiene como hijos a (2) y (5), mientras que (9) tiene a (8) y (10). En todos los nodos se cumple que los valores del subárbol izquierdo son menores que la raíz y los del derecho son mayores.

El árbol de la derecha, en cambio, no es un ABB. Su raíz también es (7), con subárbol izquierdo de raíz (2) y subárbol derecho de raíz (9). El subárbol con raíz (2) tiene como hijos a (1) y (8), y el de raíz (9) tiene a (10) como hijo derecho. Si bien los subárboles izquierdo (con raíz 2) y derecho (con raíz 9) son ABB válidos, el árbol completo no lo es: el nodo (8) —marcado en rojo — está en el subárbol izquierdo de la raíz (7) pero (8) es mayor que (7), violando la propiedad fundamental del ABB.

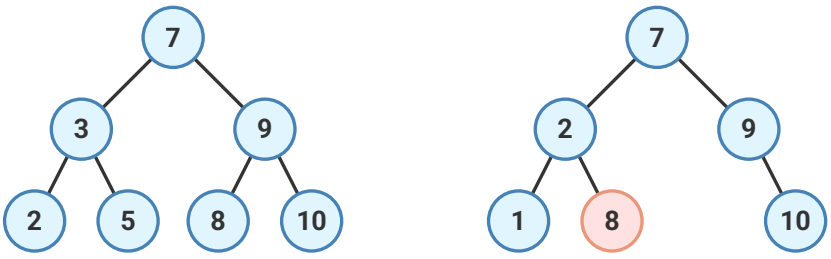
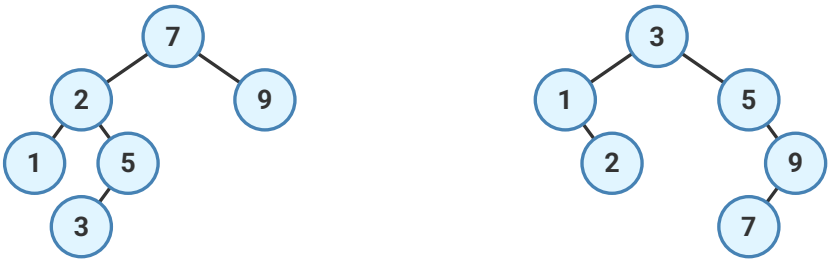


ABB No es ABB

El recorrido inorden de un árbol binario de búsqueda produce una secuencia ordenada de los valores almacenados en el árbol. Esto se debe a que, al visitar el subárbol izquierdo, el nodo raíz y luego el subárbol derecho, se garantiza que los nodos se procesen en orden ascendente.

Por ejemplo, en la siguiente figura se muestran dos árboles binarios de búsqueda que tienen el mismo recorrido inorden: 1, 2, 3, 5, 7, 9. Los árboles son diferentes porque los elementos se insertaron en diferente orden; sin embargo, al ser un ABB, el recorrido inorden es el mismo.



Árbol 1 Árbol 2

Figura 5.81: Dos árboles binarios de búsqueda con el mismo recorrido inorden

Operaciones en un árbol binario de búsqueda

Las operaciones más comunes en un árbol binario de búsqueda son:

- **Inserción**: Agregar un nuevo nodo al árbol manteniendo la propiedad de ABB.
- **Búsqueda**: Encontrar un nodo en el árbol.
- **Eliminación**: Quitar un nodo del árbol manteniendo la propiedad de ABB.

5.9.1.1 Inserción

La inserción en un árbol binario de búsqueda se realiza de manera recursiva. Se compara el valor a insertar con el valor del nodo actual y se decide si ir al subárbol izquierdo o derecho. Si el subárbol correspondiente está vacío, se inserta el nuevo nodo.

Para insertar un nuevo nodo en un árbol binario de búsqueda, se sigue el siguiente algoritmo:

text

```
FUNCION InsertarABB(raiz, valor)
    SI raiz ES nula ENTONCES
        raiz ← CrearNodo(valor)
    SINO SI valor < raiz.valor ENTONCES
        raiz.izquierdo ← InsertarABB(raiz.izquierdo, valor)
    SINO SI valor > raiz.valor ENTONCES
        raiz.derecho ← InsertarABB(raiz.derecho, valor)
    FIN SI
    RETORNAR raiz
FIN FUNCION
```

1. Se compara el valor a insertar con la raíz. Si es menor se desciende al subárbol izquierdo; si es mayor, al derecho.
2. Se repite el proceso recursivamente hasta encontrar un subárbol vacío donde insertar el nuevo nodo.
3. Una vez encontrada la posición, se inserta el nodo como hoja del árbol.

Importante

La inserción no permite valores duplicados. Si se requiere contar los elementos repetidos en el árbol, se puede modificar el algoritmo para que en lugar de insertar un nuevo nodo, se incremente un contador en el nodo existente.

La inserción de un nuevo nodo siempre se realiza en una hoja del árbol, es decir, un nodo que no tiene hijos. Esto asegura que la estructura del árbol se mantenga y que la propiedad de ABB se respete.

5.9.1.2 Búsqueda

La búsqueda en un árbol binario de búsqueda también se realiza de manera recursiva. Se compara el valor buscado con el valor del nodo actual y se decide si ir al subárbol izquierdo o derecho. Si el valor es igual al del nodo actual, se ha encontrado el nodo, si en cambio se llega a un nodo nulo, el valor no está en el árbol.

text

```
FUNCION BuscarABB(raiz, valor)
    SI raiz ES nula ENTONCES
        RETORNAR nula
    SINO SI raiz.valor = valor ENTONCES
        RETORNAR raiz
    SINO SI valor < raiz.valor ENTONCES
        RETORNAR BuscarABB(raiz.izquierdo, valor)
    SINO
        RETORNAR BuscarABB(raiz.derecho, valor)
    FIN SI
FIN FUNCION
```

5.9.1.3 Eliminación

La eliminación de un nodo en un árbol binario de búsqueda es un poco más compleja que la inserción y la búsqueda. Existen tres casos a considerar:

El nodo a eliminar es una hoja (no tiene hijos) En este caso, simplemente se elimina el nodo. En el ejemplo se elimina la hoja (6).

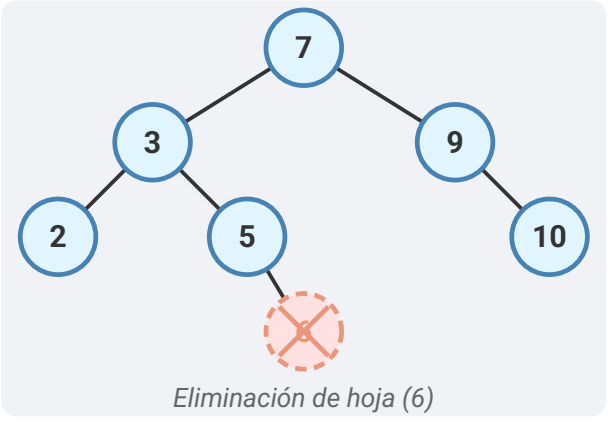


Figura 5.82: Eliminación de un nodo hoja

El nodo a eliminar tiene un solo hijo En este caso, se reemplaza el nodo a eliminar por su hijo. Esto se hace actualizando el puntero del padre del nodo a eliminar para que apunte al hijo

del nodo a eliminar. En el ejemplo se elimina el nodo (5) que tiene sólo un hijo izquierdo (3) y se reemplaza por su hijo.

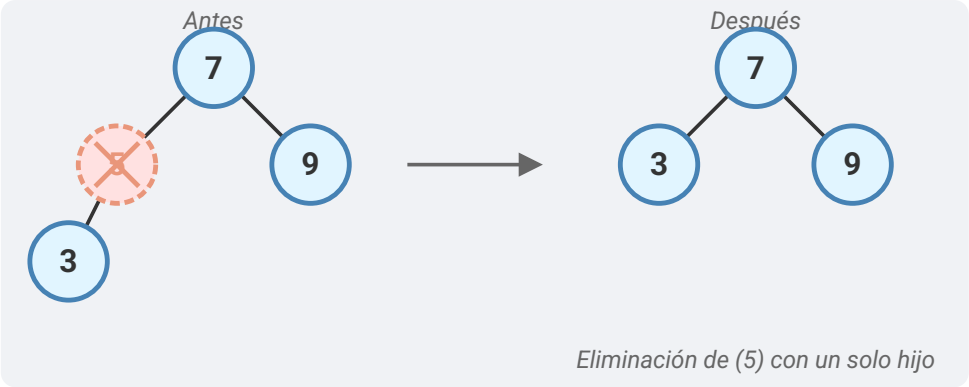


Figura 5.83: Eliminación de un nodo con un solo hijo

El nodo a eliminar tiene dos hijos En este caso, se busca el nodo más pequeño en el subárbol derecho del nodo a eliminar (sucesor) o el nodo más grande en el subárbol izquierdo (predecesor). Luego, se reemplaza el valor del nodo a eliminar por el valor del sucesor o predecesor y se elimina el sucesor o predecesor (que siempre va a ser una hoja o a lo sumo va a tener un solo hijo).

En la siguiente figura se observa la eliminación de la raíz del árbol, el nodo (7).

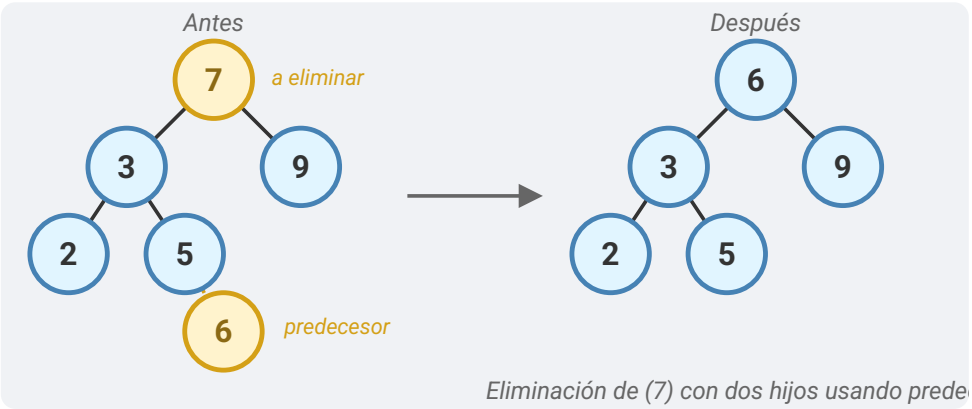


Figura 5.84: Eliminación de un nodo con dos hijos

Para ubicar al predecesor (es decir, el mayor elemento del subárbol izquierdo), se debe bajar al subárbol izquierdo y luego seguir bajando por las ramas derechas hasta llegar a un nodo que no tenga hijo derecho. En este caso el predecesor es (6).

En cambio, para ubicar al sucesor (es decir, el menor elemento del subárbol derecho), se debe bajar al subárbol derecho y luego seguir bajando por las ramas izquierdas hasta llegar a un nodo que no tenga hijo izquierdo. En este caso el sucesor es (9).

En el ejemplo se eligió reemplazar la raíz con el predecesor y eliminar el nodo (6).

Importante

Cuando se elimina un nodo con dos hijos, el nodo no se elimina directamente, sino que se reemplaza por su predecesor o sucesor. Esto asegura mantener la estructura del ABB.

A continuación se presenta el algoritmo de eliminación de un nodo en un árbol binario de búsqueda:

text

```
FUNCION EliminarABB(raiz, valor)
    SI raiz ES nula ENTONCES
        RETORNAR nula
    SINO SI valor < raiz.valor ENTONCES
        raiz.izquierdo ← EliminarABB(raiz.izquierdo, valor)
    SINO SI valor > raiz.valor ENTONCES
        raiz.derecho ← EliminarABB(raiz.derecho, valor)
    SINO
        // Nodo encontrado
        SI raiz.izquierdo ES nula Y
           raiz.derecho ES nula ENTONCES
            // Caso 1: es una hoja
            RETORNAR nula
        SINO SI raiz.izquierdo ES nula ENTONCES
            // Caso 2: tiene solo hijo derecho
            RETORNAR raiz.derecho
        SINO SI raiz.derecho ES nula ENTONCES
            // Caso 2: tiene solo hijo izquierdo
            RETORNAR raiz.izquierdo
        SINO
            // Caso 3: tiene dos hijos,
            // se reemplaza por el predecesor
            predecesor ← BuscarMaximo(raiz.izquierdo)
            raiz.valor ← predecesor.valor
            raiz.izquierdo ← EliminarABB(
                raiz.izquierdo, predecesor.valor)
        FIN SI
    FIN SI
    RETORNAR raiz
FIN FUNCION
```

text

```
FUNCION BuscarMaximo(raiz)
    SI raiz.derecho ES nula ENTONCES
        RETORNAR raiz
    SINO
        RETORNAR BuscarMaximo(raiz.derecho)
    FIN SI
FIN FUNCION
```

5.9.1.4 Orden de las operaciones

En un árbol binario de búsqueda, las operaciones de inserción, búsqueda y eliminación tienen un tiempo de ejecución que depende de la altura del árbol. Por lo tanto, es importante tratar de mantener el árbol equilibrado para que la altura no crezca demasiado.

Un árbol binario de búsqueda equilibrado tiene una altura aproximada de $O(log(n))$, donde n es el número de nodos en el árbol. En este caso, las operaciones de inserción, búsqueda y eliminación tienen un tiempo de ejecución promedio de $O(log(n))$.

Formalmente la altura h de un árbol binario de búsqueda se define como:

$$h = \tag{5.4}$$

En la siguiente figura se observa un árbol binario de búsqueda completo (con todos los nodos en todos los niveles) y perfectamente balanceado, donde cada nodo tiene dos hijos y la altura del árbol es mínima. En este caso, el árbol tiene una altura de 3 y contiene 15 nodos.

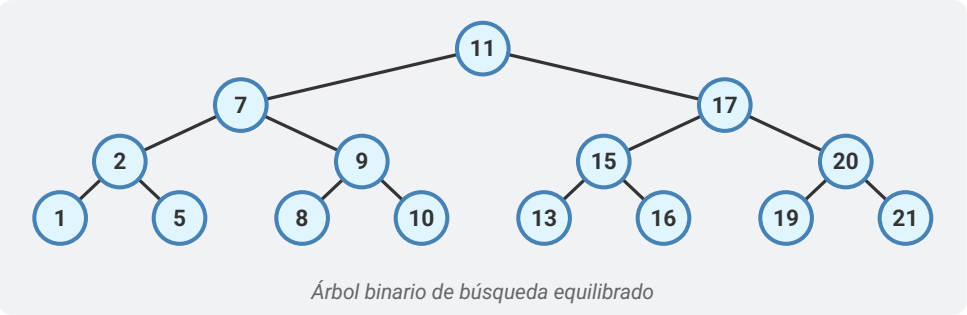


Figura 5.85: Árbol binario de búsqueda equilibrado

La altura h de un árbol binario completo y balanceado se puede calcular como:

$$\begin{aligned} h &= log_2(n+1) - 1 \\ h &= O(log(n)) \end{aligned} \tag{5.5}$$

Por otro lado, en el peor de los casos, un árbol binario de búsqueda puede degenerar en una lista enlazada, lo que resulta en una altura de $O(n)$ y un tiempo de ejecución de $O(n)$ para las operaciones. Esto ocurre cuando los nodos se insertan en orden ascendente o descendente, lo que provoca que el árbol se convierta en una estructura lineal. En la siguiente figura se observa un árbol binario que degeneró en una lista.

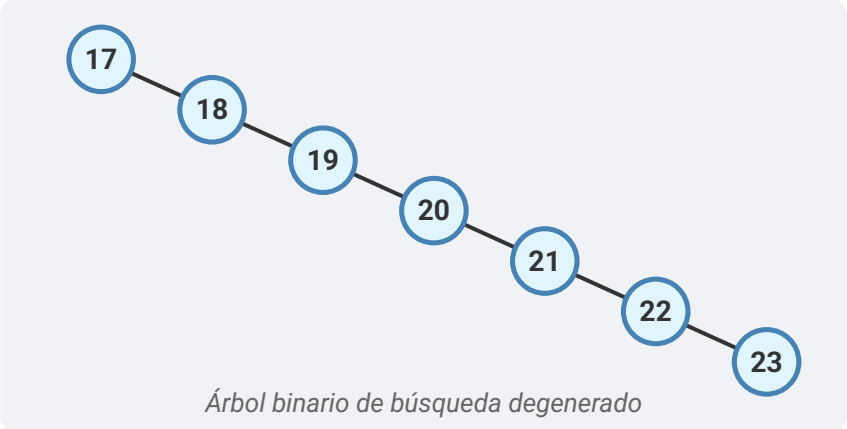


Figura 5.86: Árbol binario de búsqueda degenerado

La altura h de un árbol binario degenerado en una lista se puede calcular como:

$$\begin{aligned} h &= n - 1 \\ h &= O(n) \end{aligned} \tag{5.6}$$

Implementación de un árbol binario de búsqueda

Existen dos enfoques principales para implementar un ABB: construir la estructura desde cero, definiendo explícitamente los nodos, o aprovechar una implementación de árbol binario genérica existente mediante composición. A continuación se analizan ambas alternativas.

En los ejemplos se utiliza una función de comparación `func(T, T) int` que permite trabajar con cualquier tipo de dato, sin restringir a `cmp.Ordered`. Esta función debe devolver un valor negativo si el primer argumento es menor, cero si son iguales, y positivo si es mayor.

Nota

La función de comparación sigue la misma convención que `cmp.Compare` de la biblioteca estándar. Para tipos nativos como `int` o `string` puede usarse directamente `cmp.Compare[T]`.

5.9.2.1 Implementación desde cero

Este enfoque consiste en definir un nodo binario que contenga el valor, el hijo izquierdo y el hijo derecho, y luego construir el ABB alrededor de ese nodo. Es la implementación más directa y no depende de ninguna otra estructura.

Go

```go
package binarysearchtree

// BinaryNode representa un nodo del árbol binario de búsqueda.
type BinaryNode[T any] struct {
    Value T
    Left  *BinaryNode[T]
    Right *BinaryNode[T]
}
```

Go

```go
package binarysearchtree

// BinarySearchTree representa un árbol binario de búsqueda.
type BinarySearchTree[T any] struct {
    Root *BinaryNode[T]
    cmp  func(T, T) int
}

// NewBinarySearchTree crea un ABB vacío con la función de comparación dada.
func NewBinarySearchTree[T any](cmp func(T, T) int) *BinarySearchTree[T] {
    return &BinarySearchTree[T]{cmp: cmp}
}
```

El árbol posee una raíz que apunta al primer nodo. Las operaciones de inserción, búsqueda y eliminación se implementan como métodos recursivos que navegan por los nodos comparando valores. La lógica de ordenamiento está intrínsecamente ligada a la estructura del nodo.

Ventajas:

- Control total sobre la representación interna de los datos.

- Sin dependencias externas.
- Ideal para aprendizaje y comprensión del algoritmo.

Desventajas:

- Duplica lógica estructural que podría reutilizarse (recorridos, altura, etc.).
- Acopla la validación de la propiedad de orden con la estructura del árbol.
- Menor abstracción: cualquier cambio en la estructura del nodo impacta directamente en las operaciones del ABB.

5.9.2.1.1 Delegación de operaciones

En la implementación anterior las operaciones se manejan desde el árbol (`bst.Insert`, `bst.Search`, etc.), que delega en métodos privados recursivos. Una alternativa es delegar la lógica en los propios nodos:

Go

```go
func (n *BinaryNode[T]) Insert(value T, cmp func(T, T) int) *BinaryNode[T] {
    if n == nil {
        return &BinaryNode[T]{Value: value}
    }
    if cmp(value, n.Value) < 0 {
        n.Left = n.Left.Insert(value, cmp)
    } else if cmp(value, n.Value) > 0 {
        n.Right = n.Right.Insert(value, cmp)
    }
    return n
}
```

Delegar en los nodos hace que el árbol actúe solo como un envoltorio (`bst.Root = bst.Root.Insert(value, bst.cmp)`). La función de comparación se pasa como parámetro para que el nodo pueda decidir hacia dónde navegar. Esto acerca la implementación al paradigma de objetos donde cada nodo es responsable de su propia manipulación, pero puede dificultar el control de errores y la trazabilidad.

5.9.2.2 Implementación por composición

El segundo enfoque consiste en construir el ABB a partir de una implementación genérica de árbol binario existente, como la que se encuentra en el paquete `tree` del repositorio `data-structures`. La idea es que el ABB **contenga** un árbol binario y le agregue la semántica de orden.

Go

```go
package binarysearchtree

import "github.com/untref-ayp2/data-structures/tree"

// BinarySearchTree representa un árbol binario de búsqueda
// construido sobre un árbol binario genérico.
type BinarySearchTree[T any] struct {
    binaryTree *tree.BinaryTree[T]
    cmp        func(T, T) int
```

```
}

// NewBinarySearchTree crea un ABB vacío con la función de comparación dada.
func NewBinarySearchTree[T any](cmp func(T, T) int) *BinarySearchTree[T] {
    return &BinarySearchTree[T]{
        binaryTree: tree.NewBinaryTree[T](),
        cmp:        cmp,
    }
}
```

En esta variante, el árbol binario subyacente provee las operaciones estructurales (insertar nodo, recorrer, calcular altura), mientras que el ABB se encarga únicamente de **dónde** insertar según el orden de los valores. La lógica de recorridos se hereda del árbol binario y no necesita reimplementarse.

Ventajas:

- Reutilización de código: los recorridos y la gestión estructural del árbol binario ya están implementados.
- Separación de responsabilidades: el árbol binario maneja *cómo* se almacenan los nodos; el ABB maneja *dónde* van según el orden.
- Menor cantidad de código que mantener y testear.
- Mayor abstracción: si cambia la implementación interna del árbol binario, el ABB no se ve afectado.

Desventajas:

- Dependencia de una biblioteca externa.
- Menor control sobre la representación interna.
- Curva de aprendizaje adicional si no se conoce la API del árbol binario subyacente.
- Puede introducir overhead si la interfaz genérica no se ajusta perfectamente a las necesidades del ABB.

5.9.2.3 Comparación

Aspecto	Desde cero	Por composición
Dependencias	Ninguna	`data-structures/tree`
Control interno	Total	Limitado por la API del árbol genérico
Cantidad de código	Mayor	Menor
Reutilización	Baja	Alta (recorridos, altura, etc.)
Curva de aprendizaje	Directa	Requiere conocer la API del árbol binario
Acoplamiento	Fuerte entre nodo y ABB	Débil: el ABB solo añade semántica de orden
Mantenimiento	Mayor (cada cambio impacta en todo)	Menor (los cambios se encapsulan en el árbol genérico)
Flexibilidad	Máxima	Limitada por la interfaz del árbol genérico

Ideal para	Aprendizaje y experimentación	Proyectos que priorizan la reutilización

Ejercicios

Los ejercicios de este capítulo están en `09-abb/ejercicios/` del repositorio taller-tad.

Antes de comenzar, asegurate de tener implementadas las estructuras necesarias en data-structures, que está dentro del repositorio `taller-tad`. Ambas tareas se trabajan en paralelo: primero completás las implementaciones en `data-structures` y después las usás acá.

5.10 Árboles Binarios de Búsqueda Balanceados

Una de las propiedades fundamentales de los árboles binarios de búsqueda es que preservan el orden de los elementos, y por lo tanto se pueden usar como contenedores de datos para implementar estructuras más complejas que requieran mantener el orden de los elementos, como por ejemplo diccionarios, listas ordenadas, etc.

La eficiencia de las operaciones de búsqueda, inserción y eliminación en un árbol binario de búsqueda depende en gran medida de su altura. En el peor de los casos, las operaciones pueden ser de orden lineal $O(n)$. Para evitar esto, se utilizan árboles binarios de búsqueda balanceados, que mantienen la altura del árbol en un nivel logarítmico $O(\log n)$ al garantizar que la diferencia entre las alturas de los subárboles izquierdo y derecho sea mínima.

Existen diferentes tipos de árboles binarios de búsqueda balanceados, como los árboles AVL y los árboles rojo y negro. Estos árboles implementan diferentes algoritmos de balanceo para garantizar que la altura del árbol se mantenga equilibrada después de cada operación de inserción o eliminación.

Árboles AVL

En 1962, dos científicos soviéticos, ***Georgy Adelson-Velsky*** y ***Evgenii Landis***, publicaron un artículo titulado ***"An algorithm for the organization of information"*** Adelson-Velsky & Landis (1962) (Un algoritmo para la organización de la información). En este trabajo, introdujeron la primera estructura de datos de árbol binario de búsqueda autobalanceado conocido como **árbol AVL**.

El nombre "AVL" proviene de las iniciales de sus inventores. Su principal innovación fue definir una condición de balanceo estricta que debía mantenerse en cada nodo del árbol:

> La diferencia en la altura entre los subárboles izquierdo y derecho de cualquier nodo no puede ser mayor que uno.

Formalmente:

Definición de Árbol AVL

$$\text{AVL si} \tag{5.7}$$

En esta definición aparece el concepto de **factor de balanceo** o **factor de equilibrio** de un nodo, que se define como la diferencia entre las alturas de los subárboles izquierdo y derecho:

$$fb(n) = h_{izq} - h_{der} \tag{5.8}$$

En la siguiente figura se muestra un árbol binario de búsqueda y los factores de balanceo de cada uno de sus nodos.

Si bien la raíz está balanceada, el árbol no es AVL ya que hay varios nodos que están desbalanceados. En este caso, el nodo 54 tiene un factor de balanceo de −2, lo que significa que su subárbol derecho es dos veces más alto que su subárbol izquierdo. Por lo tanto, el árbol no cumple con la condición de balanceo.

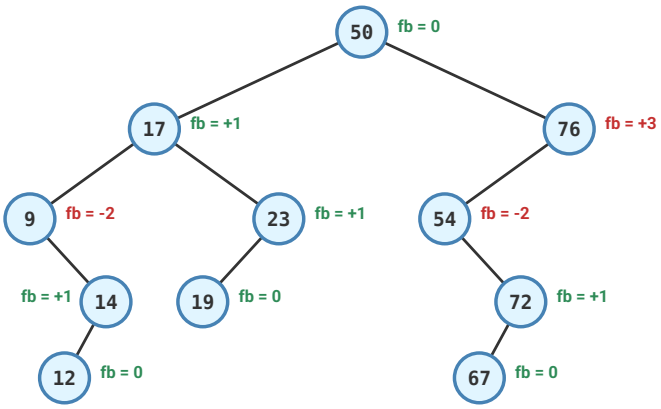


Figura 5.87: Factores de balanceo de cada nodo de un ABB

Los árboles AVL son árboles autobalanceados, es decir que se ajustan automáticamente después de cada operación de inserción o eliminación para mantener la propiedad de balanceo. Esto se logra mediante rotaciones, que son operaciones que cambian la estructura del árbol sin afectar el orden de los elementos. Las rotaciones son de orden $O(1)$ y por lo tanto no afectan la complejidad de las operaciones de búsqueda, inserción y eliminación, que siguen siendo $O(\log n)$.

Rotaciones

Las rotaciones pueden ser simples o dobles. Las rotaciones simples son suficientes para corregir la mayoría de los desbalances, pero en algunos casos se requieren rotaciones dobles.

Partimos del siguiente árbol AVL balanceado:

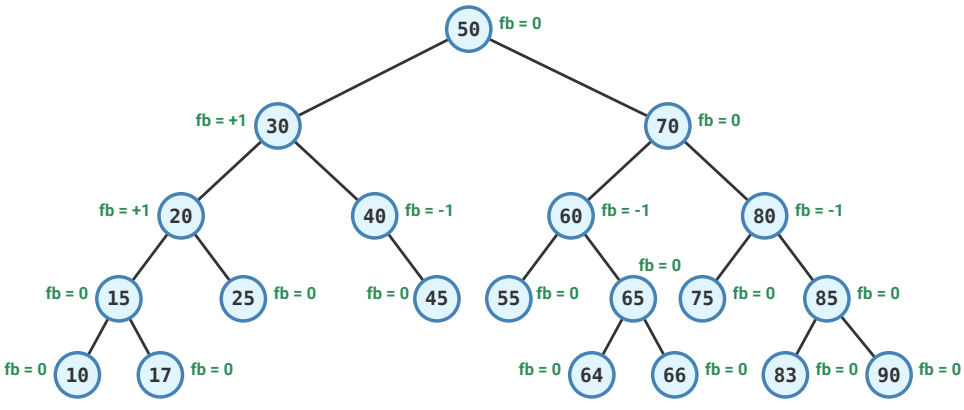


Figura 5.88: Árbol AVL balanceado

Sobre este árbol vamos a realizar cuatro inserciones, cada una en una rama distinta, para observar los cuatro tipos de desbalance y sus rotaciones correspondientes.

5.10.2.1 Rotación simple a derecha

Supongamos que en el árbol de la Figure 2 insertamos el (5) como hijo izquierdo del (10). Esto provoca un desbalance de tipo **Izquierda-Izquierda**. El primer nodo que queda desbalanceado en el camino de subida es el (20), ya que su subárbol izquierdo queda dos niveles más alto que el derecho ($fb = +2$). El nodo (15) se mantiene dentro del rango permitido ($fb = +1$).

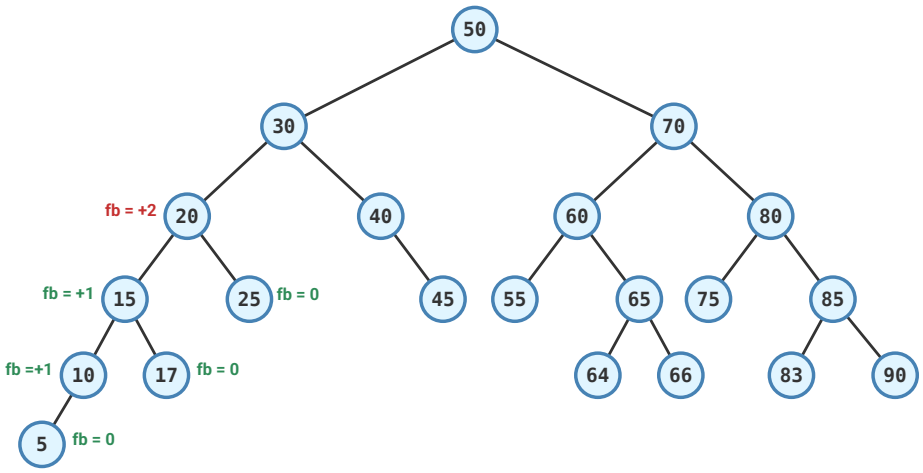


Figura 5.89: Desbalanceo Izquierda-Izquierda: fb(20) = +2

Para restaurar el equilibrio se debe realizar una **rotación simple a la derecha** del nodo (20). Los nodos involucrados son el (20), el (15), el (10) y el recién insertado (5). El (15) asciende a la posición del (20), y el (20) pasa a ser su hijo derecho. El nodo (17), que originalmente era hijo derecho del (15), pasa a ser hijo izquierdo del (20).

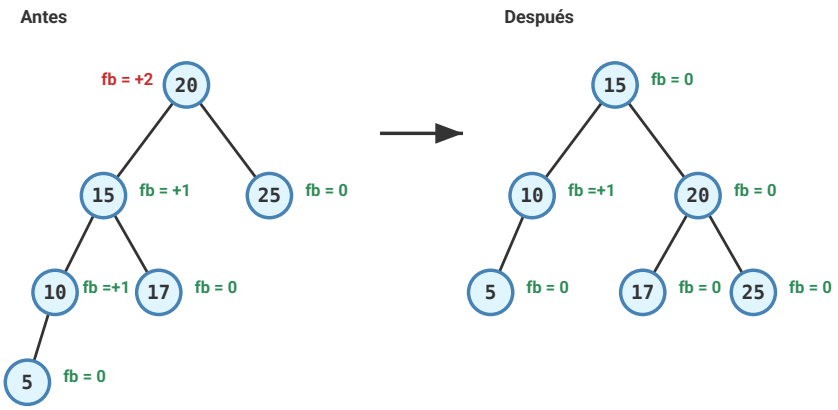


Figura 5.90: Rotación simple a derecha del nodo (20)

Como se ve en la figura a continuación, la rotación solo afecta a algunos pocos nodos del árbol, es decir, siempre son operaciones **locales**.

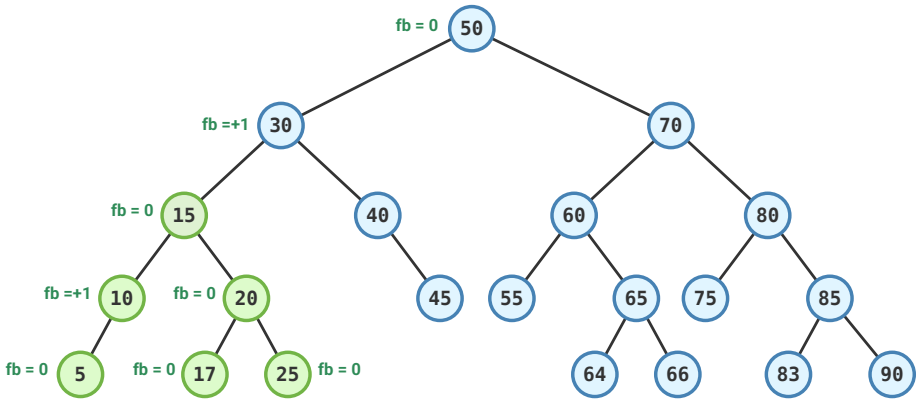


Figura 5.91: AVL restaurado luego de la rotación simple a derecha

5.10.2.2 Rotación simple a izquierda

Análogamente, cuando la inserción se produce a la derecha del hijo derecho de un nodo, el desbalance es **Derecha-Derecha** y se resuelve con una **rotación simple a la izquierda**.

Partiendo del árbol original, insertamos el (67) como hijo derecho del (66). Esto provoca un desbalance en el nodo (60), cuyo subárbol derecho queda dos niveles más alto que el izquierdo ($fb = -2$).

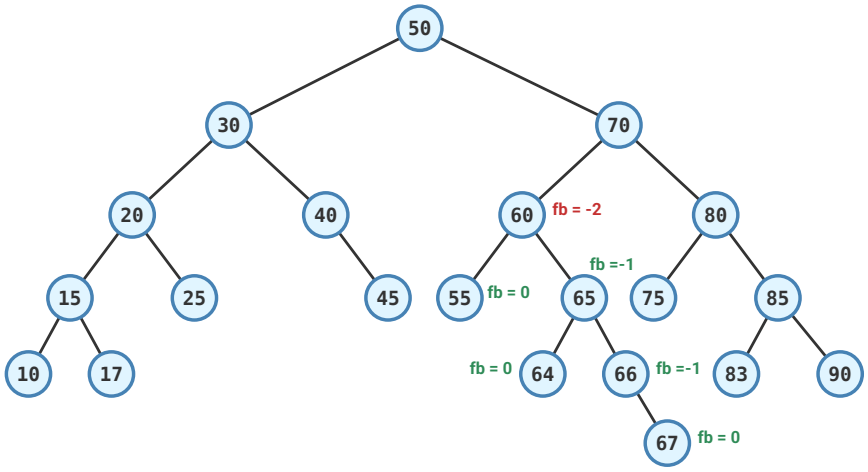


Figura 5.92: Desbalanceo Derecha-Derecha: fb(60) = −2

Para corregirlo se realiza una rotación simple a la izquierda del nodo (60). El (65) asciende y el (60) pasa a ser su hijo izquierdo. El (64), que originalmente era hijo izquierdo del (65), pasa a ser hijo derecho del (60).

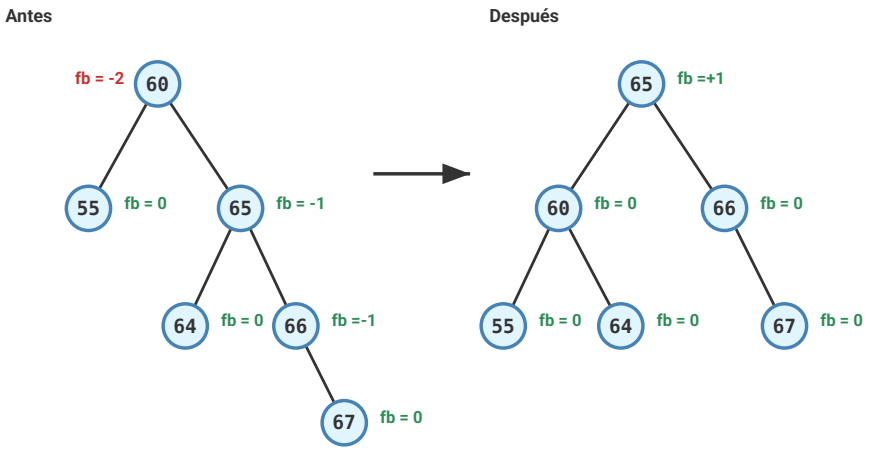


Figura 5.93: Rotación simple a izquierda del nodo (60)

Luego de la rotación el árbol vuelve a estar balanceado:

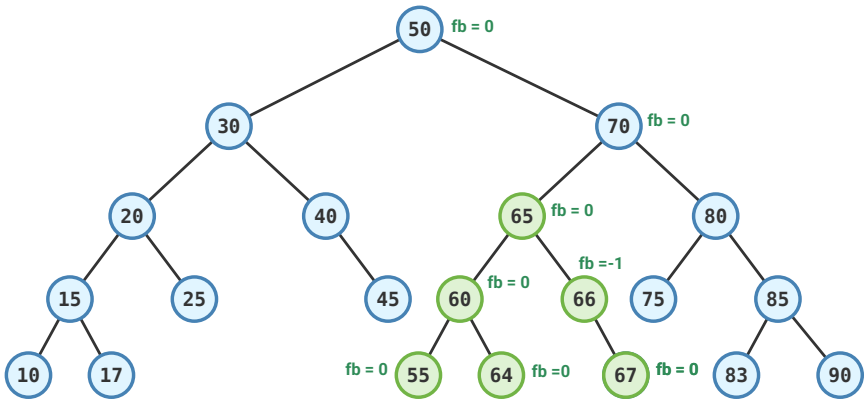


Figura 5.94: AVL restaurado luego de la rotación simple a izquierda

5.10.2.3 Rotación doble izquierda-derecha

Cuando el desbalance se produce al insertar un elemento a la *derecha* del hijo *izquierdo* de un nodo, una rotación simple no alcanza: se necesita una **rotación doble izquierda-derecha**. Esta rotación combina una rotación simple a la izquierda del hijo del nodo desbalanceado, seguida de una rotación simple a la derecha del nodo desbalanceado.

Partiendo del árbol original, insertamos el (18) como hijo derecho del (17). Esto provoca que el nodo (20) quede desbalanceado ($fb = +2$). El hijo izquierdo del (20) es el (15), que tiene $fb = -1$ (está cargado a la derecha), indicando que el desbalance es de tipo **Izquierda-Derecha**.

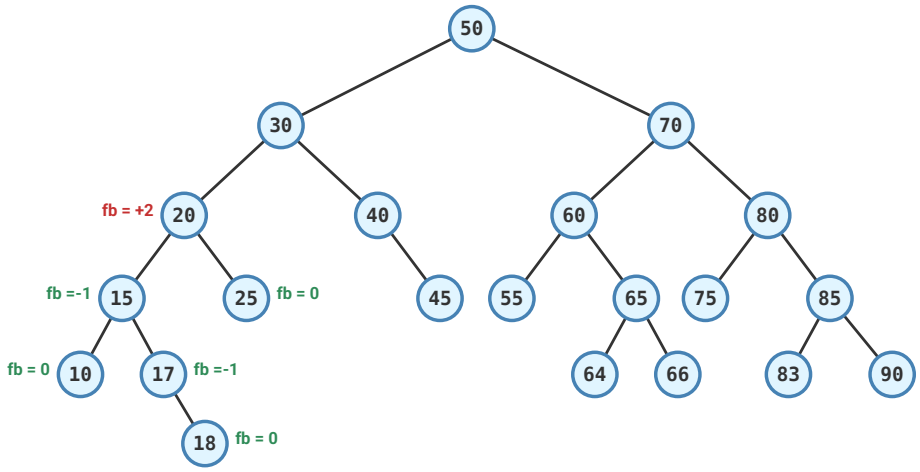


Figura 5.95: Desbalanceo Izquierda-Derecha: fb(20) = +2, fb(15) = −1

La solución consiste en dos pasos:

1. **Rotar a la izquierda** el hijo izquierdo del nodo desbalanceado, es decir el (15). Aunque el (15) está dentro del rango permitido, esta rotación alinea los nodos para que el desbalance pase a ser del tipo Izquierda-Izquierda.

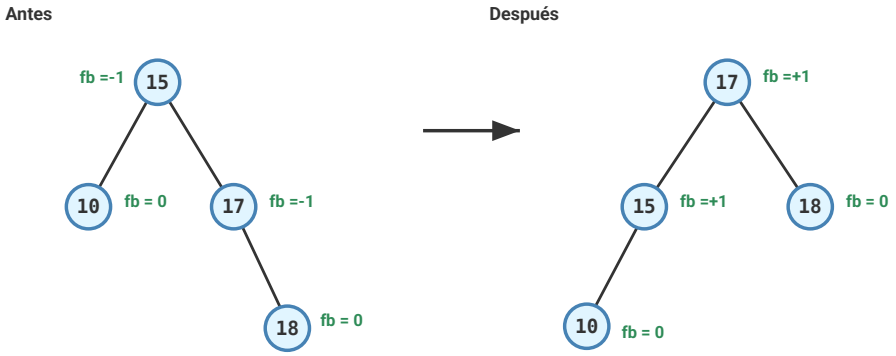


Figura 5.96: Rotación simple a izquierda del (15)

Luego de esta primera rotación, el árbol queda con el (17) en la posición que antes ocupaba el (15), y el desbalance ahora es de tipo Izquierda-Izquierda en el (20):

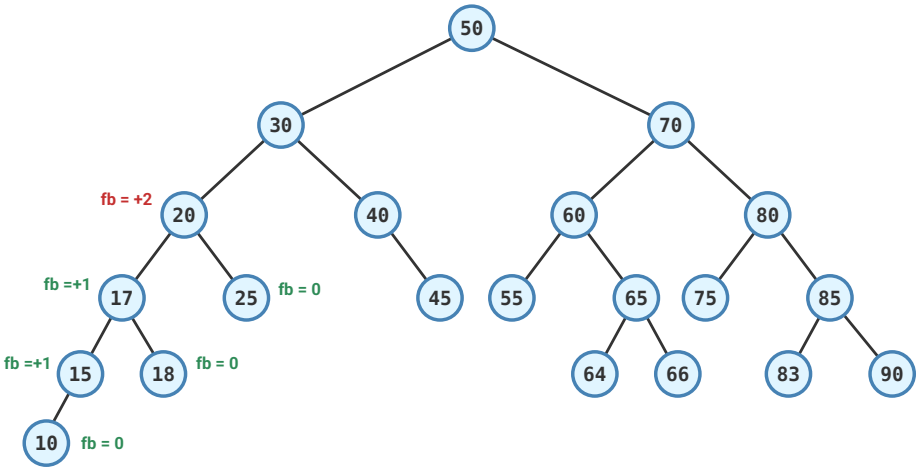


Figura 5.97: Árbol luego de la rotación a izquierda del (15): fb(20) = +2

2. **Rotar a la derecha** el nodo desbalanceado (20). El (17) asciende, el (20) pasa a ser su hijo derecho, y el subárbol derecho del (17) (que antes era el (18)) se reubica como hijo izquierdo del (20).

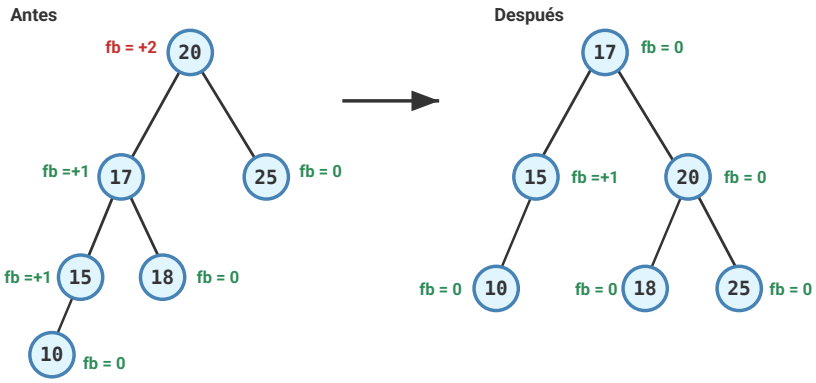


Figura 5.98: Rotación simple a derecha del (20)

Finalmente, el árbol recupera el equilibrio:

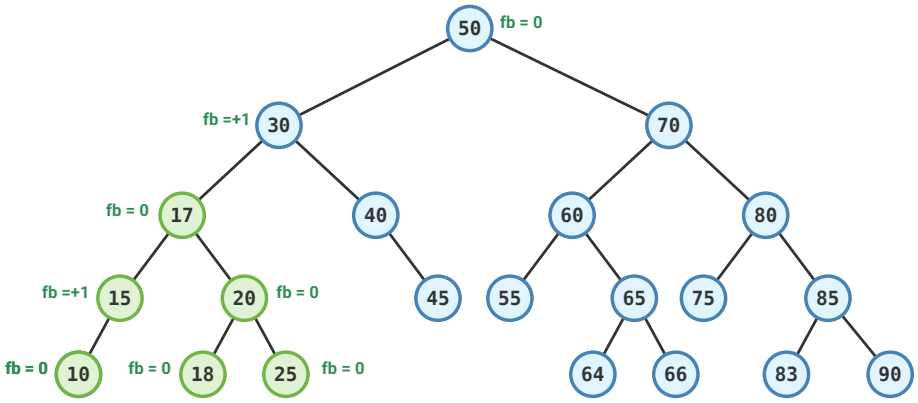


Figura 5.99: Árbol rebalanceado luego de la rotación doble izquierda-derecha

5.10.2.4 Rotación doble derecha-izquierda

El caso simétrico se da cuando la inserción ocurre a la *izquierda* del hijo *derecho* de un nodo. Se requiere una **rotación doble derecha-izquierda**.

Partiendo del árbol original, insertamos el (82) como hijo izquierdo del (83). Esto desbalancea el nodo (80) ($fb = -2$). Su hijo derecho es el (85), que tiene $fb = +1$ (cargado a la izquierda), indicando un desbalance **Derecha-Izquierda**.

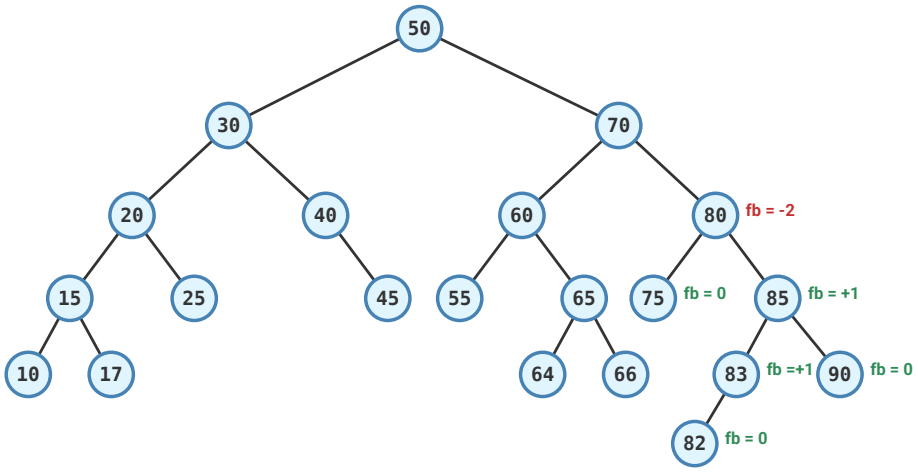


Figura 5.100: Desbalanceo Derecha-Izquierda: fb(80) = −2, fb(85) = +1

Para restablecer el balance se podría rotar el (85) a la derecha y luego el (80) a la izquierda, pero también puede pensarse a la rotación doble como un único movimiento: se “tira” del (83) hacia arriba para que sea la nueva raíz del subárbol.

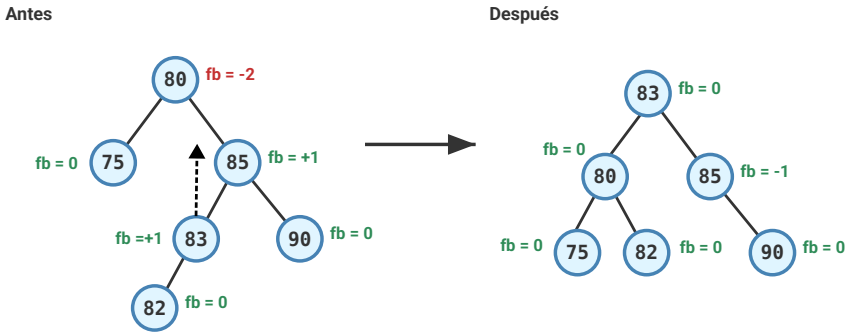


Figura 5.101: Rotación doble derecha-izquierda en un solo paso: el (83) asciende

Al ascender el (83):

- El (80), que antes era la raíz, pasa a ser su hijo izquierdo (80 < 83).
- El (85) que antes era el padre del (83) pasa a ser su hijo derecho (85 > 83).
- El subárbol izquierdo del (83), el (82), se reubica como hijo derecho del (80), porque 82 > 80.
- El subárbol izquierdo del (80), el (75), se mantiene como su hijo izquierdo.
- El subárbol derecho del (85), el (90), se mantiene como su hijo derecho.

A continuación se muestra el árbol balanceado luego de la rotación doble:

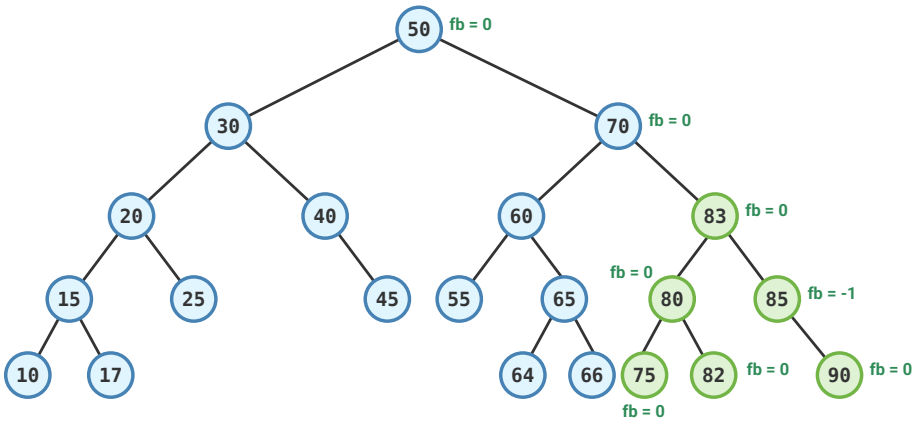


Figura 5.102: Árbol rebalanceado luego de la rotación doble derecha-izquierda

Importante

Las rotaciones son operaciones locales que solo afectan a un pequeño número de nodos. Cuando se inserta un nuevo nodo, se asciende por el camino desde la inserción hasta la raíz verificando el balanceo de cada nodo. Ante el primer nodo desbalanceado se realiza la rotación correspondiente, lo que restaura la altura del subárbol a su valor anterior a la inserción. Esto garantiza que los nodos superiores recuperen automáticamente el equilibrio.

Implementación

5.10.3.1 Estructura del nodo AVL

A diferencia de un ABB, donde los nodos solo almacenan valor y referencias a sus hijos, un nodo AVL necesita información adicional para determinar si el subárbol está balanceado. La estrategia más común es guardar la **altura** de cada nodo, y a partir de ella calcular el factor de balanceo como la diferencia entre las alturas de sus hijos derecho e izquierdo.

Go

```go
type AVLNode[T any] struct {
    value  T
    left   *AVLNode[T]
    right  *AVLNode[T]
    height int
}
```

El factor de balanceo se define como:

text

```
fb(n) = altura(hijo_izquierdo) - altura(hijo_derecho)
```

Cuando |fb(n)| > 1, el nodo está desbalanceado y se debe aplicar la rotación correspondiente.

5.10.3.2 Actualización de alturas

Cada vez que se modifica la estructura del subárbol (inserción o eliminación), se debe actualizar la altura del nodo. La altura de un nodo vacío (`nil`) es −1, y la de un nodo hoja recién creado es 0, consistente con la definición de altura del capítulo Árboles.

text

```
altura(n) = 1 + max(altura(hijo_izquierdo), altura(hijo_derecho))
```

Esta actualización debe ocurrir después de cada operación recursiva y después de cada rotación, ya que las rotaciones modifican la estructura del subárbol y por lo tanto las alturas de los nodos involucrados.

5.10.3.3 Rotaciones

Las rotaciones simple derecha y simple izquierda son operaciones locales que reestructuran el subárbol manteniendo la propiedad de BST. Ambas deben actualizar las alturas de los nodos que modifican.

Una rotación doble (derecha-izquierda o izquierda-derecha) puede implementarse como la composición de dos rotaciones simples, o como un único movimiento como se explicó en la sección anterior.

5.10.3.4 Estrategia de inserción y eliminación

Tanto la inserción como la eliminación siguen el mismo patrón recursivo:

1. Realizar la operación como en un ABB (buscar la posición, insertar o eliminar).
2. Actualizar la altura del nodo actual.
3. Calcular el factor de balanceo.
4. Si está desbalanceado, aplicar la rotación correspondiente.

La estructura del árbol AVL es similar a la del ABB:

Go

```go
type AVLTree[T any] struct {
    root *AVLNode[T]
    size int
    cmp  func(T, T) int
}
```

La función de comparación `cmp` permite trabajar con cualquier tipo de dato. Los métodos públicos delegan en funciones auxiliares recursivas que recorren el árbol desde la raíz.

Ejercicios

Implementar un árbol AVL en `data-structures` (dentro del repositorio `taller-tad`), paquete `avltree/`. El árbol debe mantener la altura en cada nodo y rebalancearse automáticamente después de cada inserción y eliminación mediante rotaciones simples y dobles.

La implementación de la eliminación ya está completa; las rotaciones (`rotateRight`, `rotateLeft`) y la inserción quedan a cargo del estudiante.

5.11 Colas de Prioridad y Montículos Binarios

Colas de Prioridad

Una cola de prioridad se comporta de forma similar a una cola, pero cada elemento que ingresa tiene asignada una prioridad y los elementos con mayor prioridad son los primeros en salir de la cola.

Si todos los elementos tienen la misma prioridad, la cola se comporta como una cola común, es decir, el primero que llega es el primero que sale de la cola.

Por ejemplo, si se tiene una cola de prioridad con los siguientes elementos y prioridades, y la posición en la tabla corresponde al orden de llegada de los elementos, es decir, se insertaron en orden alfabético y además suponemos que la prioridad 1 es la mayor, luego la 2 y así sucesivamente.

Elemento	Prioridad
A	3
B	1
C	2
D	1
E	3
F	2
G	1
H	3
I	2

Table 5.103: Cola de Prioridad

La cola de prioridad se comportará de la siguiente forma:

1. Se extrae el elemento B
2. Se extrae el elemento D
3. Se extrae el elemento G
4. Se extrae el elemento C
5. Se extrae el elemento F
6. Se extrae el elemento I
7. Se extrae el elemento A
8. Se extrae el elemento E
9. Se extrae el elemento H

En este ejemplo se puede observar que el orden de extracción combina la prioridad con el orden de llegada: a igual prioridad, los elementos se extraen en el mismo orden en que fueron insertados.

Las operaciones que definen una cola de prioridad son:

- `Enqueue`: agregar un nuevo elemento con una prioridad asociada.
- `Dequeue`: eliminar y devolver el elemento con mayor prioridad.
- `Peek`: obtener el elemento con mayor prioridad sin eliminarlo.
- `IsEmpty`: consultar si la cola está vacía.
- `Size`: obtener la cantidad de elementos.

5.11.1.1 Interfaz

En Go, definimos una interfaz que capture este comportamiento:

Go

```go
type PriorityQueue[T any] interface {
  // Enqueue agrega un nuevo elemento a la cola.
  Enqueue(element T)
  // Dequeue elimina y devuelve el elemento con mayor prioridad.
  // Error si la cola está vacía.
  Dequeue() (T, error)
  // Peek devuelve el elemento con mayor prioridad sin eliminarlo.
  // Error si la cola está vacía.
  Peek() (T, error)
  // IsEmpty devuelve true si la cola no tiene elementos.
  IsEmpty() bool
  // Size devuelve la cantidad de elementos en la cola.
  Size() int
}
```

5.11.1.2 Estrategia de comparación

Para determinar qué elemento tiene mayor prioridad, la cola necesita una forma de comparar elementos. Go no tiene una interfaz `Ordered` nativa como otros lenguajes, así que la estrategia habitual es pasar una **función de comparación** al constructor de la implementación.

La convención es usar una función del tipo `func(a, b T) int` que devuelva:

- un número **negativo** si `a` tiene prioridad sobre `b`,
- **cero** si son equivalentes,
- un número **positivo** si `b` tiene prioridad sobre `a`.

Por ejemplo, para enteros con orden ascendente (menor valor, mayor prioridad):

Go

```go
compare := func(a, b int) int {
    return a - b
}
```

Para orden descendente (mayor valor, mayor prioridad):

Go

```
compare := func(a, b int) int {
    return b - a
}
```

Para un `struct` con un campo `edad`:

Go

```
compare := func(a, b Persona) int {
    return a.edad - b.edad
}
```

Esta misma estrategia se utiliza en la biblioteca estándar de Go (`sort.Slice` recibe una función `less` similar, y `container/heap` requiere implementar `Less`).

Para implementar una cola de prioridad se puede usar un **montículo binario**, ya que optimiza las operaciones de inserción y eliminación de los elementos.

Montículos Binarios

Los montículos o *heaps* en inglés son estructuras de datos que permiten acceder al elemento que se encuentra en la "cima" muy eficientemente. Esta posición privilegiada se utiliza para mantener el mayor elemento del montículo o el menor. En el caso de que se mantenga el mayor elemento en la cima se le llama **Montículo de Máximos** y si se mantiene el menor se le llama **Montículo de Mínimos**.

Existen diferentes tipos de montículos (binario, fibonacci, suave (*soft*), etc.), pero todos comparten la propiedad de que el elemento que se encuentra en la cima es el mayor o el menor de toda la estructura. En nuestro curso nos enfocaremos en los montículos binarios.

Figura 5.104: Pirámide de Cajas

Un **Montículo Binario** consiste en un árbol binario que cumple con dos propiedades fundamentales.

Propiedad de Forma Es un árbol binario **completo** (o casi completo) e **izquierdista**, es decir que se va llenando de izquierda a derecha. Lo que implica que todos los niveles del árbol están llenos, excepto el último nivel que puede estar incompleto.

Propiedad de Orden En un montículo de mínimos, cada nodo es menor que todos sus descendientes. Análogamente si es un montículo de máximos, cada nodo es mayor que todos sus descendientes. En otras palabras, en un montículo binario de mínimos, el menor elemento de toda la estructura se encuentra en la raíz y recursivamente podemos pensar a los hijos de la raíz como otros montículos de mínimos

En la siguiente figura se puede ver un heap de mínimos donde se cumplen ambas propiedades. El elemento 10 es el menor de toda la estructura y se encuentra en la raíz. Análogamente el menor de sus descendientes por la izquierda es el elemento 20 y se encuentra en la raíz del subárbol izquierdo y por el lado derecho el elemento 30 es el menor del subárbol derecho.

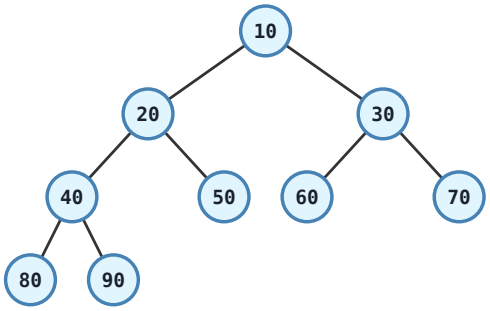


Figura 5.105: Montículo Binario de Mínimos

5.11.2.1 Representación

Como un montículo binario es un árbol binario completo, o casi completo, podemos aprovechar esta propiedad y usar un arreglo para representarlo. Usar un arreglo para representar un árbol binario completo es una técnica muy común en computación y se conoce como **representación por niveles** y tiene ventajas en términos de espacio y tiempo y facilita la implementación de las operaciones.

> **Note**
>
> Cómo es un árbol **completo e izquierdista** entonces el arreglo que se usa como contenedor de datos tendrá elementos en todas sus posiciones, es decir, no habrá "huecos" y la raíz siempre se encontrará en la posición 0. Además con algunas cuentas simples se puede determinar donde están los hijos o el padre de cualquier nodo.

En la siguiente figura se observa como se puede usar un arreglo para mantener un montículo binario. La raíz se encuentra en la posición 0, el hijo izquierdo de la raíz en la posición 1, el hijo derecho de la raíz en la posición 2 y así sucesivamente.

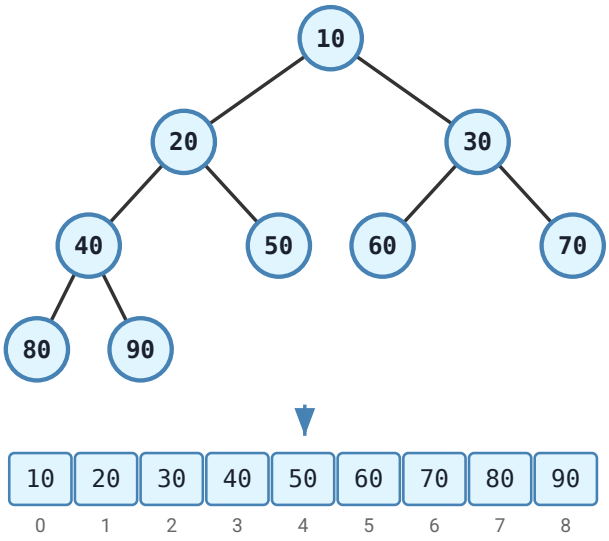


Figura 5.106: Representación con arreglos de un Montículo Binario de Mínimos

Con las siguientes fórmulas se puede calcular la posición en el arreglo de cualquier hijo o padre de un nodo dado.

Fórmulas

$H_{izq}(i) = 2\,i + 1$

$H_{der}(i) = 2\,i + 2$

$Padre(i) = floor(\frac{i-1}{2})$

Donde el símbolo $floor(\ )$ indica el piso, es decir la parte entera del cociente.

Por ejemplo en la posición 3 del arreglo se encuentra el elemento 40, su hijo izquierdo se encuentra en la posición $2 \cdot 3 + 1 = 7$ y su hijo derecho en $2 \cdot 3 + 2 = 8$. A su vez el padre se encuentra en $floor(\frac{3-1}{2}) = 1$.

5.11.2.2 Operaciones

Las operaciones básicas que se pueden realizar en un montículo son:

`Top` Devuelve el elemento que se encuentra en la cima del montículo, es decir el máximo o el mínimo de toda la estructura, pero no lo elimina.

`Size` Devuelve la cantidad de elementos que se encuentran en el montículo.

`Insert` Inserta un nuevo elemento en el montículo.

`Remove` Elimina el elemento que se encuentra en la cima del montículo y lo devuelve.

5.11.2.2.1 Insertar

Para preservar la propiedad de forma, se inserta el nuevo elemento al final del arreglo (como última hoja del árbol), manteniendo así la estructura de árbol completo e izquierdista.

Al insertar un elemento en la última posición, se pierde la propiedad de orden, ya que el nuevo elemento podría ser mayor (en un heap de máximos) que su padre. Para restaurarla se ejecuta `upHeap` desde la hoja recién insertada. Esta operación compara el elemento con su padre y, si no se cumple la propiedad de orden, los intercambia, repitiendo el proceso hacia arriba hasta que el elemento encuentra su lugar.

text

```
FUNCION Insertar(elemento)
    pos ← n                 // n es la cantidad actual de elementos
    A[pos] ← elemento
    n ← n + 1
    upHeap(pos)
FIN FUNCION
```

Algoritmo de inserción en un montículo de máximos

Con la siguiente función auxiliar se restablece la propiedad de orden, intercambiando el elemento con su padre hasta que cumple la condición del montículo. El padre de la posición i se calcula como $floor((i-1)/2)$.

text

```
FUNCION upHeap(i)
    MIENTRAS i > 0 Y A[i] > A[padre(i)] HACER
        intercambiar(A[i], A[padre(i)])
        i ← padre(i)
    FIN MIENTRAS
FIN FUNCION
```

Algoritmo upHeap

`upHeap` recorre en el peor caso toda la altura del árbol. Como el montículo binario es un árbol completo, su altura es $O(\log n)$, por lo tanto **Insertar** tiene orden $O(\log n)$.

5.11.2.2.2 Remover

La cima del montículo (el máximo o el mínimo) se encuentra siempre en la posición 0 del arreglo, por lo que consultarla cuesta $O(1)$.

Para eliminar la cima se debe preservar la propiedad de forma: se reemplaza la raíz por la última hoja del árbol (decrementando el tamaño) y luego se ejecuta `downHeap` desde la raíz para restaurar la propiedad de orden. `downHeap` compara el nodo actual con sus dos hijos; en un heap de máximos, si el nodo es menor que alguno de sus hijos, se intercambia con el **mayor** de ellos, repitiendo el proceso hacia abajo hasta encontrar la posición correcta.

text

```
FUNCION Eliminar()
    cima ← A[0]             // guardar la raíz
    n ← n - 1
    A[0] ← A[n]             // mover la última hoja a la raíz
    downHeap(0)
    RETORNAR cima
FIN FUNCION
```

Algoritmo de eliminación de la cima de un montículo de máximos

La siguiente función auxiliar busca entre los dos hijos del nodo actual cuál es el mayor. Si el nodo actual es menor que ese hijo, los intercambia y continúa hacia abajo.

text

```
FUNCION downHeap(i)
    MIENTRAS verdadero HACER
        candidato ← i
        izq ← 2 * i + 1
        der ← 2 * i + 2
        SI izq < n Y A[izq] > A[candidato] ENTONCES
            candidato ← izq
        FIN SI
        SI der < n Y A[der] > A[candidato] ENTONCES
            candidato ← der
        FIN SI
        SI candidato = i ENTONCES
            SALIR
        FIN SI
        intercambiar(A[i], A[candidato])
        i ← candidato
    FIN MIENTRAS
FIN FUNCION
```

Algoritmo downHeap

`downHeap` también recorre en el peor caso la altura del árbol, $O(\log n)$, por lo tanto **Remover** tiene orden $O(\log n)$.

5.11.2.2.3 Orden de las operaciones

Operación	Orden
`Top`	$O(1)$
`Size`	$O(1)$
`Insert`	$O(\log n)$
`Remove`	$O(\log n)$

Table 5.111: Orden de las Operaciones

Top y **Size** son $O(1)$ porque solo acceden a la posición 0 del arreglo o devuelven el tamaño almacenado, sin recorrer la estructura. **Insertar** y **Remover** son $O(\log n)$ porque en el peor caso deben recorrer la altura del árbol (que siempre es $O(\log n)$ por tratarse de un árbol completo) para reestablecer la propiedad de orden mediante `upHeap` o `downHeap`.

5.11.2.3 Visualizador de montículo binario interactivo

Seleccionar el tipo de montículo (mínimo o máximo), ingresar uno o más enteros separados por coma y presionar Enter para encolarlos. Usar **>** para procesar una operación, **>>** para procesar todas las pendientes, o **Remover cima** para eliminar la raíz. Las teclas **<** y **>** navegan entre pasos, **<<** deshace la última inserción, y Espacio inicia/pausa la reproducción automática. Solo se admiten enteros de hasta 3 cifras ($\pm$999).

Ejercicios

Los ejercicios de este capítulo se encuentran en el repositorio taller-tad, que incluye el módulo data-structures.

5.11.3.1 En data-structures

1. **Implementar un montículo binario genérico** (`heap/`). Crear las implementaciones `SliceHeap[T]` con constructores `NewMinHeap` y `NewMaxHeap` que reciben una función de comparación `func(T, T) int`. El montículo debe soportar las operaciones `Insert`, `Remove`, `Top`, `Size` e `IsEmpty`.
2. **Implementar una cola de prioridad genérica** (`priorityqueue/`). Crear `PriorityQueue[T]` que recibe un `Heap[T]` por constructor (inyección de dependencia). Operaciones: `Enqueue`, `Dequeue`, `Front`, `Size`, `IsEmpty`.

5.11.3.2 En taller-tad

Los ejercicios de aplicación están en `10-monticulo-binario/ejercicios/`:

- **01-merge-listas**: fusionar K listas ordenadas en una sola usando una cola de prioridad.
- **02-triage**: sistema de atención hospitalaria que ordena pacientes por gravedad y orden de llegada.

6. Diseño de Algoritmos

6.1 Recursividad y División y Conquista

Recursividad

En programación, la recursividad es una técnica donde **una función se llama a sí misma** dentro de su propia definición. Es como si la función se estuviera "descomponiendo" en versiones más pequeñas del mismo problema, hasta llegar a un **caso tan simple que se puede resolver directamente**.

Figura 6.1: Matryoshka.

Para que la recursión funcione, hay que prestar atención a dos aspectos importantes en el diseño de la función recursiva:

Caso base Es el caso más simple y trivial que se puede resolver directamente sin necesidad de llamarse a sí misma nuevamente. La recursión debe llegar a este caso para poder terminar. Si no hay un caso base que corte la recursión, la función se llamará a sí misma indefinidamente y se producirá un error de desbordamiento de pila (*stack overflow*).

Caso recursivo Es el punto de la función donde se vuelve a llamar a sí misma, pero siempre con una versión más pequeña del problema original. La recursión debe acercarse al caso base en cada llamada recursiva, si no se producirá un error de desbordamiento de pila.

Como ejemplo podemos escribir una función recursiva para calcular el factorial de un número. Recordemos que el factorial de un número entero positivo `n` se define como:

$$f(n) = \tag{6.1}$$

La definición de factorial es claramente recursiva y se puede implementar en Go de la siguiente manera:

Go

```go

package main

import "fmt"

func factorial(n int) int {
    if n == 0 { // Caso base
        return 1
    }
    return n * factorial(n-1) // Caso recursivo
}

func main() {
    fmt.Println(factorial(4))
}
```

output

```
24
```

En la línea 6 se define el caso base, donde el factorial de 0 es 1. En la línea 9 se encuentra la llamada recursiva, donde el factorial de `n` es `n` multiplicado por el factorial de `n-1`. La función se llama a sí misma con un valor más pequeño.

Para entender como funciona la recursión tenemos que observar la pila de ejecución. Si ejecutamos el código anterior, la pila de ejecución se verá de la siguiente manera:

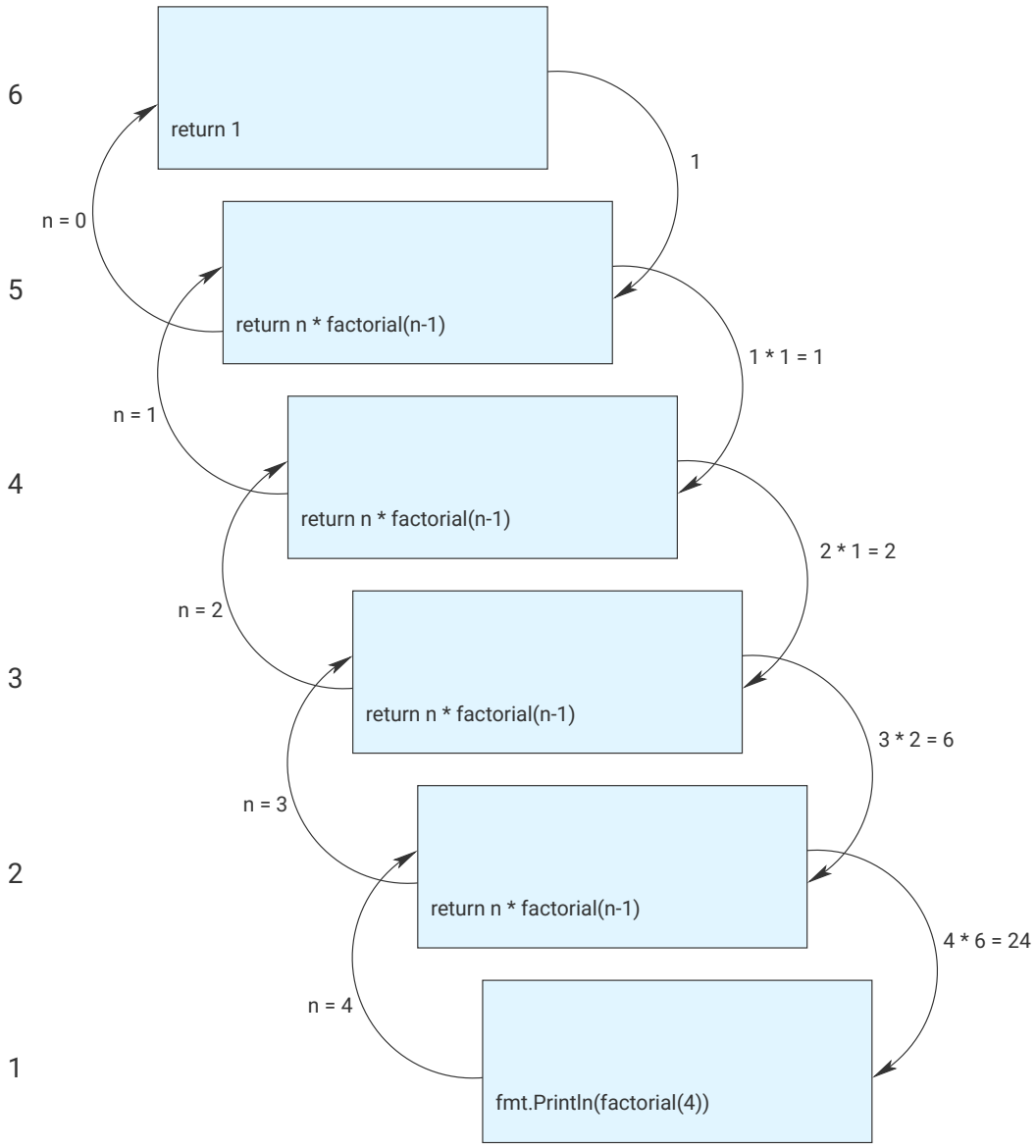


Figura 6.2: Pila de Ejecución - factorial(4).

La primera llamada que se hace a factorial recibe el valor de `n = 4`. Como `n != 0`, se llama a la función factorial con `n = 3`. Luego se llama a la función con `n = 2`, `n = 1` y finalmente `n = 0`. Cuando la función se llama a sí misma queda bloqueada su ejecución en ese punto, esperando el resultado de la llamada recursiva para continuar. Cuando se llega al caso base y se retorna `1` se retoma la ejecución de la llamada anterior y sucesivamente se van desapilando las llamadas recursivas y se van multiplicando los valores de `n` en cada nivel de la pila. Finalmente se obtiene el resultado de `24`.

6.1.1.1 Tipos de recursión

La recursividad, de acuerdo a como las funciones se llaman a sí mismas, se puede clasificar en dos tipos:

Recursión directa Es la forma más común de recursión, donde una función se llama a sí misma directamente.

Recursión indirecta En este caso, dos o más funciones se llaman entre sí de forma cíclica. Es decir, la función A llama a la función B, que a su vez llama a la función A. Este tipo de

recursión es menos común y puede ser difícil de entender y depurar. Incluso puede haber más de dos funciones que se llaman mutuamente.

El ejemplo del cálculo del factorial es un ejemplo de recursión directa, ya que la función `factorial` se llama a sí misma directamente.

A continuación se muestra un ejemplo de recursión indirecta, donde dos funciones se llaman entre sí de forma cíclica. Las funciones son capaces de determinar si el entero dado es par o impar, sin usar divisiones, resto de la división o multiplicaciones. Simplemente llegan hasta el caso base cuando `n == 0` y retornan el valor correspondiente. (Es un ejemplo didáctico, no es la forma más eficiente de determinar si un número es par o impar)

Go

```go
func esPar(n int) bool {
    if n == 0 { // El 0 es par
        return true
    }
    return esImpar(n - 1)
}

func esImpar(n int) bool {
    if n == 0 { // El 0 no es impar
        return false
    }
    return esPar(n - 1)
}
```

En este caso en la línea 5 se llama a la función `esImpar` y en la línea 12 a la función `esPar`. Cada llamada recursiva se realiza siempre con un número menor, de forma tal de acercarse al caso base cuando `n == 0`. De acuerdo si llego al caso base por la primera o la segunda función el valor de retorno será `true` o `false`.

6.1.1.2 Cálculo de la complejidad de las funciones recursivas

Para calcular el orden de las funciones recursivas se debe escribir la ecuación de recurrencia que describe el tiempo de ejecución de la función. En el caso de factorial por ejemplo tenemos que:

$$T(n) = T(n-1) + O(k) \tag{6.2}$$

Donde $T(n)$ es el tiempo de ejecución de la función factorial para un valor n. La función `factorial` se llama a sí misma con un valor más pequeño, $n-1$, por lo que aparece el término $T(n-1)$, el resto de las operaciones son constantes y se representant con el término $O(k)$.

Aplicando la ecuación de recurrencia podemos calcular $T(n-1)$

$$T(n-1) = T(n-2) + O(k) \tag{6.3}$$

Si sustituimos la ecuación anterior en la ecuación original obtenemos:

$$T(n) = T(n-2) + 2\ O(k) \tag{6.4}$$

Por lo tanto, en el paso i-ésimo tendremos que:

$$T(n) = T(n-i) + i\ O(k) \tag{6.5}$$

Finalmente en la última llamada tendremos:

$$T(n) = T(0) + n\ O(k) \tag{6.6}$$

como $T(0) = O(1)$, porque no se realiza ninguna llamada recursiva, sino que se resuelve por el caso base, entonces la complejidad de la función factorial es:

$$T(n) = n\ O(k) + O(1) = O(n) \tag{6.7}$$

División y Conquista

División y conquista, también conocida como divide y vencerás, es una técnica de diseño de algoritmos, muy vinculada a la recursividad, que se utiliza para resolver problemas que generalmente manipulan una gran cantidad de datos y que consiste aplicar tres pasos:

Dividir El problema original se divide en subproblemas más pequeños y manejables. Los subproblemas deben ser de la misma naturaleza que el problema original, pero de menor tamaño. Todos los subproblemas deben tener aproximadamente el mismo tamaño.

Conquistar Los subproblemas se resuelven por separados, generalmente de forma recursiva.

Combinar Las soluciones de los subproblemas deben combinarse para obtener la solución del problema original.

No todos los problemas se pueden resolver de esta manera, pero aquellos que sí, suelen ser más fáciles de entender y de implementar.

A modo de ejemplo vamos a implementar el algoritmo de búsqueda binaria, que es un algoritmo de búsqueda eficiente para encontrar un elemento específico en un arreglo ordenado. La búsqueda binaria divide el arreglo en mitades y compara el valor buscado con el elemento en el índice medio del arreglo. Si el valor buscado es menor que el elemento en el medio, la búsqueda se realiza en la mitad izquierda del arreglo. Si el valor buscado es mayor, la búsqueda se realiza en la mitad derecha del arreglo. Este proceso se repite hasta que el valor buscado se encuentre o el rango de búsqueda se reduzca a cero.

Importante

Las partes en la que se subdivide la entrada de datos original deben ser aproximadamente iguales. En el caso de la búsqueda binaria, si el tamaño del arreglo original era impar, entonces una mitad tendrá un elemento más que la otra, lo cual no afecta a la aplicación de esta técnica.

Esta técnica permite reducir drásticamente el tiempo de ejecución, es decir el orden de la función.

Go

```go
func busquedaBinaria(array []int, inicio int, fin int, x int) int {
    if inicio > fin {
        return -1
    }

    medio := (inicio + fin) / 2
```

```
    if array[medio] > x {
        return busquedaBinaria(array, inicio, medio-1, x)
    }

    if array[medio] < x {
        return busquedaBinaria(array, medio+1, fin, x)
    }

    return medio
}
```

Dividir En la linea 6 se calcula el medio que parte al arreglo de tamaño n en dos mitades de aproximadamente el mismo tamaño.

Conquistar Si el valor buscado se encuentra en la mitad inferior se realiza la llamada recursiva de la línea 9, en cambio si es mayor se realiza la llamada recursiva de la línea 13. Se realiza sólo una de las llamadas, no ambas.

Combinar En este ejemplo donde se busca un valor, la construcción del resultado final es trivial, ya que si en alguna de las llamadas recursivas se encuentra el elemento buscado, se devuelve el índice que se propaga debido a los `return` que acompañan a las llamadas recursivas en las líneas 9 y 13.

Caso base Como toda función recursiva debe tener un caso base para impedir que la recursión se desborde. En la búsqueda binaria podemos pensar el caso base como aquel en el que no se encuentra el número buscado y se reduce a 0 el espacio de búsqueda. Se puede ver en la línea 2 cuando `inicio > fin`.

6.1.2.1 Teorema maestro

El cálculo del orden de las funciones recursivas, en particular de las funciones que aplican la técnica de división y conquista, puede ser complicado de realizar. Por suerte el **Teorema maestro** permite agilizar el cálculo a partir de la siguiente observación:

La ecuación de recurrencia de una función recursiva que aplica la técnica de división y conquista será:

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^c) \tag{6.8}$$

Donde

- b Cantidad de partes iguales en la que se subdivide el problema original.
- a Cuantos de los subproblemas se resuelven efectivamente.
- c $O(n^c)$ es el costo de partir o subdividir el problema original más el costo de combinar las soluciones parciales.

El Teorema maestro establece que la solución a la ecuación de recurrencia es:

$$T(n) = \tag{6.9}$$

En el caso de la búsqueda binaria tenemos que:

$$\begin{aligned} a &= 1 \\ b &= 2 \\ c &= 0 \end{aligned} \tag{6.10}$$

por lo tanto, como $\log_{21} = 0$ estamos en el segundo caso

$$T(n) = O(\log_2 n) \tag{6.11}$$

Ejercicios

Los ejercicios de este capítulo están en el directorio `01-recursividad/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.2 Patrones de Diseño

Los patrones de diseño son soluciones reutilizables para problemas comunes que surgen en el desarrollo de software. Estos patrones encapsulan buenas prácticas y ofrecen un enfoque probado para resolver problemas recurrentes, facilitando el diseño de sistemas más robustos y mantenibles Gamma (2002).

Los patrones no son exclusivos de las ciencias informáticas, sino que también se encuentran en otras disciplinas como la arquitectura, el diseño industrial y la ingeniería. En cada caso, los patrones representan soluciones probadas que se pueden adaptar a diferentes contextos, manteniendo su esencia y efectividad.

Por ejemplo, en arquitectura, un patrón podría ser el diseño de una plaza central en una ciudad, que fomenta la interacción social y el flujo de personas. De manera similar, en el desarrollo de software, los patrones de diseño buscan resolver problemas recurrentes de manera eficiente, promoviendo la reutilización y la estandarización.

Es importante destacar que los patrones no son recetas estrictas, sino guías flexibles que deben ser adaptadas según las necesidades específicas del proyecto. Comprender el contexto y los requisitos es fundamental para aplicar un patrón de manera efectiva y evitar un uso inadecuado que pueda complicar el diseño en lugar de simplificarlo.

> "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe la esencia de la solución de ese problema, de tal manera en que se puede utilizar esta solución más de un millón de veces sin hacerlo igual siquiera dos veces"
>
> — Christopher Alexander

=== Características principales de los patrones de diseño

Reutilizabilidad Los patrones permiten aplicar soluciones existentes a nuevos problemas, ahorrando tiempo y esfuerzo.

Flexibilidad Se pueden personalizar para adaptarse a las necesidades específicas de un proyecto o contexto.

Comunicación Proveen un lenguaje común entre desarrolladores, facilitando la colaboración y el entendimiento del diseño.

Clasificación de los patrones de diseño

Los patrones de diseño se dividen en tres categorías principales:

Patrones creacionales Se centran en la creación de objetos, asegurando que el sistema sea independiente de cómo se crean, componen y representan los objetos. Ejemplos: :

- Singleton
- Factory Method
- Abstract Factory

Patrones estructurales Se ocupan de la composición de clases y objetos para formar estructuras más grandes. Ejemplos: :

- Adapter
- Composite
- Decorator

Patrones de comportamiento Se enfocan en la interacción y responsabilidad entre objetos. Ejemplos: :

- Observer
- Strategy
- Command

Veremos con más profundidad algunos de estos patrones.

Patrón *Adapter*

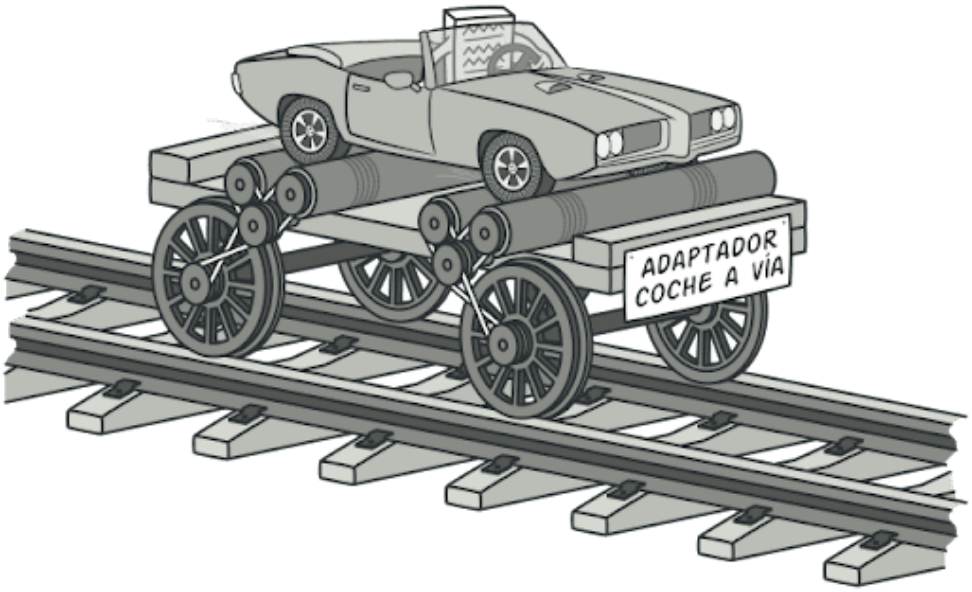


Figura 6.3: Patrón *Adapter*

El patrón *adapter* permite reutilizar código heredado o *legacy* cuya interfaz no coincide con la esperada por el sistema en el que estamos trabajando. Este patrón actúa como un puente entre la interfaz existente y la requerida, permitiendo que componentes incompatibles trabajen juntos sin modificar su código original heredado.

En la figura a continuación se observan los siguientes componentes.

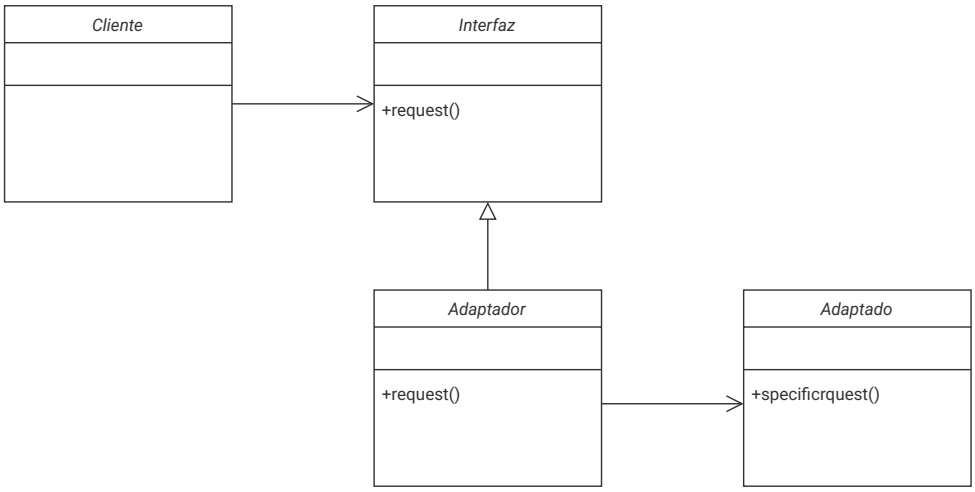


Figura 6.4: Diagrama de Clases del Patrón *Adapter*

Cliente Representa el sistema nuevo que espera una interfaz específica. En este ejemplo se observa que el cliente espera una interfaz que tiene el método `request()`.

Interfaz Define la interfaz esperada por el sistema nuevo.

Adaptado Representa la clase existente con la interfaz incompatible. En este ejemplo se observa que cuenta con el método `specificRequest()`.

Adaptador Convierte la interfaz del *Adaptado*. Dentro del método `request()` en el *Adaptador* se invoca el método específico del *Adaptado* `specificRequest()`. Eventualmente puede realizar alguna transformación de datos o invocar otros métodos del *Adaptado* para conseguir que `request()` cumpla con la interfaz esperada.

6.2.2.1 Cómo Proceder

1. Identificar los actores en juego: el ***Cliente*** y el ***Adaptado*** (componente *legacy*).
2. Identificar la ***Interfaz*** que requiere el ***Cliente***.
3. Verificar que el ***Adaptado*** que se quiere utilizar puede cumplir con la ***Interfaz*** solicitada.
4. Diseñar un envoltorio (***Adaptador***) que va a contener al ***Adaptado***.
5. Implementar el ***Adaptador*** para que cumpla con la ***Interfaz*** esperada por el ***Cliente***.
6. El ***Cliente*** interactúa con el ***Adaptador*** como si fuera el ***Adaptado***.

6.2.2.2 Ejemplo

Supongamos que tenemos un robot que realiza mediciones, cuyo sistema de control proporciona los métodos `Medir()` que devuelve un par de enteros, donde el primer número representa la distancia en metros y el segundo número la distancia en centímetros. Por ejemplo si la última medición fue de 10,5 m, entonces `Medir()` devolverá el par `(10, 50)`.

Nuestra empresa ha concretado la venta del robot a un cliente que necesita incorporar el robot a su sistema de producción, pero el sistema de control del cliente espera que el método `Medir` devuelva un solo número que represente la distancia en pulgadas.

1. Identificar los actores en juego: el ***Cliente*** y el ***Adaptado*** (componente *legacy*).
   - ***Cliente***: Sistema de control del cliente.
   - ***Adaptado***: Robot que realiza mediciones.
2. Identificar la ***Interfaz*** que requiere el ***Cliente***.
   - ***Interfaz***: Método `Medir()` que devuelve la distancia en pulgadas.
3. Verificar que el ***Adaptado*** que se quiere utilizar puede cumplir con la ***Interfaz*** solicitada.

- ***Adaptado***: Robot que realiza mediciones con el método `Medir()` que devuelve la distancia en metros y centímetros y se puede convertir a pulgadas.

4. Diseñar un envoltorio (***Adaptador***) que va a contener al ***Adaptado***.

Go

```go
// Adaptador que convierte la interfaz del Adaptado a la interfaz esperada
type RobotAdaptado struct {
    adaptado *Robot
}
```

1. Implementar el ***Adaptador*** para que cumpla con la ***Interfaz*** esperada por el ***Cliente***.

Go

```go
// Implementación del método requerido por la interfaz esperada
func (r *RobotAdaptado) Medir() float64 {
    metros, centimetros := r.adaptado.Medir()
    totalCentimetros := (metros * 100) + centimetros
    pulgadas := float64(totalCentimetros) / 2.54
    return pulgadas
}
```

1. El ***Cliente*** interactúa con el ***Adaptador*** como si fuera el ***Adaptado***.

Go

```go
// Cliente
robot := &Robot{}
adaptado := &RobotAdaptado{adaptado: robot}
distancia := adaptado.Medir()
fmt.Println(distancia) // distancia en pulgadas
```

En este ejemplo, el ***Adaptador*** `RobotAdaptado` convierte la interfaz del `Robot` en la interfaz requerida por el ***Cliente***, permitiendo que el sistema de control del cliente pueda utilizar el robot para realizar mediciones **sin modificar el código original** del robot.

Patrón *Composite*

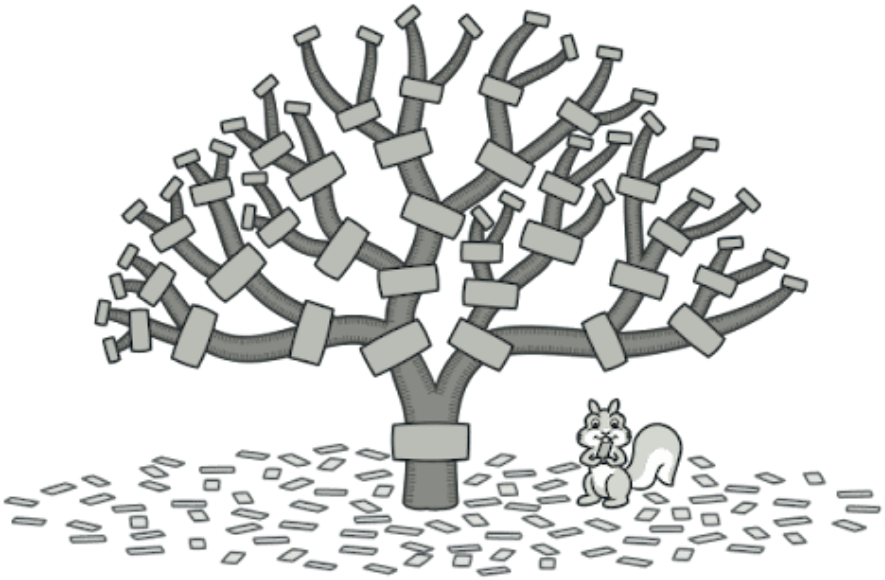


Figura 6.5: Patrón *Composite*

El patrón *composite* permite tratar tanto a objetos individuales como a composiciones de objetos de manera uniforme. Esto significa que se pueden tratar tanto a un objeto simple como a un grupo de objetos de la misma manera, sin tener que distinguir entre ellos. Esto simplifica el diseño y la implementación de estructuras jerárquicas de objetos.

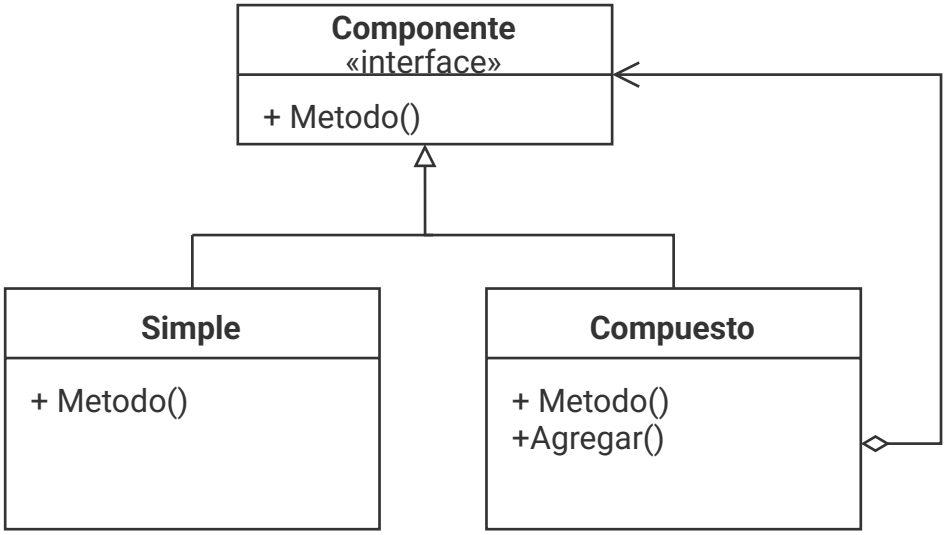


Figura 6.6: Diagrama de Clase del Patrón *Composite*

Componente Define la interfaz común para todos los elementos de la estructura.
Simple Representa los elementos individuales de la estructura.
Compuesto Representa los elementos que contienen otros elementos. Puede contener tanto objetos *Simples* como *Compuestos*. Se debe prever un método para agregar elementos a la colección, ya sea elementos *Simple* o *Compuesto*.

6.2.3.1 Cómo Proceder

1. Definir una interfaz común para todos los elementos de la estructura (***Componente***).
2. Implementar los tipos de datos que representen los elementos individuales (***Simple***), asegurándose de que cumplan con la interfaz común (***Componente***).

3. Implementar los tipos de datos que representen los elementos compuestos (***Compuesto***), que contienen una colección de elementos (***Componente***), asegurándose de que cumplan con la interfaz común (***Componente***) y contemplen la posibilidad de agregar elementos a la colección.
4. Tratar tanto a los elementos simples como a los compuestos de manera uniforme, sin tener que distinguir entre ellos.

6.2.3.2 Ejemplo

Supongamos que queremos modelar un sistema de archivos. Cada elemento del sistema —ya sea un archivo individual o una carpeta que contiene otros elementos— debe poder reportar su tamaño total en bytes. Queremos calcular el tamaño de una carpeta que puede contener archivos y otras carpetas, sin tener que distinguir entre ellos.

1. Definir una interfaz común para todos los elementos de la estructura (***Componente***).

Go

```go
// Componente define la operación común a archivos y carpetas
type Componente interface {
    Tamanio() int64
}
```

1. Implementar los tipos de datos que representen los elementos individuales (***Simple***), asegurándose de que cumplan con la interfaz común (***Componente***).

Go

```go
// Archivo representa un elemento simple (hoja) del sistema
type Archivo struct {
    nombre string
    bytes  int64
}

func (a *Archivo) Tamanio() int64 {
    return a.bytes
}
```

1. Implementar los tipos de datos que representen los elementos compuestos (***Compuesto***), que contienen una colección de elementos (***Componente***), asegurándose de que cumplan con la interfaz común (***Componente***) y contemplen la posibilidad de agregar elementos a la colección.

Go

```go
// Carpeta representa un elemento compuesto que contiene otros componentes
type Carpeta struct {
    nombre      string
    componentes []Componente
}
```

```
func (c *Carpeta) Tamanio() int64 {
    var total int64
    for _, comp := range c.componentes {
        total += comp.Tamanio()
    }
    return total
}

func (c *Carpeta) Agregar(comp Componente) {
    c.componentes = append(c.componentes, comp)
}
```

1. Tratar tanto a los elementos simples como a los compuestos de manera uniforme, sin tener que distinguir entre ellos.

Por ejemplo, queremos calcular el tamaño total de un directorio de proyecto que contiene archivos y subcarpetas:

Go

```
readme := &Archivo{nombre: "README.md", bytes: 2048}
licencia := &Archivo{nombre: "LICENSE", bytes: 1024}

src := &Carpeta{nombre: "src"}
src.Agregar(&Archivo{nombre: "main.go", bytes: 4096})
src.Agregar(&Archivo{nombre: "utils.go", bytes: 1536})

docs := &Carpeta{nombre: "docs"}
docs.Agregar(&Archivo{nombre: "manual.pdf", bytes: 524288})

proyecto := &Carpeta{nombre: "mi-proyecto"}
proyecto.Agregar(readme)
proyecto.Agregar(licencia)
proyecto.Agregar(src)
proyecto.Agregar(docs)

fmt.Println(proyecto.Tamanio()) // 532992
```

`proyecto.Tamanio()` recorre recursivamente todos los componentes —archivos y carpetas— sin necesidad de saber si cada uno es simple o compuesto. El método `Agregar` recibe un `Componente`, por lo que acepta tanto `Archivo` como `Carpeta`.

Patrón *Iterator*

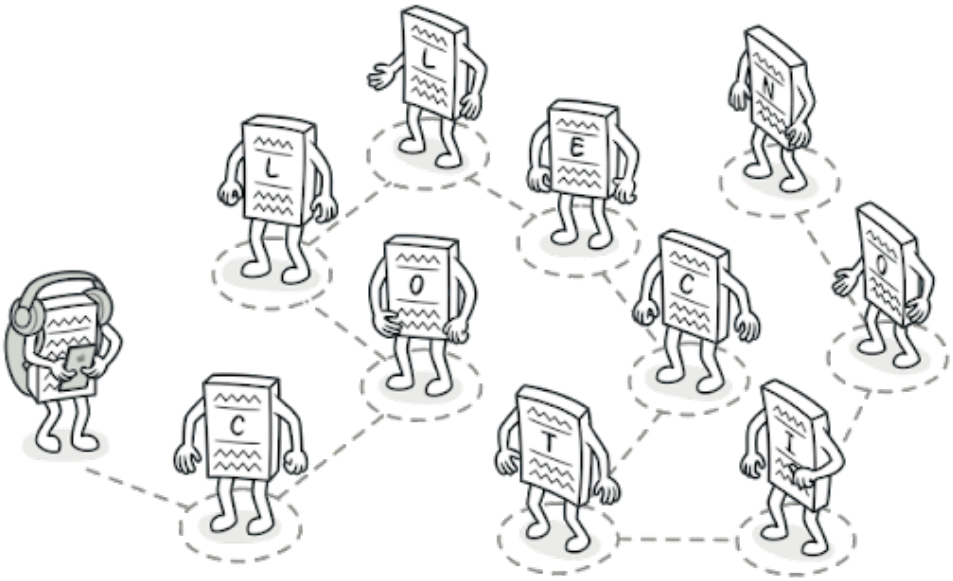


Figura 6.7: Patrón *Iterator*

El patrón *Iterator* o Iterador permite recorrer los elementos de una colección cualquiera sin exponer su estructura interna. El Iterador provee una interfaz uniforme con dos métodos:

`Siguiente() bool` Avanza el iterador al siguiente elemento y devuelve `true`. Si no quedan más elementos por recorrer, devuelve `false`. La primera llamada a `Siguiente()` posiciona el iterador en el primer elemento de la colección.

`Valor() int` Devuelve el valor del elemento sobre el cual está posicionado el iterador. Solo debe llamarse después de que `Siguiente()` haya devuelto `true`.

El patrón de uso típico en Go es:

Go

```go
it := coleccion.Iterador()
for it.Siguiente() {
    fmt.Println(it.Valor())
}
```

Cada llamada a `Siguiente()` cumple dos funciones: verifica si hay un elemento disponible y avanza la posición interna. `Valor()` simplemente retorna el elemento actual sin modificar el estado del iterador.

6.2.4.1 Cómo Proceder

1. Definir el comportamiento del **Iterador** con los métodos `Siguiente()` y `Valor()`. Si se necesita recorrer en ambos sentidos, se puede agregar un método `Anterior()`.
2. Dentro de la **colección** definir un método fábrica `Iterador()` que devuelva un iterador nuevo apuntando al inicio.
3. Implementar el **Iterador** vinculado siempre a una única colección.
4. Recorrer la **colección** con `for it.Siguiente()`.

6.2.4.2 Ejemplo

Supongamos que tenemos una lista enlazada simple y queremos recorrerla con un iterador. Por simplicidad la lista contiene solo números enteros.

Primero definimos la estructura de la lista y su operación de inserción al final:

Go

```go
type Nodo struct {
    valor int
    sig   *Nodo
}

type Lista struct {
    cabeza *Nodo
}

func (l *Lista) AgregarAlFinal(valor int) {
    nuevo := &Nodo{valor: valor}
    if l.cabeza == nil {
        l.cabeza = nuevo
        return
    }
    actual := l.cabeza
    for actual.sig != nil {
        actual = actual.sig
    }
    actual.sig = nuevo
}
```

El iterador se implementa como un struct que mantiene una referencia al nodo actual:

Go

```go
type Iterador struct {
    actual *Nodo
}

func (l *Lista) Iterador() *Iterador {
    return &Iterador{actual: l.cabeza}
}

// Siguiente avanza al próximo nodo.
// La primera llamada posiciona el iterador en la cabeza.
// Devuelve false cuando no hay más elementos.
func (it *Iterador) Siguiente() bool {
    if it.actual == nil {
        return false
    }
    // La primera vez, no avanza: devuelve la cabeza.
    // Las siguientes, avanza al nodo siguiente.
    if it.actual != nil {
        // guardamos el actual y avanzamos
```

```
    }
    return true
}

// Valor devuelve el valor del nodo actual.
func (it *Iterador) Valor() int {
    return it.actual.valor
}
```

El código anterior es una simplificación. La implementación real necesita un mecanismo para distinguir la primera llamada de las siguientes. A continuación la versión completa:

Go

```
type Iterador struct {
    actual  *Nodo
    primera bool
}

func (l *Lista) Iterador() *Iterador {
    return &Iterador{actual: l.cabeza, primera: true}
}

func (it *Iterador) Siguiente() bool {
    if it.actual == nil {
        return false
    }
    if !it.primera {
        it.actual = it.actual.sig
        if it.actual == nil {
            return false
        }
    }
    it.primera = false
    return true
}

func (it *Iterador) Valor() int {
    return it.actual.valor
}
```

La bandera `primera` evita que la llamada inicial a `Siguiente()` avance más allá del primer elemento. A partir de la segunda llamada, `Siguiente()` avanza al nodo siguiente antes de verificar si hay elemento.

Finalmente, el uso:

Go

```
lista := &Lista{}
lista.AgregarAlFinal(1)
lista.AgregarAlFinal(2)
```

```
lista.AgregarAlFinal(3)

it := lista.Iterador()
for it.Siguiente() {
    fmt.Println(it.Valor())
}
// Salida:
// 1
// 2
// 3
```

Ejercicios

Los ejercicios de este capítulo cubren los tres patrones vistos: **Adapter**, **Composite** e **Iterator**. Están en el directorio `02-patrones-de-diseno/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// Completar` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.3 Algoritmos Ávidos

Los algoritmos ávidos o *greedy* en inglés, son algoritmos diseñados generalmente para resolver problemas de optimización. Los algoritmos que resuelven este tipo de problemas generalmente lo hacen en etapas, realizando, a cada paso, algunos cálculos hasta llegar al resultado buscado. Un algoritmo ávido o codicioso siempre hace la elección que parece mejor en el momento. Es decir, hace una elección localmente óptima con la esperanza de que esta elección conduzca a una solución globalmente óptima. En otras palabras con los datos que está procesando en ese momento, toma la mejor decisión posible.

La estrategia ávida no siempre funciona, depende de la naturaleza del problema y de los datos, por lo que hay que analizar bien el problema antes de seguir esta estrategia.

La principal ventaja de este tipo de algoritmos es que es muy fácil de implementar y entender.

Por ejemplo veamos a continuación un problema clásico:

Cambio de moneda

Un cajero automático tiene que ser capaz de entregar la cantidad de pesos que se le requiere utilizando la menor cantidad de billetes y monedas.

Por ejemplo, en Argentina, a mayo de 2024 se emplean monedas de $1, $2, $5 y $10, y billetes de $10, $20, $50, $100, $200, $500, $1.000, $2.000 y $10.000. Si el cajero tiene que entregar $5.528

La solución será:

$$2 times 2000 + 1 times 1000 + 1 times 500 + 1 times 20 + 1 times 5 + 1 times 2 + 1 times 1 (6.12)$$

En total se entregan 8 piezas.

Algoritmo: Cambio de moneda

Entrada

- `billetes`: lista con las denominaciones disponibles ordenada de mayor a menor.
- `cantidad`: monto a cambiar

Salida

- `cambio`: diccionario cuyas claves son las denominaciones y cuyo valor son las cantidades a entregar de cada denominación

text

```
PARA CADA denominacion EN billetes HACER
    SI cantidad >= denominacion ENTONCES
        cantidad_billetes ← cantidad / denominacion
        cambio[denominacion] ← cantidad_billetes
        cantidad ← cantidad MOD denominacion
```

```
    FIN SI
FIN PARA
```

A partir de este fragmento nos podemos preguntar:

1. ¿Siempre puede entregar el cambio solicitado?
2. ¿Siempre entrega la menor cantidad de billetes?

Para que siempre pueda entregar el monto solicitado, es preciso contar con la moneda de $1, para asegurarnos que podrá entregar cualquier monto.

Para asegurar que siempre da la menor cantidad de billetes, el sistema de denominaciones debe ser canónico. Es decir las denominaciones no pueden ser cualquiera, sino que deben ser un conjunto que asegure que siempre funciona el algoritmo ávido. Además deben estar ordenadas de mayor a menor.

Por ejemplo si las denominaciones fueran: $1, $2, $5, $10, $20, $50, $100, $200, $500, $800 y $1.000 y tiene que dar $2.400, nuestro algoritmo ávido daría 2 x $1.000 + 2 x $200 en lugar de dar 3 billetes de $800.

La verificación formal de que un algoritmo ávido es correcto es un tema complejo y no siempre se puede hacer. En general se hace por inducción, es decir se prueba que la elección localmente óptima lleva a una solución globalmente óptima. Se puede hacer utilizando técnicas de programación dinámica.

A continuación otro ejemplo de algoritmo ávido muy popular en la literatura:

Plan de actividades

Se tiene un conjunto de actividades que se pueden realizar en un tiempo determinado. Cada actividad a_i tiene un tiempo de inicio s_i y un tiempo de finalización f_i. Se desea seleccionar la mayor cantidad de actividades que no se superpongan.

Dos actividades a_i y a_j son compatibles si los intervalos $[s_i, f_i)$ y $[s_j, f_j)$ no se superponen. Es decir:

$$f_i \leq s_j \tag{6.13}$$

O

$$f_j \leq s_i \tag{6.14}$$

Supongamos que tenemos las siguientes actividades, ordenadas por tiempo de finalización:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	15

Figura 6.8: Lista de actividades

A continuación el diagrama de actividades. En principio no hay ninguna actividad seleccionada.

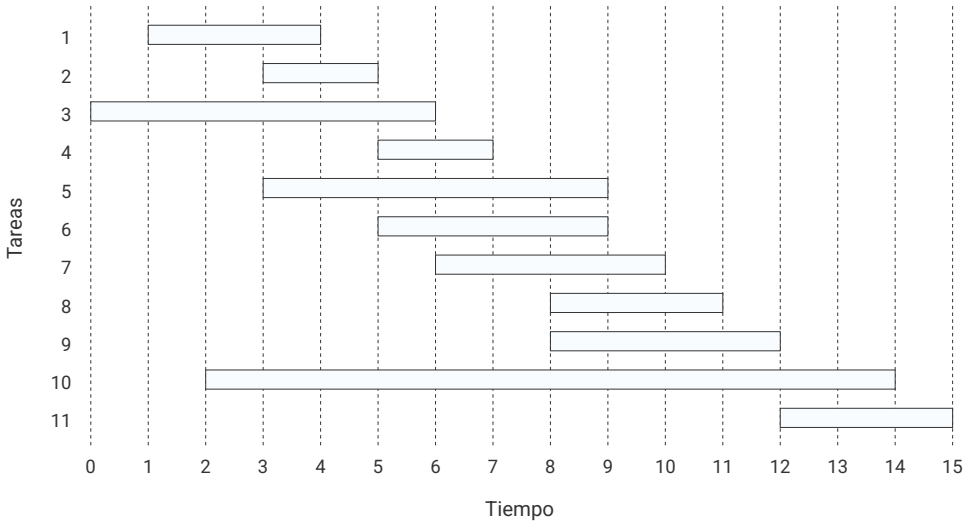


Figura 6.9: Diagrama de actividades

Paso 1 Se selecciona la tarea que termina primero de entre todas las tareas a planificar. En este caso la tarea 1. Inmediatamente las tareas 2, 3, 5, 10 se marcan como incompatibles, ya que se superponen con la tarea 1.

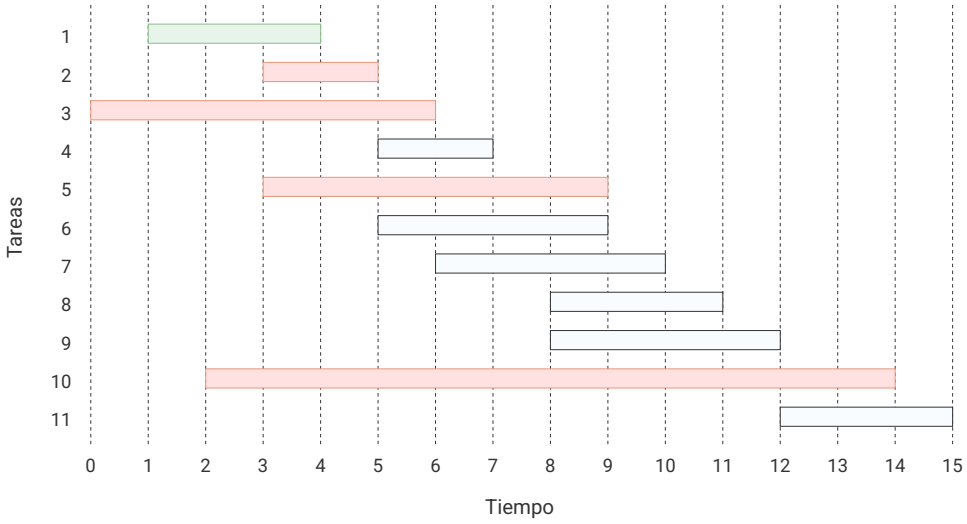


Figura 6.10: Diagrama de actividades

Paso 2 De entre las tareas disponibles (4, 6, 7, 8, 9 y 11) se selecciona la tarea 4, que es la que finaliza primero, por lo tanto las tareas 6 y 7 dejan de ser compatibles con la planificación.

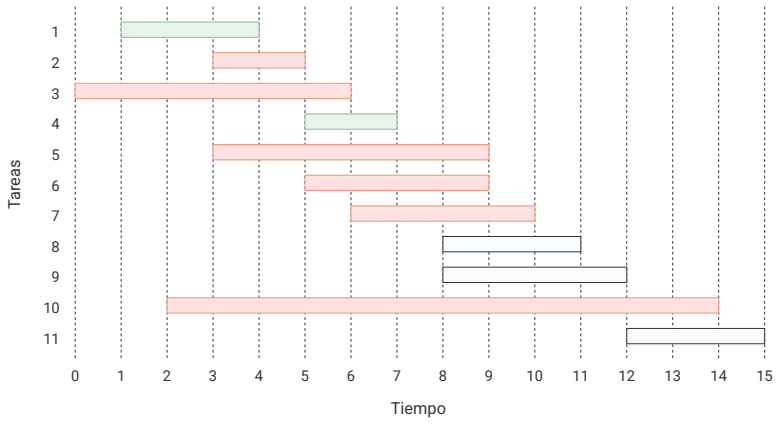


Figura 6.11: Diagrama de actividades

Paso 3 Se selecciona la tarea 8, ya que es la que finaliza primero entre las tareas disponibles (8, 9 y 11). La tarea 9 se vuelve incompatible.

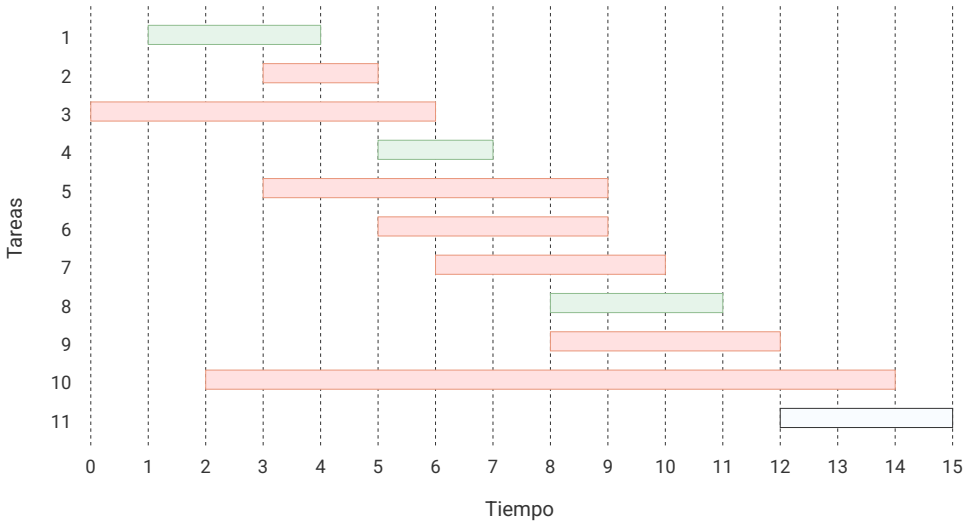


Figura 6.12: Diagrama de actividades

Paso 4 Por último se selecciona la única tarea disponible, la 11.

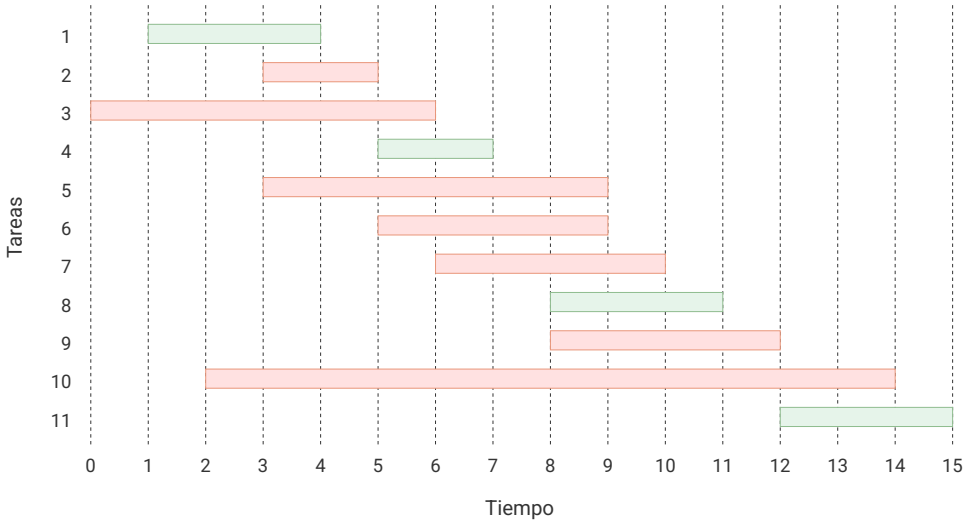


Figura 6.13: Diagrama de actividades

El algoritmo ávido para este problema es el siguiente:

Algoritmo: Plan de actividades

Entrada: Lista de actividades ordenadas por tiempo de finalización

Salida: Lista de actividades seleccionadas

text

```
actividades_seleccionadas ← []
actividad_actual ← actividades[0]
```

```
actividades_seleccionadas ← actividades_seleccionadas + [actividad_actual]
PARA CADA actividad EN actividades HACER
    SI actividad_inicio >= actividad_actual_fin ENTONCES
        actividades_seleccionadas ← actividades_seleccionadas + [actividad]
        actividad_actual ← actividad
    FIN SI
FIN PARA
```

Observación

Como la lista de entrada de las tareas está ordenada por tiempo de finalización, no hace falta programar la ecuación (3)

Ejercicios

Los ejercicios de este capítulo están en el directorio `03-algoritmos-avidos/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.4 Backtracking (Vuelta Atrás)

Backtracking es una técnica algorítmica para resolver problemas donde la solución se construye probando posibilidades paso a paso y descartando caminos inviables. Siempre que el problema tenga solución, con esta técnica la podemos encontrar, ya que explora todas las posibilidades hasta encontrarla, al costo de no ser la más eficiente.

Se utiliza comúnmente en problemas de decisión, combinatoria y optimización donde la solución se puede construir de manera incremental.

La idea es agregar un nuevo elemento a la solución parcial y verificar si es una solución válida. Si no lo es, se descarta y se prueba con otro elemento. Este proceso se repite hasta encontrar la solución o encontrar que no hay más opciones disponibles y por lo tanto no hay solución.

Con esta técnica también se pueden encontrar todas las soluciones posibles a un problema, no solo una. Para ello, se debe modificar el algoritmo para que no termine al encontrar la primera solución, sino que continúe buscando.

Ejemplo: Problema de las N Reinas

El problema de las N reinas consiste en colocar N reinas en un tablero de ajedrez de $N \times N$ de tal manera que ninguna reina ataque a otra. Esto significa que no pueden estar en la misma fila, columna o diagonal. En la siguiente figura se muestra una solución para el caso de tableros de 8×8 donde se colocan 8 reinas.

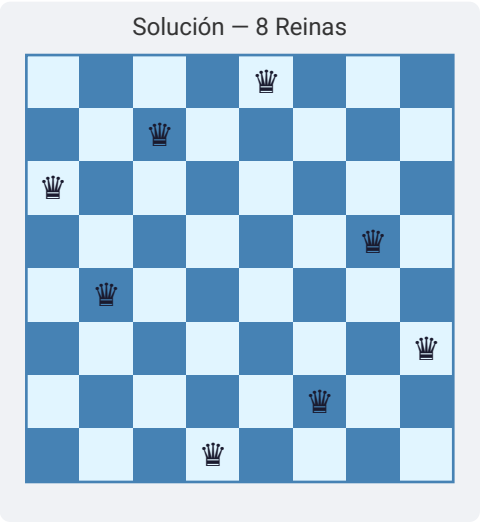


Figura 6.14: Solución al problema de las N reinas para $N = 8$

Para encontrar una solución posible, la idea es ir colocando una reina en cada fila del tablero, comenzando desde la primera fila. En cada fila se va probando con las distintas columnas, verificando si la posición es válida (no hay otra reina en la misma columna o diagonal). Si se encuentra una posición válida, se coloca la reina y se pasa a la siguiente fila. Si no se encuentra

ninguna posición válida en una fila, se vuelve atrás (*backtracking*) y se prueba con otra columna en la fila anterior.

Algoritmo de *Backtracking*

A continuación se muestra la estructura general de este esquema:

text

```
FUNCION backtracking(solucionParcial, ...)
    SI esSolucion(solucionParcial) ENTONCES
        mostrar(solucionParcial)
        SALIR // o continuar buscando si se desea encontrar todas las soluciones
    SINO
        PARA CADA opcion posible de extender solucionParcial HACER
            SI esFactible(opcion) ENTONCES
                registrar(solucionParcial, opcion) // agregar opcion a la solución
parcial
                backtracking(solucionParcial, ...) // llamada recursiva
                borrar(opcion, solucionParcial)    // vuelta atrás
            FIN SI
        FIN PARA
    FIN SI
FIN FUNCION
```

6.4.2.1 Cómo proceder para resolver un problema con *Backtracking*

Para resolver un problema con *backtracking*, se deben identificar y definir los siguientes elementos:

`solucionParcial` Determinar una estructura de datos adecuada para representar la solución parcial. Puede ser un vector, una lista, etc. `solucionParcial` debe permitir agregar nuevas opciones y verificar si es una solución válida.

`esSolucion` Definir una función que verifique si `solucionParcial` es una solución completa.

`extender` Definir una función que tome un `solucionParcial` y devuelva una a una todas las opciones posibles para extenderla. Esta función debe generar todas las combinaciones posibles de extender `solucionParcial`.

`esFactible` Definir una función que verifique si una opción es válida para extender `solucionParcial`. Esta función debe verificar si la opción no viola ninguna restricción del problema.

`registrar` Definir una función que registre la opción generada con `extender` en la `solucionParcial`. Eventualmente puede ser necesario registrar información adicional de la opción generada.

`borrar` Definir una función que borre la última opción registrada en `solucionParcial`. Es decir es el paso opuesto al de `registrar`. Si es necesario también debe restaurar cualquier información extra o de contexto registrada anteriormente.

Implementación de la solución al problema de las N reinas

`solucionParcial` puede ser un arreglo de tamaño N donde las posiciones del arreglo representan las filas y los valores almacenados en cada posición del arreglo representan las columnas. Por ejemplo, para el caso de $N = 4$, una solución parcial podría ser $[2, 0, 3, 1]$, lo que significa que hay una reina en la fila 0 en la columna 2, una reina en la fila 1 en la columna 0, etc.

`esSolucion` es una función que verifica si la `solucionParcial` tiene N reinas. En este caso, se verifica si la longitud del arreglo `solucionParcial` es igual a N.

`extender` con un solo ciclo entre $[0, N)$, podemos generar todas las posibles columnas en la que se puede colocar una reina en la fila i.

`esFactible` es una función que debe chequear si en la misma columna o en las diagonales hay otras reinas que entren en conflicto con la nueva reina que intentamos agregar al tablero.Para chequear si hay ya una reina en la misma columna basta con revisar los valores del arreglo `solucionParcial` para asegurarse que la columna candidata no está atacada por otra reina, es decir no puede haber dos valores iguales en el arreglo y para chequear si hay una reina en alguna de las dos diagonales que se pueden generar en cualquier posición del tablero, se puede usar la propiedad de que en todas las diagonales directas, por nombrarlas de alguna forma, las de color azul, $fila - columna = constante$. Mientras que para las diagonales inversas, las de color rojo, se cumple que $fila + columna = constante$. Ver la figura a continuación.

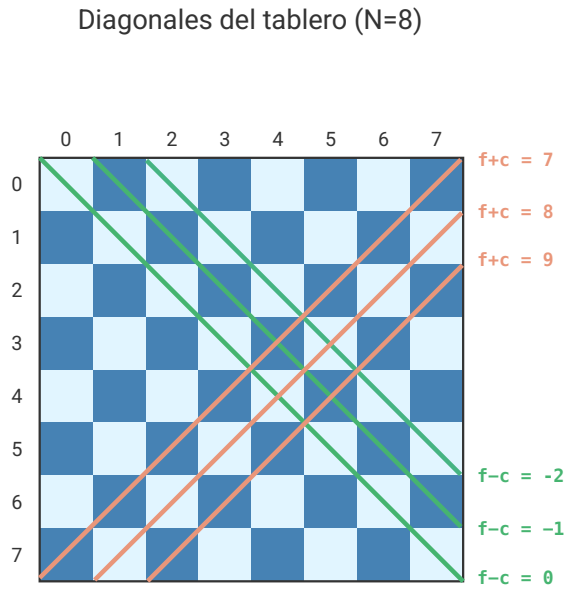


Figura 6.15: Reinas en la misma diagonal

El código completo de esta implementación está disponible en el repositorio `taller-algoritmos`, en el directorio `04-backtracking/ejemplos/nreinas/`. Allí se puede ejecutar con `go run ./04-backtracking/ejemplos/nreinas/`.

Análisis de la complejidad computacional

Para entender cuánto tarda un algoritmo de *backtracking*, pensemos en el problema de las N reinas:

- En la primera fila podemos poner una reina en cualquiera de las N columnas -> N opciones.
- En la segunda fila, como no puede repetir columna, nos quedan $N-1$ opciones.
- En la tercera fila, $N-2$ opciones, y así sucesivamente.

Por eso, en el peor de los casos el algoritmo prueba $N \times (N-1) \times (N-2) \times ... \times 1 = N!$ combinaciones. Decimos que su complejidad es $O(N!)$ (orden factorial).

Pensemos con números concretos: para $N = 8$, $N! = 40320$ combinaciones. La computadora lo resuelve rápido. Pero para $N = 20$, $N! \approx 2.4 \times 10^{18}$, una cantidad tan enorme que el algoritmo tardaría siglos.

6.4.4.1 Vista general: el árbol de búsqueda

Podemos pensar el *backtracking* como un árbol donde:

- Cada **nivel** del árbol representa un paso en la construcción de la solución (por ejemplo, colocar una reina en una fila).
- De cada nodo salen **ramas** que representan las opciones disponibles en ese paso.

En la figura siguiente, el árbol tiene $d = 3$ niveles (profundidad) y de cada nodo salen $b = 3$ ramas (factor de ramificación):

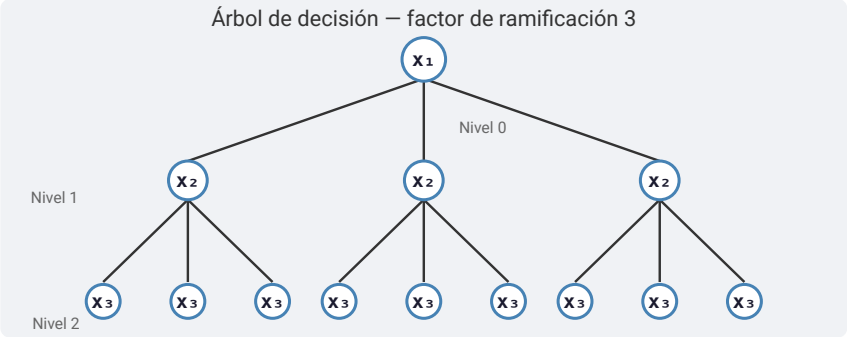


Figura 6.16: Árbol de búsqueda para el caso de $b = 3$ y $d = 3$

La cantidad de nodos en un árbol así es b^d (en este ejemplo $3^3 = 27$). Como el algoritmo podría tener que recorrer todo el árbol, su complejidad general es $O(b^d)$.

¿Qué significa esto en la práctica? Si b y d son grandes, b^d crece muy rápido. Por ejemplo, con $b = 10$ y $d = 10$ tendríamos $10^{10} = 10000$ millones de nodos, demasiados para recorrerlos todos.

6.4.4.2 Cómo mejorar la eficiencia

Un orden de complejidad factorial ($N!$) o exponencial (b^d) no es aceptable para problemas grandes. Para mejorar el tiempo de ejecución se usan dos técnicas principales:

Poda (*pruning*): consiste en cortar ramas del árbol que no pueden llevar a una solución válida, evitando recorrerlas. Por ejemplo, en el problema de las N reinas, si ya colocamos dos reinas en la misma columna, no tiene sentido seguir probando opciones en esa rama: la cortamos.

Heurística: usar información del problema para decidir qué rama explorar primero, con suerte encontrando la solución más rápido. Por ejemplo, podríamos empezar probando las columnas del centro del tablero porque estadísticamente dan más soluciones.

Estas técnicas no cambian la complejidad en el peor caso, pero en la práctica reducen mucho el tiempo de ejecución.

Conclusiones

Backtracking es una técnica algorítmica poderosa para resolver problemas de decisión, combinatoria y optimización. Aunque su complejidad computacional puede ser alta, en muchos casos es la única forma de encontrar una solución. La clave para implementar un algoritmo de *backtracking* eficiente es definir correctamente los elementos que componen el algoritmo y utilizar técnicas como la poda o heurísticas para reducir el tiempo de ejecución.

La implementación de *backtracking* puede ser sencilla y elegante, como se ha visto en el ejemplo del problema de las N reinas. Sin embargo, es importante tener en cuenta que no siempre es la mejor opción para resolver un problema, ya que en algunos casos puede ser más eficiente utilizar otras técnicas algorítmicas como la programación dinámica, algoritmos ávidos o soluciones aproximadas.

Ejercicios

Los ejercicios de este capítulo están en el directorio `04-backtracking/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.5 Programación Dinámica

La programación dinámica es otra técnica de optimización utilizada para resolver problemas complejos mediante la descomposición en subproblemas más pequeños y su solución incremental, con el objetivo de encontrar la solución óptima global.

A diferencia de los algoritmos ávidos, que toman decisiones locales óptimas en cada paso sin reconsiderarlas, la programación dinámica resuelve todos los subproblemas posibles y combina sus soluciones para construir la solución óptima global.

El problema original se subdivide en subproblemas de menor tamaño y se almacenan sus soluciones para evitar cálculos redundantes. Esto se logra mediante la técnica de **memoización**, que almacena los resultados de los subproblemas ya resueltos, y la técnica de **tabulación**, que construye una tabla para almacenar las soluciones de los subproblemas.

Importante

La principal característica de la programación dinámica es que los cálculos se realizan una sola vez y se reutilizan en lugar de volver a calcularlos.

La programación dinámica es especialmente útil para problemas de optimización, donde se busca maximizar o minimizar una función objetivo. Se utiliza en una amplia variedad de aplicaciones, como la teoría de grafos, la teoría de juegos, la bioinformática y la economía.

Esta técnica fue concebida para resolver problemas con **subestructura óptima** y **subproblemas superpuestos**.

Subestructura óptima Un problema tiene subestructura óptima si su solución óptima puede construirse a partir de soluciones óptimas de sus subproblemas.

Subproblemas superpuestos Ocurren cuando un problema puede dividirse en subproblemas que se repiten múltiples veces.

Serie de Fibonacci

La serie de Fibonacci puede definirse de la siguiente manera:

$$F(n) = \tag{6.15}$$

Una posible implementación del cálculo de la serie de Fibonacci utilizando recursión es la siguiente:

Go

```go
func Fibonacci(n int) int {
    if n <= 1 {
        return n
    }
```

```
    return Fibonacci(n-1) + Fibonacci(n-2)
}
```

En la línea 5 se observa que la función `Fibonacci` se llama a sí misma dos veces, lo que provoca que se realicen cálculos redundantes. Por ejemplo, al calcular `Fibonacci(5)`, se realizan las siguientes llamadas:

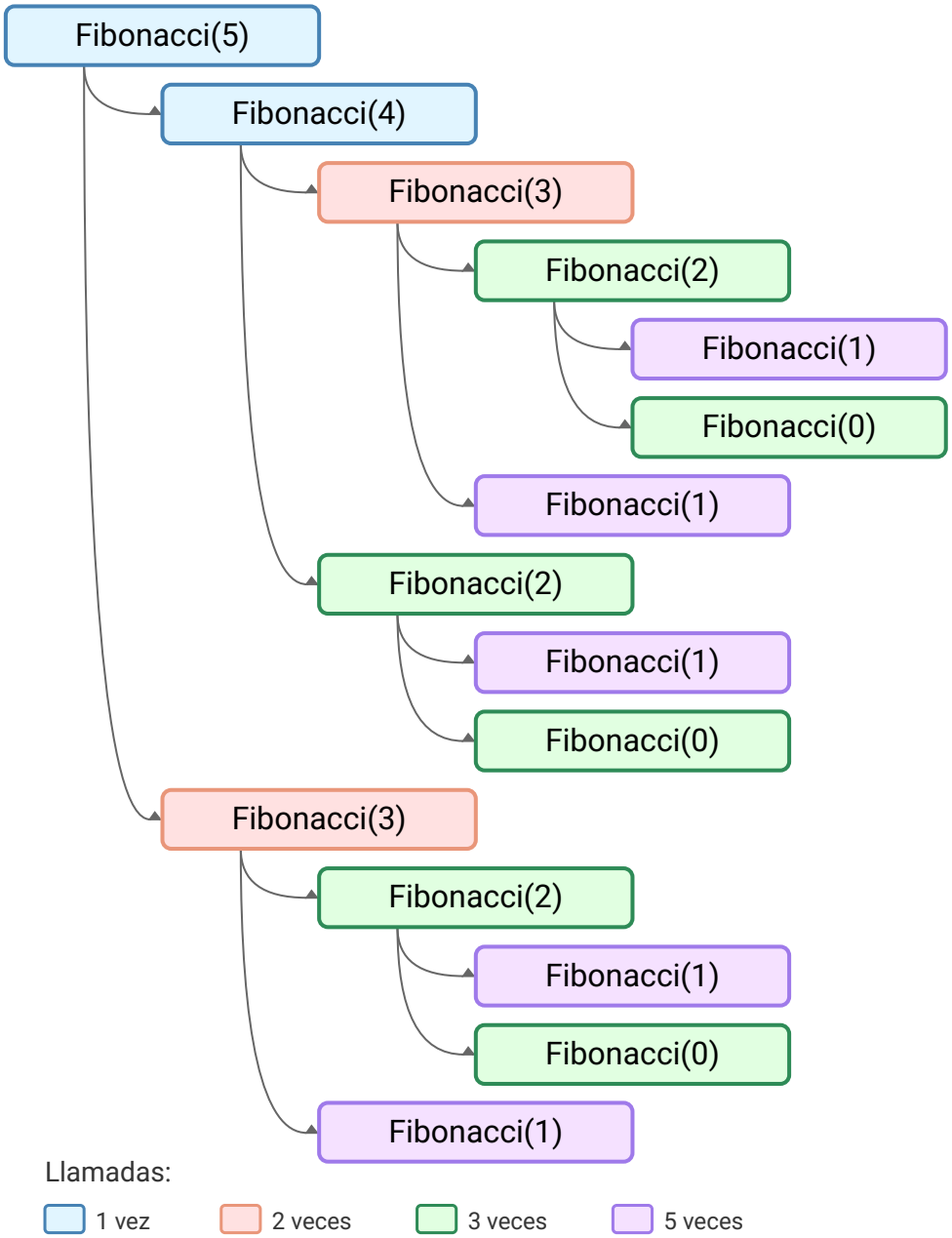


Figura 6.17: Árbol de llamadas recursivas de Fibonacci(5). Los colores indican cuántas veces se repite cada cálculo.

Se puede observar que `Fibonacci(5)` y `Fibonacci(4)` se calcularon 1 vez, `Fibonacci(3)` se calculó 2 veces, `Fibonacci(2)` y `Fibonacci(0)` se calcularon 3 veces cada uno, y `Fibonacci(1)` se calculó 5 veces. Esto significa que el tiempo de ejecución del algoritmo es exponencial, lo que lo hace ineficiente para valores grandes de `n`.

A continuación se muestra cómo se construyen los valores utilizando una tabla para almacenar los resultados de los subproblemas:

Paso	Valor calculado	Cómo se obtiene
0	`f(0) = 0`	Caso base: se inicializa la tabla
1	`f(1) = 1`	Caso base: se inicializa la tabla
2	`f(2) = 1`	`f(1) + f(0) = 1 + 0` (reutiliza valores)
3	`f(3) = 2`	`f(2) + f(1) = 1 + 1` (reutiliza valores)
4	`f(4) = 3`	`f(3) + f(2) = 2 + 1` (reutiliza valores)
5	`f(5) = 5`	`f(4) + f(3) = 3 + 2` (reutiliza valores)

Observar que los valores de `f(0)` y `f(1)` se usan directamente (no hace falta recalcularlos), y cada paso reutiliza los valores ya almacenados en la tabla.

Esta forma de implementar algoritmos por programación dinámica usando una tabla para almacenar los valores se conoce como **tabulación**, en inglés ***Bottom-Up***.

Se pueden distinguir dos etapas en la tabulación:

Inicialización de la tabla Se inicializa una tabla con los valores base de la serie de Fibonacci. Estos valores iniciales son necesarios para calcular los valores posteriores de la serie.

Llenado iterativo de la tabla Se van completando los valores de la tabla de forma iterativa, utilizando los valores previamente calculados. En cada iteración, se calcula el valor de `Fibonacci(n)` sumando los dos valores anteriores en la tabla, es decir, `Fibonacci(n-1)` y `Fibonacci(n-2)`. El proceso se repite hasta que se alcanza el valor deseado de `n`.

El resultado final se encuentra en la última celda de la tabla, que contiene el valor de Fibonacci(n). Este enfoque evita la recursión y los cálculos redundantes, lo que mejora significativamente la eficiencia del algoritmo.

En el siguiente fragmento de código se muestra la implementación de la serie de Fibonacci utilizando tabulación:

Go

```go
func Fibonacci(n int) int {
    // Inicialización de la tabla
    fib := make([]int, n+1)
    fib[0] = 0 // no hace falta inicializarlo, pero es una buena práctica
    fib[1] = 1

    // Llenado iterativo de la tabla
    for i := 2; i <= n; i++ {
        fib[i] = fib[i-1] + fib[i-2]
    }

    return fib[n]
}
```

En este caso donde la tabla es en realidad un arreglo, se puede estimar la complejidad temporal y espacial del algoritmo:

- Complejidad temporal: $O(n)$, ya que se realiza un solo recorrido a través de la tabla.
- Complejidad espacial: $O(n)$, ya que se utiliza una tabla de tamaño `n+1` para almacenar los resultados de los subproblemas.

Problema de la mochila (*Knapsack Problem*)

Figura 6.19: Problema de la mochila

El problema de la mochila es un clásico en programación dinámica y se puede enunciar como:

> Dado un conjunto de objetos, cada uno con un peso y un valor, el objetivo es determinar qué objetos se deben incluir en una mochila de capacidad limitada para maximizar el valor total.

Supongamos que tenemos una mochila de capacidad 5 y los siguientes objetos:

Objeto	Peso	Valor
1	3	2
2	4	3
3	1	4
4	2	2
5	5	5

La solución se puede encontrar utilizando nuevamente tabulación. Para ello se construye una tabla con las siguientes características:

- **Cada columna** representa una capacidad posible de la mochila, desde 0 hasta la capacidad máxima ($w = 5$ en este ejemplo).
- **Cada fila** representa los objetos que se van incorporando de a uno. La fila i corresponde a tener disponibles los primeros i objetos.
- **Cada celda** (i, w) almacena el máximo valor total que se puede obtener usando los i primeros objetos con una mochila de capacidad w.

Inicialización de la tabla Para poder inicializar la tabla se debe considerar que si no hay objetos o la capacidad de la mochila es 0, el valor máximo que se puede obtener es 0. Por lo tanto, se inicializan la primera fila y la primera columna de la tabla con 0.

La ecuación de recurrencia que determina el valor de cada celda es:

$$V(i, w) = \tag{6.16}$$

Donde p_i y v_i son el peso y valor del objeto i, w es la capacidad considerada en esa columna, y $V(i, w)$ es el valor máximo alcanzable con los primeros i objetos y capacidad w.

La implementación de esta estrategia utilizando tabulación es la siguiente:

Go

```go
type Item struct {
    peso  int
    valor int
}

func Mochila(obj []Item, capacidad int) int {
    n := len(obj)
    // Inicialización de la tabla
    dp := make([][]int, n+1)
    for i := range dp {
        dp[i] = make([]int, capacidad+1)
    }
    // Llenado iterativo de la tabla
    for i := 1; i <= n; i++ {
        for w := 1; w <= capacidad; w++ {
            if obj[i-1].peso > w {
                dp[i][w] = dp[i-1][w]
            } else {
                sin := dp[i-1][w]
                con := obj[i-1].valor + dp[i-1][w-obj[i-1].peso]
                if con > sin {
                    dp[i][w] = con
                } else {
                    dp[i][w] = sin
                }
            }
        }
    }
    return dp[n][capacidad]
}
```

Como en el caso de Fibonacci, la tabla se inicializa con ceros y se recorre de forma iterativa. En cada celda se aplica la ecuación de recurrencia: si el objeto no entra se hereda el valor de la fila anterior, y si entra se elige el máximo entre no incluirlo e incluirlo.

La complejidad temporal y espacial del algoritmo es:

Complejidad temporal $O(n \times w)$, donde n es el número de objetos y w es la capacidad de la mochila. Esto se debe a que se recorre la tabla de tamaño n por w.

Complejidad espacial $O(n \times w)$, ya que se utiliza una tabla de tamaño n por w para almacenar los resultados de los subproblemas.

Tabulación (*Bottom-Up*) vs. Memoización (*Top-Down*)

La **tabulación** y la **memoización** son dos enfoques diferentes para implementar la programación dinámica, y cada uno tiene sus propias ventajas y desventajas.

6.5.3.1 Tabulación (*Bottom-Up*)

Como vimos, la **tabulación** es un enfoque iterativo que construye una tabla y la recorre en orden sistemático. Para el problema de la mochila, se recorre fila por fila y columna por columna, resolviendo todos los subproblemas posibles. El código de la sección anterior sigue exactamente esta estrategia: inicializa `dp[n+1][capacidad+1]` con ceros y los recorre con dos `for` anidados.

6.5.3.2 Memoización (*Top-Down*)

La **memoización** ataca el mismo problema desde el otro extremo: en lugar de llenar toda la tabla de abajo hacia arriba, arranca desde el problema original y baja recursivamente hasta los casos base, almacenando los resultados intermedios en una estructura de datos. Si un subproblema ya fue resuelto, se recupera del caché en lugar de recalcularlo.

Aplicada al problema de la mochila, la versión memoizada usa una matriz `memo` inicializada con `-1` para indicar «no calculado aún». La primera llamada es `Mochila(obj, n, capacidad, memo)` y a partir de ahí la recursión desciende hasta `i == 0` o `w == 0`:

Go

```
func Mochila(obj []Item, i int, w int, memo [][]int) int {
    if i == 0 || w == 0 {
        return 0
    }
    if memo[i][w] != -1 {
        return memo[i][w]
    }
    if obj[i-1].peso > w {
        memo[i][w] = Mochila(obj, i-1, w, memo)
    } else {
        sin := Mochila(obj, i-1, w, memo)
        con := obj[i-1].valor + Mochila(obj, i-1, w-obj[i-1].peso, memo)
        if con > sin {
            memo[i][w] = con
        } else {
            memo[i][w] = sin
        }
    }
    return memo[i][w]
}
```

En este caso se utiliza una matriz `memo` inicializada en `-1` para detectar si un subproblema ya fue calculado. Cuando `memo[i][w] != -1` se devuelve el valor cacheado sin expandir la recursión, evitando así repetir cálculos.

La complejidad de ambas versiones aplicadas al mismo problema es idéntica:

Complejidad temporal $O(n \times w)$, tanto para tabulación como para memoización. En tabulación se recorren todas las celdas; en memoización cada celda se calcula una sola vez (la primera que se necesita) y luego se recupera del caché.

Complejidad espacial $O(n \times w)$, ya que ambas versiones almacenan la tabla completa de tamaño $(n+1) \times (w+1)$.

6.5.3.3 Comparación

Aunque las complejidades coinciden, los enfoques difieren en aspectos prácticos:

Característica	Tabulación (*Bottom-Up*)	Memoización (*Top-Down*)
Enfoque	Iterativo	Recursivo
Orden de llenado	Por filas, sistemático	Por demanda, recursivo
Subproblemas calculados	Todos	Solo los necesarios
Complejidad temporal	$O(n \times w)$	$O(n \times w)$
Complejidad espacial	$O(n \times w)$	$O(n \times w)$

Ejercicios

Los ejercicios de este capítulo están en el directorio `05-programacion-dinamica/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.6 Ordenamientos por Comparación de Claves

Introducción: La Importancia del Orden

El **ordenamiento de datos** es una de las operaciones fundamentales en la informática y la ciencia de datos. Desde organizar una lista de contactos por nombre hasta clasificar resultados de búsqueda por relevancia o procesar grandes volúmenes de datos para análisis estadísticos, la capacidad de ordenar eficientemente la información es crucial. Un conjunto de datos ordenado facilita enormemente la búsqueda, la inserción y la eliminación, mejorando el rendimiento de casi cualquier aplicación que trabaje con colecciones de elementos.

Pensemos en la vida cotidiana: una biblioteca donde los libros están ordenados alfabéticamente por autor, un cajón de herramientas donde los destornilladores están por tamaño, o una lista de precios en un supermercado. En todos estos casos, el orden nos permite encontrar lo que necesitamos de forma rápida y eficiente. En el ámbito computacional, este principio se traduce en algoritmos que, a través de diversas estrategias, transforman una secuencia desordenada en una secuencia ordenada.

Algunas características de los algoritmos de ordenamiento incluyen:

Complejidad temporal La cantidad de tiempo que un algoritmo tarda en ordenar un conjunto de datos. Se mide en función del número de elementos a ordenar y se expresa comúnmente en notación O grande (por ejemplo, $O(n \log n)$, $O(n^2)$). En los algoritmos de ordenamiento, además de establecer la complejidad temporal para el peor caso, se suele analizar los casos promedio y el mejor caso, teniendo en cuenta cuando se presenta cada escenario.

Peor Caso Describe la situación en la que el algoritmo tarda el mayor tiempo posible en completarse. Esto ocurre cuando los datos de entrada están dispuestos de tal manera que el algoritmo se ve obligado a realizar el número máximo de operaciones. El análisis del peor caso nos da una cota superior. Nos permite garantizar que el algoritmo nunca tardará más de un cierto tiempo, sin importar cómo estén los datos de entrada.

Caso Promedio Representa el tiempo de ejecución esperado del algoritmo cuando los datos de entrada son aleatorios o no tienen un patrón específico. Calcularlo puede ser más complejo, ya que implica considerar la probabilidad de diferentes configuraciones de entrada. El análisis del caso promedio nos da una idea más realista del rendimiento del algoritmo en situaciones típicas.

Mejor Caso Es la situación en la que el algoritmo tarda el menor tiempo posible en completarse. Esto sucede cuando los datos de entrada están dispuestos de una manera ideal para el algoritmo. Aunque no es tan útil como el peor caso o el caso promedio para predecir el rendimiento general, el mejor caso puede ser de utilidad en algoritmos de ordenamiento adaptativos que puedan adaptarse a los datos de entrada.

Estabilidad Un algoritmo es estable si mantiene el orden relativo de los elementos iguales después del ordenamiento. Esto es importante en situaciones donde los elementos tienen múltiples claves de ordenamiento. Por ejemplo si una clave de ordenamiento aparece

repetida varias veces en una secuencia de datos desordenadas, puede ser interesante mantener el orden relativo (el primero que aparece, luego el segundo, etc.) en la secuencia ordenada.

En el lugar (*In Place*) Un algoritmo es "*In Place*" si requiere una cantidad acotada de espacio adicional para ordenar los datos, lo que significa que puede ordenar los datos sin necesidad de estructuras auxiliares del tamaño del arreglo original. En otras palabras, el espacio adicional que se necesita no es proporcional a n, la cantidad de elementos a ordenar.

En línea (*Online*) Un algoritmo es "*Online*" si puede procesar los datos a medida que llegan, sin necesidad de conocer el conjunto completo de antemano. Esto es útil en aplicaciones donde los datos se generan en tiempo real y se pueden ir ordenando a medida que llegan.

Existen muchos algoritmos de ordenamiento, cada uno con sus propias fortalezas y debilidades. Algunos de los más conocidos, y a menudo los primeros que se estudian, son los **algoritmos de ordenamiento simple**, por nombrarlos de alguna forma.

Algoritmos de Ordenamiento Simple

Ordenamiento por Inserción (*Insertion Sort*) Construye la lista final ordenada un elemento a la vez. Es eficiente para pequeñas cantidades de datos o datos casi ordenados. Su idea es similar a ordenar cartas en la mano: se toma una carta nueva y se inserta en su posición correcta entre las ya ordenadas. Las ventajas de este algoritmo es que es **Estable**, ***In Place*** y ***Online***. En el mejor de los casos donde el arreglo está casi ordenado, salvo algunos pocos elementos, se puede estimar que el orden es $O(n)$. En el peor caso y en el caso promedio es $O(n^2)$

Ordenamiento por Selección (*Selection Sort*) Encuentra repetidamente el elemento mínimo (o máximo) del resto no ordenado y lo coloca al principio (o al final) de la lista ordenada. Es simple de entender pero no muy eficiente para grandes conjuntos de datos. No es **Estable**, ni ***Online***, pero si ***In Place***. Su complejidad temporal es $O(n^2)$ tanto para el peor, el mejor y el caso promedio.

Ordenamiento por Burbujeo (*Bubble Sort*) Recorre repetidamente la lista, compara elementos adyacentes y los intercambia si están en el orden incorrecto. Las "burbujas" de elementos más grandes (o más pequeños) se mueven hacia el final (o el principio) de la lista. Es un algoritmo muy intuitivo pero extremadamente ineficiente para grandes conjuntos de datos. Es **Estable** e ***In Place***, pero no ***Online***. En el mejor caso cuando los datos están casi ordenados, salvo unos pocos elementos puede presentar una complejidad de $O(n)$. En el peor caso y en el caso promedio es $O(n^2)$.

En la siguiente tabla se resumen las principales características de estos algoritmos:

Algoritmo	Inserción	Selección	Burbujeo
Peor Caso	$O(n^2)$	$O(n^2)$	$O(n^2)$
Caso Promedio	$O(n^2)$	$O(n^2)$	$O(n^2)$
Mejor Caso	$O(n)$	$O(n^2)$	$O(n)$
Estable	Sí	No	Sí
In Place	Sí	Sí	Sí
Online	Sí	No	No

Table 6.22: Algoritmos de Ordenamiento Simple

Ver implementaciones en Go

Go

```go
// OrdenarInsercion ordena el slice en orden ascendente.
// Complejidad: O(n²) peor caso, O(n) mejor caso. Estable, In Place, Online.
func OrdenarInsercion(arr []int) {
    for i := 1; i < len(arr); i++ {
        clave := arr[i]
        j := i - 1
        for j >= 0 && arr[j] > clave {
            arr[j+1] = arr[j]
            j--
        }
        arr[j+1] = clave
    }
}
```

Go

```go
// OrdenarSeleccion ordena el slice en orden ascendente.
// Complejidad: O(n²) en todos los casos. No es estable. In Place.
func OrdenarSeleccion(arr []int) {
    for i := 0; i < len(arr)-1; i++ {
        minIdx := i
        for j := i + 1; j < len(arr); j++ {
            if arr[j] < arr[minIdx] {
                minIdx = j
            }
        }
        arr[i], arr[minIdx] = arr[minIdx], arr[i]
    }
}
```

Go

```go
// OrdenarBurbujeo ordena el slice en orden ascendente.
// Complejidad: O(n²) peor caso, O(n) mejor caso. Estable, In Place.
func OrdenarBurbujeo(arr []int) {
    for i := 0; i < len(arr)-1; i++ {
        intercambiado := false
        for j := 0; j < len(arr)-i-1; j++ {
            if arr[j] > arr[j+1] {
                arr[j], arr[j+1] = arr[j+1], arr[j]
                intercambiado = true
            }
        }
        if !intercambiado {
            break
        }
```

```
    }
}
```

Estos algoritmos son fáciles de implementar y entender, lo que los hace ideales para introducir conceptos básicos de ordenamiento. Sin embargo, su rendimiento se degrada rápidamente a medida que el número de elementos crece, lo que los limita a conjuntos de datos pequeños o casi ordenados. Por ejemplo, el **ordenamiento por inserción** es eficiente para listas pequeñas o listas que ya están casi ordenadas, mientras que el **ordenamiento por selección** y el **ordenamiento por burbujeo** son más adecuados para fines educativos que para aplicaciones prácticas en grandes volúmenes de datos.

Los tres algoritmos tienen una complejidad computacional de $O(n^2)$, donde n es el número de elementos a ordenar. Esto significa que a medida que n crece, el tiempo de ejecución aumenta cuadráticamente, lo que los hace imprácticos para conjuntos de datos grandes. Para superar esta limitación, se han desarrollado algoritmos de ordenamiento más avanzados, a menudo basados en el paradigma "**divide y vencerás**", que explotan la recursión para lograr eficiencias significativamente mayores. Nos centraremos en tres de los algoritmos de ordenamiento más eficientes: **Mergesort, Quicksort y Heapsort**, analizando en detalle su funcionamiento y su complejidad computacional.

Algoritmos de Ordenamiento Recursivo

Los algoritmos de ordenamiento recursivo que analizaremos son ejemplos paradigmáticos de cómo el diseño recursivo puede conducir a soluciones eficientes para problemas complejos. A menudo logran una complejidad de tiempo $O(n \log n)$, lo que representa una mejora sustancial sobre los algoritmos de $O(n^2)$ para grandes conjuntos de datos.

6.6.3.1 Mergesort (Ordenamiento por Mezcla)

Mergesort es un algoritmo **Estable**, que requiere espacio adicional proporcional al tamaño del arreglo, es decir no es ***In Place***, basado en el paradigma "divide y vencerás". Fue creado en 1945 por John von Neumann y es uno de los algoritmos de ordenamiento más eficientes y ampliamente utilizados.

6.6.3.1.1 Funcionamiento

Dividir El algoritmo divide recursivamente la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño hasta que cada sublista contiene solo un elemento (que por definición está ordenado).

Conquistar Cada sublista de un solo elemento se considera ordenada (caso base).

Combinar Las sublistas ordenadas se mezclan repetidamente para producir nuevas sublistas ordenadas hasta que solo quede una sublista, que es la lista original pero ahora completamente ordenada. La operación clave aquí es la **mezcla (merge)** de dos listas ordenadas en una sola lista ordenada.

```text
```

```
FUNCION Mergesort(arreglo)
    SI longitud(arreglo) <= 1 ENTONCES // Caso base: si el arreglo tiene
                                       // 0 o 1 elementos, ya está ordenado
        RETORNAR arreglo
    FIN SI

    mitad ← longitud(arreglo) / 2
    izquierda ← Mergesort(subarreglo desde 0 hasta mitad-1)
    derecha ← Mergesort(subarreglo desde mitad hasta fin)
    RETORNAR merge(izquierda, derecha)
FIN FUNCION
```

Mergesort (Ordenamiento por Mezcla)

text

```
FUNCION merge(izquierda, derecha)
    resultado ← arreglo vacío // Arreglo para almacenar la mezcla
    i ← 0 // Índice para la sublista izquierda
    j ← 0 // Índice para la sublista derecha

    MIENTRAS i < longitud(izquierda) Y j < longitud(derecha) HACER
        SI izquierda[i] <= derecha[j] ENTONCES
            agregar izquierda[i] a resultado
            i ← i + 1
        SINO
            agregar derecha[j] a resultado
            j ← j + 1
        FIN SI
    FIN MIENTRAS

    // Agregar los elementos restantes de izquierda, si los hay
    MIENTRAS i < longitud(izquierda) HACER
        agregar izquierda[i] a resultado
        i ← i + 1
    FIN MIENTRAS

    // Agregar los elementos restantes de derecha, si los hay
    MIENTRAS j < longitud(derecha) HACER
        agregar derecha[j] a resultado
        j ← j + 1
    FIN MIENTRAS

    RETORNAR resultado
FIN FUNCION
```

Función de Mezcla (Merge)

En la línea 2 de la función `merge`, por cada mezcla que se realiza se crea un nuevo arreglo `resultado` que contendrá los elementos ordenados de las dos sublistas. Esto es necesario porque Mergesort no modifica las listas originales, sino que crea una nueva lista ordenada a partir de ellas.

En la línea 7 de la función `merge` es fundamental que se compare por menor igual (<=) para garantizar la estabilidad del algoritmo, es decir, que los elementos iguales mantengan su orden relativo original.

6.6.3.1.2 Análisis de la Complejidad Computacional

La recurrencia para el tiempo de ejecución de Mergesort se puede expresar como: $T(n) = 2T(n/2) + O(n)$ Donde:

- $2T(n/2)$ representa el costo de ordenar las dos sublistas de tamaño $n/2$.
- $O(n)$ representa el costo de la operación de mezcla (merge), ya que en el peor caso se recorren todos los elementos de ambas sublistas una vez para fusionarlas.

Aplicando el Teorema Maestro, se obtiene que la complejidad de tiempo es $\boldsymbol{O(n \log n)}$, ya que subdivide recursivamente el arreglo en mitades hasta llegar al caso base y luego combina los resultados de manera lineal.

En este algoritmo no hay un caso mejor o peor, ya que la división y la mezcla siempre ocurren de la misma manera, lo que da como resultado una complejidad de tiempo consistente.

6.6.3.2 Quicksort (Ordenamiento Rápido)

Quicksort, como **Mergesort**, es un algoritmo de ordenamiento basado en el paradigma "divide y vencerás". Sin embargo, su enfoque es diferente y su eficiencia en la práctica lo ha convertido en uno de los algoritmos de ordenamiento más utilizados. Quicksort **no es un algoritmo de ordenamiento estable** ni ***Online*** pero si es ***In Place*** y estrictamente hablando tampoco se puede asegurar que sea de división y conquista, ya que por la forma de partir el arreglo, no se puede asegurar que las sublistas resultantes sean de tamaño similar. A pesar de esto, es ampliamente considerado como uno de los algoritmos más eficientes para ordenar grandes volúmenes de datos.

La idea principal detrás de Quicksort es seleccionar un elemento de la lista, llamado **pivote**, y reorganizar los elementos de tal manera que todos los elementos menores que el pivote queden a su izquierda y todos los elementos mayores queden a su derecha. Luego, se aplica recursivamente el mismo proceso a las sublistas resultantes. Los elementos a la izquierda y derecha del pivote no están necesariamente ordenados, pero el pivote estará en su posición final ordenada después de la partición.

Esta técnica de partición es lo que distingue a Quicksort de otros algoritmos de ordenamiento, se conoce como **partición de 2 vías**.

Quicksort fue creado y publicado por Tony Hoare en 1961.

6.6.3.2.1 Funcionamiento

Dividir Se elige un elemento de la lista llamado **pivote**. Se reorganiza la lista de tal manera que todos los elementos menores que el pivote queden a su izquierda y todos los elementos mayores queden a su derecha. Los elementos iguales al pivote pueden ir a la izquierda o a la derecha del pivote. Después de esta partición, el pivote se encuentra en su posición final ordenada.

Conquistar Se ordenan recursivamente las dos sublistas (la de elementos menores al pivote y la de elementos mayores al pivote).

Combinar No hay una fase de combinación explícita, ya que la lista se ordena "en el lugar" a medida que las sublistas se ordenan recursivamente. La lista completa queda ordenada una vez que todas las sublistas han sido procesadas.

text

```
FUNCION Quicksort(arreglo, inicio, fin)
    SI inicio >= fin ENTONCES
        RETORNAR // Caso base: si el subarreglo tiene 0 o 1 elementos, ya está
ordenado
    FIN SI
    // Particionar el arreglo y obtener el índice del pivote
    pivote_indice ← Particionar(arreglo, inicio, fin)
    Quicksort(arreglo, inicio, pivote_indice - 1)
    Quicksort(arreglo, pivote_indice + 1, fin)
FIN FUNCION
```

Quicksort (Ordenamiento Rápido)

text

```
  FUNCION Particionar(arreglo, inicio, fin)
      // Elegir el elemento del medio como pivote
      medio ← (inicio + fin) / 2
      pivote_valor ← arreglo[medio]

      // Mover el pivote al final para simplificar la partición
      Intercambiar(arreglo, medio, fin)

      i ← inicio
      j ← fin - 1

      MIENTRAS i <= j HACER
          // Buscar desde la izquierda un elemento mayor que el pivote
          MIENTRAS i <= j Y arreglo[i] <= pivote_valor HACER
              i ← i + 1
          FIN MIENTRAS

          // Buscar desde la derecha un elemento menor que el pivote
          MIENTRAS i <= j Y arreglo[j] >= pivote_valor HACER
              j ← j - 1
          FIN MIENTRAS

          // Si ambos índices no se han cruzado, intercambiar los elementos
          SI i < j ENTONCES
              Intercambiar(arreglo, i, j)
          FIN SI
      FIN MIENTRAS

      // Colocar el pivote en su posición final
      Intercambiar(arreglo, i, fin)

      RETORNAR i // Retorna el índice final del pivote
  FIN FUNCION
```

Partición de 2 vías

text

```
FUNCION Intercambiar(arreglo, i, j)
    temp ← arreglo[i]
    arreglo[i] ← arreglo[j]
    arreglo[j] ← temp
FIN FUNCION
```

Intercambio de elementos

6.6.3.2.2 Pivote

La elección del pivote es crucial para el rendimiento de Quicksort. Estrategias comunes incluyen elegir el primer elemento, el último elemento, el elemento central, o un elemento aleatorio.

Si el pivote elegido es justo el mayor elemento del arreglo (o el menor), la partición resultará en una sublista de tamaño $n-1$ y otra de tamaño 0, lo que llevará a un rendimiento cuadrático en el peor caso.

Una técnica común para mitigar este comportamiento es elegir el pivote como la **"mediana de tres"**, que toma el primer, el último y el elemento del medio del arreglo y selecciona el que está en el medio de esos tres valores. Esto ayuda a evitar particiones muy desequilibradas.

Por ejemplo, si tenemos un arreglo `{...}[3, 6, 8, 10, 1, 2, 1]` la mediana de tres compara el primer elemento `3`, el último elemento `1` y el del medio `10`, y selecciona `3` como pivote, lo que ayuda a evitar un caso de peor rendimiento.

6.6.3.3 Análisis de la Complejidad Computacional

La complejidad temporal de Quicksort depende de la elección del pivote y de cómo se distribuyen los elementos en el arreglo:

Mejor Caso Ocurre cuando el pivote divide el arreglo en dos subarreglos de tamaño aproximadamente igual en cada llamada recursiva. En este caso, la recurrencia es: $T(n) = 2T(n/2) + O(n)$. Aplicando el Teorema Maestro, obtenemos una complejidad de tiempo de $\boldsymbol{O(n \log n)}$.

Caso Promedio En promedio, una buena implementación de Quicksort también logra una complejidad de tiempo de $\boldsymbol{O(n \log n)}$. Es fundamental que la elección del pivote sea aleatoria o que se utilice una estrategia como la mediana de tres para evitar particiones muy desequilibradas.

Peor Caso Ocurre cuando el pivote es el elemento más grande o más pequeño en cada llamada recursiva, lo que lleva a una partición muy desequilibrada. En este caso, la recurrencia es $T(n) = T(n-1) + O(n)$, lo que resulta en una complejidad de tiempo de $\boldsymbol{O(n^2)}$.

Si bien Quicksort tiene un peor caso de $O(n^2)$, este escenario es raro en la práctica si se elige un buen pivote, por lo que su uso es muy común en aplicaciones reales. No se debe usar este algoritmo si tenemos que garantizar que el peor caso sea menor $O(n^2)$, por ejemplo en algunas aplicaciones de tiempo real.

6.6.3.4 Heapsort (Ordenamiento por Montículos)

Heapsort es un algoritmo que no se basa en el paradigma de "división y conquista", sino que utiliza un *heap* o montículo como estructura de datos subyacente para ordenar los elementos.

Heapsort no es estable ni ***Online***, pero si ***In Place*** por lo que no requiere espacio adicional significativo.

Fue desarrollado por J. W. J. Williams en 1964 y es un algoritmo eficiente que garantiza un tiempo de ejecución de $O(n \log n)$ en todos los casos (mejor, promedio y peor).

6.6.3.4.1 Representación del *heap* en un arreglo

Un *heap* es un árbol binario completo que se representa de forma implícita en un arreglo, sin necesidad de punteros explícitos. Para un nodo en la posición i del arreglo:

- Su **padre** (si existe) está en la posición $floor((i-1)/2)$
- Su **hijo izquierdo** (si existe) está en la posición $2i+1$
- Su **hijo derecho** (si existe) está en la posición $2i+2$

Esta representación es muy eficiente en términos de espacio: el árbol y el arreglo son la misma estructura vista de dos formas diferentes. Un *max-heap* cumple la **propiedad de heap**: todo nodo padre es mayor o igual que sus hijos, lo que garantiza que la raíz contiene el máximo.

6.6.3.4.2 Funcionamiento

Heapsort consta de dos fases principales:

Construcción del *Heap* (*Heapify*) Se transforma el arreglo de entrada en un *heap* de máximos. Esto se hace comenzando desde el último nodo no hoja (índice $floor(n/2)-1$) y aplicando la operación *downHeap* (hundir) sobre cada nodo hasta llegar a la raíz. *downHeap* compara un nodo con sus hijos y, si es menor que alguno, lo intercambia con el mayor de ellos, repitiendo el proceso hacia abajo hasta que el nodo quede en posición correcta. Esta fase toma $O(n)$ tiempo.

Extracción de Elementos (*Sort*) Una vez construido el *heap*, la raíz contiene el máximo. Se intercambia la raíz con la última posición del *heap*, se reduce el tamaño del *heap* en uno (el elemento intercambiado queda en su posición final) y se aplica *downHeap* sobre la nueva raíz para restaurar la propiedad de *heap*. Se repite este proceso hasta que el *heap* queda con un solo elemento. Esta etapa es similar al algoritmo de **Selección**.

Importante

Para ordenar de menor a mayor se construye un *max-heap* (la raíz es el mayor). Para ordenar de mayor a menor se construye un *min-heap* (la raíz es el menor).

text

```
FUNCION Heapsort(arreglo)
    n ← longitud(arreglo)

    // Heapify: Construir el _heap_ de máximo
    // En un _heap_ el último nodo que no es hoja es el nodo en la posición n/2 - 1
    // Al hundir cada nodo desde (n/2)-1 hasta 0, se asegura que todos los nodos
    // cumplen la propiedad de heap
    // y por lo tanto el arreglo se convierte en un _heap_ de máximos
    PARA i DESDE (n/2)-1 HASTA 0 HACER
        downHeap(arreglo, n, i)

    // Extraer elementos del _heap_ uno por uno
    PARA i DESDE n - 1 HASTA 1 HACER
        Intercambiar(arreglo, 0, i) // Mover el máximo (raíz del heap) al final del
_heap_.
```

```
        downHeap(arreglo, i, 0) // Restaurar la propiedad de _heap_ en el _heap_
reducido
    FIN PARA
    RETORNAR arreglo // El arreglo ahora está ordenado
FIN FUNCION
```

Heapsort (Ordenamiento por Montículos)

6.6.3.4.3 Análisis de la Complejidad Computacional

La complejidad temporal de Heapsort es $\boldsymbol{O(n \log n)}$ en todos los casos (mejor, promedio y peor). Esto se debe a que la fase de construcción del *heap* toma $O(n)$ tiempo y la etapa de extracción de elementos toma $O(n \log n)$.

6.6.3.4.3.1 Heapify

El proceso de heapify (construir un *heap* a partir de un arreglo desordenado) se realiza de abajo hacia arriba, comenzando desde el último nodo no hoja (índice $floor(n/2) - 1$) y subiendo hasta la raíz (índice 0).

Para cada nodo, se aplica la operación *downHeap*, que asegura que el subárbol con raíz en ese nodo cumpla la propiedad de *heap*. La operación *downHeap* toma un tiempo proporcional a la altura del subárbol, es decir, $O(h)$.

Aunque la altura máxima de un árbol es $\log_2(n)$, la mayoría de los nodos en un árbol binario completo se encuentran en los niveles inferiores (cerca de las hojas), donde la altura es pequeña (uno o dos). A medida que subimos en el árbol, hay menos nodos en cada nivel, pero la altura de sus subárboles aumenta.

La complejidad total de construir un *heap* es la suma de los costos de *downHeap* para todos los nodos. Matemáticamente, esto se puede expresar como:

$$\sum_{h=0}^{floor(\log_2 n)} \frac{n}{2^{h+1}} \cdot h \tag{6.17}$$

Donde h es la altura del nodo, y $\frac{n}{2^{h+1}}$ es el número aproximado de nodos que hay a esa altura. Esta suma se puede simplificar y se demuestra que es $O(n)$. Es decir, el tiempo requerido para convertir un arreglo desordenado en un *heap* es lineal con el número de elementos en el arreglo.

En resumen, a pesar de que una sola operación de *downHeap* puede tomar $O(\log n)$, la suma de todas las operaciones de *downHeap* en el proceso de construcción del *heap* resulta en una complejidad de tiempo lineal $O(n)$, lo que lo hace muy eficiente.

6.6.3.4.3.2 Etapa de Extracción de Elementos

En esta etapa el algoritmo es similar al algoritmo de selección, en el sentido que ambos algoritmos localizan el mayor elemento del arreglo y lo colocan al final del mismo. La diferencia fundamental es que en Heapsort, el mayor elemento siempre está en la posición 0, es decir, en la raíz del árbol, y por lo tanto solo tiene que intercambiarlo con el último elemento del *heap* y hacer un *downHeap* de la nueva raíz, lo que tiene un costo $O(\log n)$. La operación se repite n veces dando como resultado un costo total de $O(n)$ del *heapify*, más $O(n \log n)$ de la etapa de extracción de elementos, quedando $O(n \log n)$.

Comparación y Resumen de Complejidades

Algoritmo	Mergesort	Quicksort	Heapsort
Peor Caso	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Caso Promedio	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Mejor Caso	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Estable	Sí	No	No
In Place	No	Sí	Sí
Online	No	No	No

Table 6.29: Algoritmos de Ordenamiento Recursivos

Como podemos observar, los algoritmos recursivos de "divide y vencerás" (Mergesort, Quicksort) y el basado en estructura de datos eficiente (Heapsort) superan significativamente a los algoritmos simples de $O(n^2)$ para grandes volúmenes de datos, llevando la complejidad a un óptimo $O(n \log n)$. La elección entre ellos dependerá de factores como la disponibilidad de memoria (Mergesort vs. Quicksort/Heapsort), la necesidad de estabilidad, y la garantía de rendimiento en el peor caso.

Importante

Los algoritmos de ordenamiento, basados en comparaciones (es decir que deben comparar cada elemento con los restantes) realizan al menos $n \log(n)$ operaciones simples. Es decir no hay forma de obtener un algoritmo con una mejor cota temporal que $O(n \log n)$ para el peor caso o el caso promedio.

Ejercicios

Los ejercicios de este capítulo están en el directorio `06-ordenamientos-por-comparacion/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

6.7 Ordenamientos en Tiempo Lineal

Los algoritmos de ordenamiento del capítulo anterior (Mergesort, Quicksort, Heapsort) se basan en **comparaciones** entre elementos para determinar su orden. Se puede demostrar que cualquier algoritmo de ordenamiento basado en comparaciones requiere al menos $\Omega(n \log n)$ operaciones en el peor caso[6]. Esto significa que ningún algoritmo de este tipo puede superar esa cota, sin importar cuán ingenioso sea su diseño.

Sin embargo, si explotamos propiedades específicas de los datos —como que los valores pertenecen a un rango acotado, que están uniformemente distribuidos, o que se pueden descomponer en dígitos— podemos diseñar algoritmos que ordenen en **tiempo lineal** $O(n)$, superando la barrera de $\Omega(n \log n)$. La clave está en que estos algoritmos **no comparan** elementos entre sí: usan los valores como índices, los distribuyen en *buckets* o los procesan dígito a dígito.

Vamos a estudiar tres algoritmos de ordenamiento en tiempo lineal: **Ordenamiento por Conteo (*Counting Sort*)**, **Ordenamiento por Urnas (*Bucket Sort*)** y **Radix Sort**.

Ordenamiento por Conteo (*Counting Sort*)

El **Ordenamiento por Conteo** ordena elementos enteros dentro de un rango conocido $[0, k]$. En lugar de comparar elementos, cuenta cuántas veces aparece cada valor y luego reconstruye el arreglo ordenado a partir de esas cuentas. El algoritmo sigue estos pasos:

1. **Inicialización**: Crear un arreglo de conteo de tamaño $k + 1$ e inicializarlo a cero.
2. **Conteo**: Recorrer el arreglo original y para cada elemento incrementar su posición correspondiente en el arreglo de conteo.
3. **Reconstrucción**: Recorrer el arreglo de conteo y escribir cada valor tantas veces como se contó.

text

```
FUNCION CountingSort(arreglo, k)
    conteo ← arreglo de tamaño k+1 inicializado en 0
    n ← longitud(arreglo)

    PARA i ← 0 HASTA n-1 HACER
        valor ← arreglo[i]
        conteo[valor] ← conteo[valor] + 1
    FIN PARA
```

[6] La notación Ω (Omega grande) es la contraparte de O (O grande) que vimos en el capítulo Análisis de Algoritmos. Mientras que O establece una **cota superior** ($T(n) \leq c \cdot f(n)$), Ω establece una **cota inferior** ($T(n) \geq c \cdot g(n)$). Decir que un algoritmo requiere $\Omega(n \log n)$ significa que, para entradas suficientemente grandes, necesita **al menos** del orden de $n \log n$ operaciones; ningún algoritmo basado en comparaciones puede hacerlo mejor en el peor caso.

```
    indice ← 0
    PARA i ← 0 HASTA k HACER
        MIENTRAS conteo[i] > 0 HACER
            arreglo[indice] ← i
            indice ← indice + 1
            conteo[i] ← conteo[i] - 1
        FIN MIENTRAS
    FIN PARA

    RETORNAR arreglo
FIN FUNCION
```

Counting Sort (Ordenamiento por Conteo)

6.7.1.1 Complejidad

La complejidad del **Ordenamiento por Conteo** es $O(n + k)$, donde n es la cantidad de elementos y k es el rango de valores. Es eficiente cuando k es del orden de n o menor. Si k es mucho mayor que n, el arreglo de conteo se vuelve impracticable.

- **Estable**: sí, si se recorre el arreglo original de derecha a izquierda en la fase de reconstrucción (la versión presentada no es estable; la variante estable acumula sumas prefijo para determinar posiciones finales).
- ***In Place***: no, requiere espacio adicional $O(k)$ para el arreglo de conteo.

Ordenamiento por Urnas (*Bucket Sort*)

El **Ordenamiento por Urnas** distribuye los elementos en varios *buckets* (urnas o cubos), ordena cada *bucket* individualmente y luego concatena los resultados. Es especialmente útil cuando los datos están uniformemente distribuidos en un rango conocido.

1. **División en *buckets***: Dividir el rango de los datos en k intervalos.
2. **Distribución**: Colocar cada elemento en el *bucket* correspondiente según su valor.
3. **Ordenamiento intra-bucket**: Ordenar cada *bucket* individualmente (típicamente con Inserción o Counting Sort si los datos lo permiten).
4. **Concatenación**: Concatenar los *buckets* en orden para obtener el arreglo ordenado.

text

```
FUNCION BucketSort(arreglo, k)
    buckets ← arreglo de k buckets vacíos
    n ← longitud(arreglo)
    max ← máximo valor en arreglo
    min ← mínimo valor en arreglo
    rango ← (max - min + 1) / k

    PARA i ← 0 HASTA n-1 HACER
        indice ← PISO((arreglo[i] - min) / rango)
        SI indice >= k ENTONCES
            indice ← k - 1
```

```
                    FIN SI
                    agregar arreglo[i] a buckets[indice]
                FIN PARA

                PARA i ← 0 HASTA k-1 HACER
                    Ordenar(buckets[i])   // Ej: Inserción
                FIN PARA

                resultado ← arreglo vacío
                PARA i ← 0 HASTA k-1 HACER
                    PARA cada elemento en buckets[i] HACER
                        agregar elemento a resultado
                    FIN PARA
                FIN PARA

                RETORNAR resultado
            FIN FUNCION
```

Bucket Sort (Ordenamiento por Urnas)

6.7.2.1 Complejidad

La complejidad del **Ordenamiento por Urnas** es $O(n + k)$ en promedio, donde n es la cantidad de elementos y k la cantidad de *buckets*. En el peor caso, si todos los elementos caen en un mismo *bucket*, la complejidad es la del algoritmo de ordenamiento intra-bucket (típicamente $O(n^2)$ si se usa Inserción).

- **Estable**: sí, si el algoritmo intra-bucket es estable.
- ***In Place***: no, requiere espacio para los *buckets*.

Radix Sort

El **Radix Sort** ordena elementos procesándolos dígito por dígito, desde el menos significativo (LSD) hasta el más significativo (MSD). Utiliza un algoritmo de ordenamiento **estable** (como Counting Sort o Bucket Sort) como subrutina para ordenar por cada dígito.

1. **Identificación del dígito menos significativo**: Comenzar con la posición de dígito $d = 0$ (unidades).
2. **Ordenamiento estable**: Ordenar los elementos según el dígito en la posición d usando un algoritmo estable.
3. **Repetir**: Incrementar d y repetir hasta procesar todos los dígitos del valor más grande.

text

```
FUNCION RadixSort(arreglo)
    max ← máximo valor en arreglo
    exp ← 1

    MIENTRAS max / exp > 0 HACER
        CountingSortPorDigito(arreglo, exp)
        exp ← exp * 10
    FIN MIENTRAS
```

```
    RETORNAR arreglo
FIN FUNCION

FUNCION CountingSortPorDigito(arreglo, exp)
    n ← longitud(arreglo)
    salida ← arreglo de tamaño n
    conteo ← arreglo de tamaño 10 inicializado en 0

    PARA i ← 0 HASTA n-1 HACER
        digito ← PISO(arreglo[i] / exp) MOD 10
        conteo[digito] ← conteo[digito] + 1
    FIN PARA

    // Sumas prefijo para hacerlo estable
    PARA i ← 1 HASTA 9 HACER
        conteo[i] ← conteo[i] + conteo[i - 1]
    FIN PARA

    // Recorrer de derecha a izquierda para estabilidad
    PARA i ← n-1 HASTA 0 PASO -1 HACER
        digito ← PISO(arreglo[i] / exp) MOD 10
        salida[conteo[digito] - 1] ← arreglo[i]
        conteo[digito] ← conteo[digito] - 1
    FIN PARA

    // Copiar salida en arreglo original
    PARA i ← 0 HASTA n-1 HACER
        arreglo[i] ← salida[i]
    FIN PARA
FIN FUNCION
```

Radix Sort (LSD)

6.7.3.1 Complejidad

La complejidad del **Radix Sort** es $O(n \times d)$, donde n es la cantidad de elementos y d la cantidad de dígitos del valor más grande. Si d es constante (por ejemplo, enteros de 32 bits), la complejidad es $O(n)$.

- **Estable**: sí, porque usa una subrutina estable en cada paso.
- ***In Place***: no, requiere espacio auxiliar para la subrutina estable.

Comparación

Algoritmo	Complejidad	Estable	In Place	Requisito
Counting Sort	$O(n + k)$	Sí*	No	Rango acotado $[0, k]$
Bucket Sort	$O(n + k)$	Sí**	No	Distribución uniforme
Radix Sort	$O(n \times d)$	Sí	No	Valores descomponibles en dígitos

Table 6.33: Algoritmos de Ordenamiento Lineal

* La variante estable requiere recorrer de derecha a izquierda con sumas prefijo. ** Si el algoritmo intra-bucket es estable.

Ejercicios

Los ejercicios de este capítulo están en el directorio `07-ordenamientos-lineales/ejercicios/` del repositorio `taller-algoritmos`. Cada ejercicio tiene un esqueleto con `// TODO` y su correspondiente batería de tests. Para resolverlos, clonar el repositorio, completar las funciones y ejecutar `go test ./...`.

7. Anexos

7.1 Introducción a Git y GitHub

Cuando trabajamos en un proyecto, es común que necesitemos compartir nuestro trabajo con otros. Para ello, es necesario contar con una herramienta que nos permita llevar un registro de los cambios realizados, y que nos permita compartirlos con otros.

Cuando trabajamos en equipo en un proyecto surgen muchas preguntas:

- ¿Cómo **sincronizamos** el trabajo?
- ¿Cómo **dividimos** las partes a realizar?
- ¿Cómo volvemos a **juntarlas**?
- ¿Qué sucede si la **persona** que iba a juntar todo, no llega a hacerlo?
- ¿Qué hacemos si hay que volver a una **versión anterior**?
- ¿Cómo hacemos **experimentos** sobre el trabajo?

Para responder a estas preguntas, existen herramientas que nos permiten llevar un registro de los cambios realizados en un proyecto y compartirlos con otros. Una de las herramientas más utilizadas es **Git**.

Git es un sistema de control de versiones (SCV).

¿Qué es un SCV?

Un sistema de control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

> **Analogía**
>
> Imaginemos que Git es una cámara de fotos. En este caso, la analogía es muy buena, ya que Git es la herramienta, y no "la cosa" que haremos.

Configuración inicial de Git

Antes de empezar a usar Git, es necesario configurar tu nombre y correo electrónico. Git los usa para firmar cada uno de tus *commits*:

console

```
~ $ git config --global user.name "Tu Nombre"

~ $ git config --global user.email "tuemail@ejemplo.com"
```

Este paso se hace una sola vez. Git asocia esos datos a cada *commit* que crees. Se puede verificar la configuración con:

console

```
~ $ git config --global --list
user.name=Tu Nombre
user.email=tuemail@ejemplo.com
```

Creando un repositorio

El primer paso, es la creación de un repositorio Git. Nótese que no es “un Git”, sino un repositorio (repo). Entonces, dentro de la carpeta donde queremos crear nuestro repo, ejecutamos el siguiente comando:

console

```
~ $ mkdir ayp2-repo

~ $ cd ayp2-repo

~/ayp2-repo $ git init
Initialized empty Git repository in ~/ayp2-repo/.git/
```

Antes de comenzar a hacer cambios, queremos ver el estado de nuestro repo:

console

```
~/ayp2-repo $ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

¡Consejo!

> Es muy importante siempre prestar atención a los mensajes que los distintos comandos de Git nos devuelven. En este caso, nos dice que no hay nada para hacer, ya que no hemos creado nada aún.
>
> En general, son comentarios útiles y nos ayudan a saber que posibilidades tenemos desde donde estamos parados.

Ahora podemos empezar a trabajar en nuestro proyecto. Cuando hayamos creado una porción de código que consideramos suficiente, debemos indicar cuáles de esos cambios queremos dejar asentados en el registro de cambios de nuestro repo.

Registrando cambios

Imaginemos que hemos creado un archivo llamado `ejercicios.go` y hemos escrito un par de líneas de código. Si ahora ejecutamos `git status`, veremos que el archivo no está en el registro de cambios, por lo que debemos primero **agregarlos**.

console

```
~/ayp2-repo $ nvim ejercicios.go

~/ayp2-repo $ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ejercicios.go

nothing added to commit but untracked files present (use "git add" to track)
```

Podemos ver que Git no solo nos cuenta cuál es el estado de nuestros cambios, sino que también nos dice qué comando deberíamos usar para agregar el archivo al registro de cambios, entonces hagamos eso mismo:

console

```
~/ayp2-repo $ git add ejercicios.go

~/ayp2-repo $ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   ejercicios.go
```

Ahora nuestro archivo pasó del grupo de los *Untracked files* a *Changes to be committed*. Esto significa que ya está preparado para ser registrado en el repo.

Estamos listos para dejar registro de nuestro primer cambio. Como lo que hacemos es registrar la historia de cómo nuestro código va evolucionando, debemos llevar una bitácora que nos servirá de guía para saber qué hicimos en cada momento. En este caso, la bitácora es un mensaje que describe el cambio que estamos haciendo.

console

```
~/ayp2-repo $ git commit -m "Primera versión de ejercicios.go"
[main (root-commit) a068b64] Primera versión de ejercicios.go
 1 file changed, 5 insertions(+)
 create mode 100644 ejercicios.go
```

Si ahora volvemos a consultar el estado del repo (cosa que como ya se habrán dado cuenta, se hace muy seguido), podemos ver que ya no hay cambios pendientes de registrar y nuestro *working tree* está limpio.

console

```
~/ayp2-repo $ git status
On branch main
nothing to commit, working tree clean
```

Cuando "commiteamos" los cambios indicamos un mensaje, ese mensaje junto con información relacionada al commit generado puede ser consultado usando:

console

```
~/ayp2-repo $ git log
commit a068b643d8b6bb6b8ba51d51e919694dd665ed36 (HEAD -> main)
Author: Santiago Rojo <tiagox@gmail.com>
Date:   Fri Mar 28 18:38:18 2025 -0300

    Primera versión de ejercicios.go
```

En resumen

Desde aquí en adelante, cada vez que hagamos un cambio en el repo, debemos seguir los mismos pasos:

1. Editar el archivo
2. Agregarlo al registro de cambios: `git add <archivo>`
3. Registrar el cambio con un mensaje: `git commit -m "<mensaje>"`

… volver al principio.

7.1.4.1 Ignorando archivos

No todos los archivos de un proyecto deberían estar en el repositorio. Archivos ejecutables, binarios compilados o archivos temporales del editor no forman parte del código fuente. Para indicarle a Git qué archivos ignorar, se crea un archivo `.gitignore` en la raíz del repositorio:

text

```
# Compilados
*.exe
*.out

# Archivos temporales
*.tmp
*.log
```

Cada línea del `.gitignore` es un patrón de archivos que Git no va a trackear. Esto mantiene el repositorio limpio y evita subir archivos innecesarios.

Trabajando en equipo

7.1.5.1 *Branches*

Las *branches* o ramas, nos permiten trabajar en paralelo en un proyecto. Podríamos crear ramas paralelas de desarrollo en donde hacer pruebas o cambios experimentales, para luego decidir si son cambios que queremos integrar a nuestra rama principal o no.

Para crear una nueva rama, usamos el comando:

console

```
~/ayp2-repo $ git switch -c pruebas
Switched to a new branch 'pruebas'
```

Esto crea una nueva rama llamada `pruebas` y nos "cambia" a esa rama. Si ahora hacemos `git status`, veremos que estamos en la rama `pruebas`.

console

```
~/ayp2-repo $ git status
On branch pruebas
nothing to commit, working tree clean
```

Para listar las *branches* que tenemos, usamos el comando `git branch`:

console

```
~/ayp2-repo $ git branch
  main
* pruebas
```

El asterisco indica en qué rama estamos parados. En este caso, en la rama `pruebas`.

Para movernos entre las distintas *branches*, usamos el mismo comando `git switch`. Por ejemplo, si quisiéramos volver a la rama principal, usaríamos:

console

```
~/ayp2-repo $ git switch main
Switched to branch 'main'
```

El flujo de trabajo es el mismo que comentamos al principio de este capítulo. Primero editamos el archivo, luego lo agregamos al registro de cambios y finalmente lo registramos con un mensaje.

7.1.5.2 Integrando los cambios de nuestra rama

Si quisiéramos incorporar la *branch* `pruebas` que creamos anteriormente, debemos utilizar el comando `git merge`.

Primero debemos estar parados sobre la *branch* donde queremos integrar los cambios. En este caso, la rama principal `main`.

console

```
~/ayp2-repo $ git switch main
Switched to branch 'main'

~/ayp2-repo $ git merge pruebas
Updating a068b64..13495a7
Fast-forward
 ejercicios.go | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

7.1.5.3 Resolviendo conflictos

Si dos ramas modifican las mismas líneas de un archivo, Git no puede decidir automáticamente qué versión conservar. En ese caso, el *merge* se pausa y Git marca el conflicto en el archivo:

console

```
~/ayp2-repo $ git merge pruebas
Auto-merging ejercicios.go
CONFLICT (content): Merge conflict in ejercicios.go
Automatic merge failed; fix conflicts and then commit the result.
```

Al abrir el archivo conflictivo se ven marcadores que indican los cambios enfrentados:

Go

```
funcion := "saludar"
<<<<<<< HEAD
fmt.Println("Hola")
=======
fmt.Println("Hello")
>>>>>>> pruebas
```

La sección entre `<<<<<<< HEAD` y `=======` es el cambio en la rama actual. La sección entre `=======` y `>>>>>>> pruebas` es el cambio en la rama `pruebas`. Para resolver, hay que editar el archivo, quedarse con una de las versiones (o una combinación), eliminar los marcadores y luego:

console

```
~/ayp2-repo $ git add ejercicios.go

~/ayp2-repo $ git commit -m "Resuelve conflicto entre main y pruebas"
```

Git registra el *commit* de *merge* con la resolución.

7.1.5.4 Clonando un repositorio existente

Si ya existe un repositorio en GitHub (o cualquier otro servidor), no hace falta crearlo desde cero con `git init`. Se puede obtener una copia local completa con `git clone`:

console

```
~ $ git clone https://github.com/usuario/repositorio.git

~ $ cd repositorio

~/repositorio $ git status
On branch main
nothing to commit, working tree clean
```

Esto descarga todo el historial del proyecto y deja el repo listo para trabajar.

7.1.5.5 Configurando un repositorio remoto

Para compartir un repositorio local a través de GitHub, primero hay que asociarlo con un remoto:

console

```
~/ayp2-repo $ git remote add origin https://github.com/tuusuario/ayp2-repo.git
```

El nombre `origin` es una convención que identifica al remoto principal.

7.1.5.6 Subiendo cambios

Una vez configurado el remoto, se pueden enviar los *commits* locales:

console

```
~/ayp2-repo $ git push -u origin main
```

El flag `-u` (o `--set-upstream`) vincula la rama local `main` con la remota, de modo que en adelante alcance con escribir solo `git push`.

7.1.5.7 Descargando cambios

Para traer los últimos cambios del remoto al repositorio local:

console

```
~/ayp2-repo $ git pull
```

`git pull` combina dos operaciones: descarga los cambios del remoto (`git fetch`) y los integra en la rama actual (`git merge`). Es el comando que se usa para mantenerse al día con el trabajo de otros miembros del equipo.

Flujo de trabajo típico

1. `git status` — ver el estado actual del repositorio
2. `git add <archivo>` — preparar los cambios para el *commit*
3. `git commit -m "<mensaje>"` — registrar los cambios
4. `git push` — enviar los *commits* al remoto (si existe)
5. `git pull` — traer los cambios del remoto (si se trabaja en equipo)

Links recomendados

- Git-Book

7.2 Iteradores en Árboles Binarios de Búsqueda (ABB)

En este anexo, vamos a implementar iteradores para un Árbol Binario de Búsqueda (ABB) en Go. Los iteradores nos permiten recorrer los nodos del árbol de manera ordenada, facilitando la obtención de los valores almacenados. Elegimos implementarlos sobre un ABB para aprovechar su estructura ordenada.

Interfaz del Iterador

En el siguiente fragmento de código definimos la interfaz `Iterator` que contendrá los métodos necesarios para iterar cualquier colección de elementos y en particular los nodos de un ABB.

Go

```go
// La interfaz Iterator define los métodos para iterar sobre una colección
type Iterator[T any] interface {
    HasNext() bool
    Next() (T, error)
}
```

Los métodos que debe implementar un iterador son los siguientes

`HasNext` Indica si hay un siguiente elemento en la colección. Devuelve `true` si hay más elementos por recorrer, o `false` si se ha llegado al final de la colección.

`Next` Devuelve el siguiente elemento de la colección. El comportamiento de `Next` consiste en avanzar al próximo elemento y devolverlo. Si no hay más elementos para iterar devuelve un elemento nulo y un error.

Árbol Binario de Búsqueda (ABB)

En esta implementación de ABB la funcionalidad del árbol se encuentra en el árbol propiamente dicho y no en los nodos, con el propósito de simplificar el código y mejorar la legibilidad.

Los métodos incluidos son los mínimos y necesarios para crear un árbol, insertar elementos y obtener los distintos iteradores.

El árbol usa un tipo genérico con una función de comparación, lo que permite usar cualquier tipo que provea un criterio de orden.

A continuación se muestra la estructura del nodo del ABB.

Go

```go
// BinaryNode representa un nodo en el árbol binario
type BinaryNode[T any] struct {
    value T
    left  *BinaryNode[T]
    right *BinaryNode[T]
}
```

El tipo `BinaryNode` solo tiene los métodos necesarios para crear un nodo y obtener su valor. No contiene métodos para insertar o eliminar nodos, ya que toda la funcionalidad del árbol está implementada en `BinarySearchTree` directamente.

En la implementación del ABB tenemos los métodos para obtener los iteradores:

Go

```go
// BinarySearchTree implementa un árbol binario de búsqueda genérico
type BinarySearchTree[T any] struct {
    root *BinaryNode[T]
    cmp  func(T, T) int
}

// NewBinarySearchTree crea un nuevo ABB con la función de comparación dada
func NewBinarySearchTree[T any](cmp func(T, T) int) *BinarySearchTree[T] {
    return &BinarySearchTree[T]{cmp: cmp}
}

// Insert agrega un valor al árbol
func (t *BinarySearchTree[T]) Insert(value T) { ... }

// InorderIterator retorna un iterador inorder
func (t *BinarySearchTree[T]) InorderIterator() Iterator[T] { ... }
```

Al crear un iterador se ejecuta un setup inicial que depende de cada tipo de iterador.

Implementación de los iteradores. El Desafío de Mantener el Estado: ¿Por qué no solo recursión?

Cuando estudiamos los recorridos de árboles (inorder, preorder, postorder), la forma más intuitiva y elegante de implementarlos es usando recursión, ya que el código queda conciso, limpio y fácil de entender.

Sin embargo, cuando hablamos de un iterador, la situación cambia. Un iterador, por definición (según el patrón Iterador), debe permitirnos obtener el “siguiente” elemento de la colección a demanda, sin tener que procesar el resto de la colección de antemano. Es decir, queremos que nuestro método `Next()` devuelva un único valor y luego `HasNext()` nos diga si hay más, esperando una nueva llamada a `Next()`.

Aquí es donde la recursión directa presenta un desafío:

Manejo del estado Una función recursiva completa su ejecución y devuelve un resultado. No "pausa" su estado para ser reanudada en el mismo punto más tarde. Cada llamada recursiva crea un nuevo *stack frame* (marco de pila) que se destruye al finalizar.

Devolución de un solo elemento Si intentáramos adaptar la recursión para devolver un solo elemento en cada llamada a `Next()`, nos encontraríamos con que la función recursiva ya ha avanzado en el recorrido, y sería muy difícil mantener el "punto exacto" en el que nos quedamos para la siguiente llamada.

Espacio en memoria (*stack overflow*) Para árboles muy grandes o muy desbalanceados (degenerados), una implementación recursiva podría consumir una gran cantidad de memoria de la pila de llamadas, llevando a un error de stack overflow.

Para superar estos desafíos y crear un iterador que sea *lazy* (bajo demanda) y eficiente en memoria, necesitamos una forma de simular la pila de llamadas recursiva de forma explícita. Y la estructura de datos perfecta para esto es, precisamente, una pila (*stack*).

> **Nota**
>
> Que sea *lazy* significa que el iterador no calcula todos los elementos de antemano, sino que los obtiene a medida que se le solicita un nuevo elemento. Esto es fundamental para manejar colecciones grandes o infinitas sin consumir demasiada memoria.

7.2.3.1 La Pila como Sustituto de la Recursión Explícita

Al usar una pila en nuestros iteradores, nosotros somos los que gestionamos el "estado" de la recursión. En lugar de que el sistema operativo maneje la pila de llamadas por nosotros, nosotros empujamos (*Push*) y sacamos (*Pop*) nodos de nuestra propia estructura de pila para recordar qué camino hemos tomado y qué nodos nos quedan por visitar.

Esta técnica nos permite:

Pausar y Reanudar Cuando `Next()` es llamado, podemos extraer el siguiente nodo a visitar de la pila, devolverlo, y luego dejar la pila en un estado que representa el "resto" del recorrido. La próxima vez que se llame `Next()`, la pila nos recordará dónde estábamos.

Eficiencia en Memoria La profundidad de la pila que necesitamos es proporcional a la altura del árbol ($O(h)$), no al número total de nodos ($O(N)$), lo que es mucho más eficiente para árboles grandes y balanceados ($O(\log N)$).

Control Explícito Tenemos un control total sobre el orden en que los nodos se agregan y se eliminan de la pila, lo que es fundamental para implementar correctamente los diferentes tipos de recorridos.

Iterador Inorder

La estrategia para el iterador inorder es la siguiente:

1. Comenzamos desde la raíz del árbol.
2. Vamos hacia la izquierda hasta llegar al nodo más a la izquierda apilando los nodos en la pila.
3. El menor elemento del árbol será el nodo más a la izquierda es el que se encuentra en el tope de la pila.

4. Cuando llamamos a `Next()`, sacamos el nodo del tope de la pila, chequeamos si tiene un hijo derecho. Si lo tiene, vamos a ese hijo derecho y repetimos el proceso de ir hacia la izquierda, apilando los nodos.
5. Una vez que no hay más nodos a la izquierda, el tope de la pila contendrá el siguiente nodo en orden.
6. Devolvemos el elemento del nodo que acabamos de sacar de la pila.

En el siguiente fragmento de código se puede observar la implementación del iterador Inorder

El setup inicial consiste en apilar la raíz y toda la rama izquierda para iniciar. De esta forma el primer nodo que se desapila es el menor de todo el árbol

Go

```go
// InorderIterator implementa un iterador inorder
type InorderIterator[T any] struct {
    stack []*BinaryNode[T]  // Pila principal para el recorrido
}

func (t *BinarySearchTree[T]) InorderIterator() Iterator[T] {
    iter := &InorderIterator[T]{stack: make([]*BinaryNode[T], 0)}
    // Setup: apilar raíz y toda la rama izquierda
    node := t.root
    for node != nil {
        iter.stack = append(iter.stack, node)
        node = node.left
    }
    return iter
}

func (iter *InorderIterator[T]) HasNext() bool {
    return len(iter.stack) > 0
}

func (iter *InorderIterator[T]) Next() (T, error) {
    // Implementación del Next...
    var zero T
    return zero, nil
}
```

Iterador Preorder

La estrategia para el iterador preorder varía un poco, ya que lo primero que debemos listar es la raíz, por lo tanto el setup inicial consiste en apilar sólo la raíz

1. Comenzamos desde la raíz del árbol.
2. Apilamos sólo la raíz
3. Al desapilar un nodo cuando se llama a `Next()`, se apilan primero su hijo derecho y luego su hijo izquierdo (para que el orden de salida de la pila sea primero el izquierdo y luego el derecho). El próximo en salir de la pila será la raíz del subárbol izquierdo.
4. Devolvemos el elemento del nodo que acabamos de sacar de la pila.

5. Se repite hasta que la pila queda vacía.

En el siguiente fragmento de código se puede observar la implementación del iterador Preorder

El setup inicial consiste en apilar la raíz solamente.

Go

```go
// PreorderIterator implementa un iterador preorder
type PreorderIterator[T any] struct {
    stack []*BinaryNode[T]
}

func (t *BinarySearchTree[T]) PreorderIterator() Iterator[T] {
    iter := &PreorderIterator[T]{stack: make([]*BinaryNode[T], 0)}
    if t.root != nil {
        iter.stack = append(iter.stack, t.root)
    }
    return iter
}

func (iter *PreorderIterator[T]) HasNext() bool {
    return len(iter.stack) > 0
}

func (iter *PreorderIterator[T]) Next() (T, error) {
    // Implementación...
    var zero T
    return zero, nil
}
```

Iterador Postorder

Para implementar el iterador Postorder se necesitan 2 pilas.

El recorrido postorder visita los nodos en el orden: izquierda, derecha, raíz. Sin embargo, si usamos una sola pila y simplemente apilamos los hijos izquierdo y derecho, no es sencillo garantizar este orden sin usar recursión.

Por eso, se usan dos pilas para simular el recorrido postorder de manera iterativa, la primera pila se usa para recorrer el árbol y la segunda para almacenar los nodos en postorder. La inicialización del iterador implica recorrer todo el árbol y apilar los nodos en la segunda pila. Luego se pueden desapilar los nodos de la segunda pila a demanda.

- **Setup inicial:**
  - Se apila la raíz en la primera pila.
  - Mientras stack1 no esté vacía:
    - Se desapila un nodo de stack1 y se apila en la segunda pila (stack2).
    - Si el nodo tiene hijo izquierdo, se apila en stack1.
    - Si el nodo tiene hijo derecho, se apila en stack1.

Así, stack1 sirve para recorrer el árbol y stack2 va guardando los nodos en un orden tal que, al desapilar de stack2, obtenemos el recorrido postorder.

- **Iteración:**
- Cuando se llama a `Next()`, simplemente se desapila un nodo de stack2 y se devuelve su valor.
- `HasNext()` verifica si stack2 aún tiene nodos.

En el siguiente fragmento de código se puede observar la implementación del iterador Postorder

El setup inicial consiste en apilar la raíz en la primera pila y luego procesar el árbol para llenar la segunda pila en orden postorder.

Go

```go
// PostorderIterator implementa un iterador postorder usando dos pilas
type PostorderIterator[T any] struct {
    stack1 []*BinaryNode[T]
    stack2 []*BinaryNode[T]
}

func (t *BinarySearchTree[T]) PostorderIterator() Iterator[T] {
    iter := &PostorderIterator[T]{
        stack1: make([]*BinaryNode[T], 0),
        stack2: make([]*BinaryNode[T], 0),
    }
    if t.root != nil {
        iter.stack1 = append(iter.stack1, t.root)
    }
    for len(iter.stack1) > 0 {
        node := iter.stack1[len(iter.stack1)-1]
        iter.stack1 = iter.stack1[:len(iter.stack1)-1]
        iter.stack2 = append(iter.stack2, node)
        if node.left != nil {
            iter.stack1 = append(iter.stack1, node.left)
        }
        if node.right != nil {
            iter.stack1 = append(iter.stack1, node.right)
        }
    }
    return iter
}

func (iter *PostorderIterator[T]) HasNext() bool {
    return len(iter.stack2) > 0
}

func (iter *PostorderIterator[T]) Next() (T, error) {
    // Implementación...
    var zero T
    return zero, nil
}
```

Código completo en Go

El código completo de este anexo está disponible en el repositorio guia-iteradores-abb.

7.3 Herramientas para Cursar

En esta materia vamos a usar varias herramientas: un editor de código, un lenguaje de programación, un sistema de control de versiones y una plataforma para recibir y entregar trabajos. Este anexo explica cómo instalar y usar cada una.

> **Importante**
>
> La mayoría de las instrucciones asumen que empezás desde cero, sin nada instalado.

Instalación de Git y Git Bash

Git es el sistema de control de versiones que usamos para gestionar el código. En Windows, al instalar Git se incluye **Git Bash**, una terminal que funciona como la de Linux.

Windows

1. Ir a https://git-scm.com y descargar el instalador.
2. Ejecutar el `.exe` y dejar las opciones por defecto.
3. Al finalizar, abrir **Git Bash** desde el menú Inicio.
4. Verificar la instalación:

console

```
git --version
```

output

```
git version 2.4x.x
```

Linux

console

```
sudo apt install git
```

O el comando equivalente según la distribución (pacman, dnf, etc.). Verificar:

console

```
git --version
```

output

```
git version 2.4x.x
```

macOS

Opción A — Xcode Command Line Tools (incluye Git):

console

```
xcode-select --install
```

Opción B — Descargar desde https://git-scm.com.

Verificar:

console

```
git --version
```

output

```
git version 2.4x.x
```

Terminal / línea de comandos

La terminal permite ejecutar comandos para navegar archivos, correr programas y usar Git. En Windows usamos **Git Bash**; en Linux y macOS la terminal nativa.

Los comandos básicos son los mismos en los tres sistemas:

Comando	Qué hace
`pwd`	Muestra el directorio actual
`ls`	Lista los archivos y carpetas
`cd <carpeta>`	Cambia al directorio indicado
`cd ..`	Sube un nivel
`mkdir <nombre>`	Crea una carpeta
`touch <archivo>`	Crea un archivo vacío

Tab	Autocompleta comandos y rutas
⬆️ ⬇️	Navega el historial de comandos

Tip

En Git Bash (Windows) los comandos son los mismos que en Linux y macOS. No hace falta aprender comandos diferentes para cada sistema.

Estructura de rutas:

- **Windows (Git Bash):** `/c/Users/TuUsuario/` equivale a `C:\Users\TuUsuario\`
- **Linux:** `/home/tu-usuario/`
- **macOS:** `/Users/tu-usuario/`

Instalación de Go

Necesitamos Go para compilar y ejecutar los ejercicios de los talleres.

Windows

1. Ir a https://go.dev/dl y descargar el `.msi` para Windows.
2. Ejecutar el instalador. Agrega Go al PATH automáticamente.
3. Abrir **Git Bash** y verificar:

console

```
go version
```

output

```
go version go1.22.x windows/amd64
```

Linux

console

```
sudo apt install golang
```

O descargar el tarball oficial desde https://go.dev/dl y seguir las instrucciones de instalación.

Verificar:

console

```
go version
```

output

```
go version go1.22.x linux/amd64
```

macOS

Opción A — Descargar el `.pkg` desde https://go.dev/dl.

Opción B — Con Homebrew:

console

```
brew install go
```

Verificar:

console

```
go version
```

output

```
go version go1.22.x darwin/amd64
```

Visual Studio Code

VS Code es el editor que usamos en la cursada. Es liviano y tiene soporte para Go con las extensiones adecuadas.

Windows

1. Ir a https://code.visualstudio.com y descargar el instalador.
2. Ejecutar el `.exe`. Marcar **"Agregar a PATH"** durante la instalación.
3. Abrir VS Code desde el menú Inicio.

Linux

Descargar el `.deb` o `.rpm` desde https://code.visualstudio.com, o usar el gestor de paquetes:

```
console
sudo snap install code --classic
```

macOS

Descargar el `.zip` desde https://code.visualstudio.com, o con Homebrew:

```
console
brew install --cask visual-studio-code
```

Extensiones de VS Code

Para trabajar con Go necesitamos algunas extensiones. Desde VS Code:

1. Ir al icono de extensiones (cuadrados en la barra izquierda, o `Ctrl+Shift+X`).
2. Buscar cada extensión por nombre o ID.
3. Click en **Install**.

Extensión	ID	Para qué sirve
Go	`golang.go`	Autocompletado, formato, navegación, análisis de código y explorador visual de tests
Error Lens	`usernamehw.errorlens`	Muestra los errores inline, al lado del código
GitHub Pull Requests	`GitHub.vscode-pull-request-github`	Revisar y comentar Pull Requests (incluyendo el Feedback PR)

GitHub Codespaces

Codespaces es un entorno de desarrollo en la nube de GitHub: te da un VS Code completo con Git y Go ya instalados, directamente desde el navegador. Es una buena alternativa si no podés (o no querés) instalar nada en tu computadora.

7.3.6.1 Cómo abrirlo

1. Ir al repositorio de la tarea en github.com.
2. Click en el botón verde **Code**.
3. Ir a la pestaña **Codespaces**.
4. Click en **Create codespace on main**.

Después de unos segundos se abre un VS Code en el navegador, con el repositorio ya clonado y listo para editar, correr tests y usar la terminal integrada.

7.3.6.2 Limitaciones a tener en cuenta

Importante

Las cuentas gratuitas de GitHub incluyen **60 horas de Codespaces por mes**. El tiempo se consume mientras el codespace está activo, así que conviene **detenerlo** cuando terminás de trabajar: cerrar la pestaña del navegador no lo detiene.

Para detenerlo tenés dos opciones:

- **Desde adentro del codespace:** click en el menú de GitHub Codespaces abajo a la izquierda de la barra de estado (o `Ctrl+Shift+P`) -> **Stop Current Codespace**.
- **Desde github.com:** ir a https://github.com/codespaces y, a la derecha del codespace que esté corriendo (`Active`), click en los tres puntos (**…**) -> **Stop codespace**. Esta opción solo aparece mientras el codespace está corriendo; si ya está detenido vas a ver otras opciones como **Delete** o **Rebuild container**.

Si no lo detenés manualmente, se detiene solo después de 30 minutos de inactividad (configurable), pero ese tiempo sigue consumiendo horas.

Además, trabajar en Codespaces **no guarda nada automáticamente en GitHub**: los cambios quedan solo en el entorno en la nube. Para que impacten en el repositorio hay que hacer **commit y push** desde la terminal integrada, igual que en cualquier otro entorno:

console

```
git add .
git commit -m "mensaje"
git push
```

Para las instancias finales recomendamos tener el entorno local instalado; Codespaces es una alternativa para la cursada diaria.

Slack

Usamos Slack para la comunicación del curso: anuncios, consultas, canales por tema.

1. Crear una cuenta en https://slack.com si no tenés una.
2. Unirte al workspace del curso con el enlace que te dan los docentes.
3. Explorar los canales: `#general` (anuncios), `#consultas`, `#taller-go`, etc.
4. Para consultar a un docente o ayudante, escribir `@usuario` en cualquier canal.
5. Las notificaciones llegan en la aplicación y por correo.

Classroom50 y gh CLI

Classroom50 es la plataforma que usamos para recibir y entregar los trabajos. Se accede por línea de comandos con `gh` (GitHub CLI) o por web.

La materia se dicta en dos sedes, todas en una misma organización con una clase por sede:

Sede	Organización	Clase
UNTREF	`untref-ayp2-estudiantes`	`ayp2-untref-2026`
CUDI	`untref-ayp2-estudiantes`	`ayp2-cudi-2026`

Usar la organización `untref-ayp2-estudiantes` y la clase que corresponda a tu cursada en todos los comandos.

7.3.8.1 Instalación de gh CLI

Windows

Descargar el instalador desde https://cli.github.com y ejecutarlo. Luego abrir Git Bash y continuar con los pasos siguientes.

Linux

console

```
sudo apt install gh
```

O seguir las instrucciones en https://cli.github.com.

macOS

console

```
brew install gh
```

7.3.8.2 Extensión de Classroom50

console

```
gh extension install foundation50/gh-student
```

7.3.8.3 Iniciar sesión

console

```
gh student login
```

Se abre el navegador para autorizar. Completar el proceso y volver a la terminal.

7.3.8.4 Aceptar una tarea

console

```
gh student accept <organizacion> <clase> <tarea>
```

Reemplazar `<organizacion>` por `untref-ayp2-estudiantes`, `<clase>` por tu clase (`ayp2-untref-2026` o `ayp2-cudi-2026`) según tu sede, y `<tarea>` por el slug de la tarea (ej. `taller-go`).

Esto crea un repositorio propio en GitHub con el código inicial.

7.3.8.5 Entregar

console

```
cd <repositorio>
gh student submit
```

Esto ejecuta los tests automáticos y publica el resultado. Se puede entregar varias veces; la última entrega es la que cuenta.

Classroom50 desde la web

Además de la línea de comandos, Classroom50 se puede usar desde el navegador.

1. Ir a https://classroom50.org.
2. Click en **"Sign in with GitHub"** y autorizar la aplicación.
3. Seleccionar la organización `untref-ayp2-estudiantes` y la clase correspondiente a tu sede.
4. Elegir la clase correspondiente.
5. Se ven las tareas asignadas, su estado (pendiente/entregada) y el puntaje obtenido.
6. Desde cada tarea se puede acceder al repositorio, al Feedback PR y al detalle de la última entrega.

7.3.9.1 Entregar desde la web (release)

También se puede entregar directamente desde github.com, creando un **release** con el tag `submit`:

1. Completar los ejercicios y hacer **commit y push** de tus cambios.
2. Ir al repositorio de la tarea en github.com.
3. En la barra lateral derecha, sección **Releases**, click en **Create a new release** (o **Draft a new release** si ya hay releases).
4. En **Choose a tag**, escribir `submit` y elegir la opción **"+ Create new tag: submit on publish"** que aparece. Opcionalmente podés agregar texto después de una barra, ej.: `submit/intento-2`.
5. Completar título y descripción (opcional).
6. Click en **Publish release**.

Eso dispara la entrega: se corren los tests automáticos y se publica el resultado, igual que con `gh student submit`.

Importante

- Se pueden hacer **todas las entregas que quieran** hasta que se cierre la tarea; la última entrega es la que cuenta.
- El puntaje se calcula **únicamente cuando se hace una entrega** (`gh student submit` o un release con tag `submit`). Hacer commit y push solo guarda tu trabajo en el repositorio: no genera puntaje ni feedback.

Feedback PR

El **Feedback PR** es un Pull Request que se crea automáticamente al entregar la primera vez. Es el espacio donde los docentes dejan comentarios sobre tu código.

Está en la pestaña **"Pull requests"** del repositorio, con el título **Feedback**.

7.3.10.1 Desde el navegador web (GitHub)

1. Ir al repositorio en github.com.
2. Click en la pestaña **Pull requests**.
3. Click en **"Feedback"**.
4. Para comentar una línea específica:
    - Click en el número de línea -> icono + -> escribir comentario -> **"Add single comment"**.
5. Para hacer una consulta general:
    - Ir al final del PR, escribir en la caja de texto y click en **"Comment"**.
6. Para `@mencionar` a un docente o ayudante:
    - Escribir `@usuario` (ej. `@martin-albarracin`).
    - Le llega una notificación por correo.
7. Para responder a un comentario:
    - Click en **"Reply"** debajo del comentario.
8. Las notificaciones se ven en la campana de GitHub (arriba a la derecha).

7.3.10.2 Desde VS Code

1. Abrir el repositorio en VS Code.
2. Click en el icono de PR en la barra lateral izquierda (extensión GitHub Pull Requests).
3. Seleccionar **"Feedback"** de la lista.
4. Comentar una línea: click en el número de línea -> icono de comentario -> escribir.
5. `@mencionar` y responder funciona igual que en la web.

Cómo ejecutar tests

Los tests verifican que tu código funciona correctamente. Se ejecutan desde la terminal (Git Bash, terminal de Linux o macOS).

7.3.11.1 Todos los tests

console

```
go test ./...
```

7.3.11.2 Tests de un tema específico

console

```
go test ./01-introduccion/...
```

7.3.11.3 Con detalle (muestra cada caso)

console

```
go test -v ./...
```

7.3.11.4 Usando Makefile (si el repo tiene uno)

console

```
make test
```

7.3.11.5 Cómo leer el output

- `ok` — todos los tests pasaron.
- `FAIL` — algún test falló.
- `=== RUN` — nombre del test que se está ejecutando.
- `--- PASS` — test pasado.
- `--- FAIL` — test fallado, con el mensaje de error.

Referencias

Adelson-Velsky, G., & Landis, E. (1962). An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, *146*, 263–266. https://zhjwpku.com/assets/pdf/AED2-10-avl-paper.pdf

Gamma, E. (2002). *Patrones de diseño: elementos de software orientado a objetos reutilizable*. Pearson Education. https://books.google.com.ar/books?id=gap_AAAACAAJ