

Struktur Data dengan Pemrograman Go-Lang



Pencipta:

Yohanes Putra P. Muwa Dae
Nabila Syahnas Sesarani
Nadiyah Hana Mufidah
Nurliasta Jayanti
Putri Zahra Panggih Setiawati
Safira Salsabila
Muhammad Jauharul Fuady, S.T., M.T.

KATA PENGANTAR

Puji syukur kami panjatkan ke hadirat Allah SWT karena atas rahmat dan karunia-Nya, buku berjudul “Pengimplementasian Struktur Data dalam Bahasa Pemrograman Go” ini dapat disusun dan diselesaikan dengan baik. Buku ini bertujuan untuk memberikan pemahaman mengenai konsep dan implementasi struktur data menggunakan bahasa pemrograman Go.

Penyusunan buku ini merupakan hasil kerja sama yang solid dan dedikasi dari seluruh anggota kelompok. Kami berusaha menyajikan materi secara sistematis dan aplikatif agar mudah dipahami oleh pembaca, khususnya mahasiswa yang sedang mempelajari struktur data.

Kami menyadari bahwa buku ini masih jauh dari kata sempurna. Oleh karena itu, kami sangat mengharapkan kritik dan saran yang membangun demi perbaikan di masa mendatang. Semoga buku ini dapat menjadi referensi yang bermanfaat bagi mahasiswa maupun pembaca umum yang ingin memahami dan mengimplementasikan struktur data dalam bahasa Go. Akhir kata, kami mengucapkan terima kasih kepada dosen pembimbing dan semua pihak yang telah memberikan dukungan serta motivasi dalam penyusunan buku ini.

DAFTAR ISI

| | |
|---|----|
| Kata Pengantar | i |
| Daftar Isi..... | ii |
| Bab 1. Pengantar Bahasa Go | 1 |
| Bahasa Pemrograman Go | 2 |
| Instalasi | 3 |
| Bab 2. Pemrograman Dasar Go..... | 6 |
| Sintaks Dasar..... | 7 |
| Operator | 15 |
| Struktur Kontrol..... | 21 |
| Pengantar Array, Slice Dan Map..... | 26 |
| Error Handling | 29 |
| Bab 3. Fungsi | 33 |
| Pengenalan Fungsi..... | 34 |
| Fungsi Return Single Dan Return Multiple Value..... | 35 |
| Fungsi Rekursif..... | 37 |
| Defer, Panic Dan Recover | 39 |
| Bab 4. Pointer | 44 |
| Operator Address | 45 |
| Pengenalan Pointer | 45 |
| Pointer Di Fungsi..... | 46 |
| Pointer Di Method | 49 |
| Bab 5. Module Dan Package | 51 |
| Pengenalan Module | 52 |
| Pengelolaan Module..... | 52 |
| Struktur Direktori Go | 53 |
| Menggunakan Custom Package..... | 53 |
| Mengimpor Package Standar | 54 |
| Bab 6. Struktur Data Bawaan..... | 55 |
| Array | 56 |
| Slice..... | 61 |
| Map..... | 64 |
| Struct | 67 |

| | |
|---|------------|
| Linked List | 71 |
| Bab 7. Struktur Data Lanjutan | 76 |
| Stack | 77 |
| Queue | 80 |
| Graph..... | 84 |
| Tree..... | 90 |
| Hash..... | 95 |
| Trie | 99 |
| Bab 8. Implementasi | 104 |
| Sistem Antrian Pada Rumah Sakit | 105 |
| Sistem Navigasi Kota Sederhana..... | 110 |
| Sistem Undo Redo Dalam Pengolahan Data Dinamis..... | 116 |
| Daftar Pustaka | 123 |

BAB 1

PENGANTAR BAHASA GO

A. BAHASA PEMROGRAMAN GO

Apa itu Bahasa Pemrograman

Program komputer atau seringkali disingkat sebagai program adalah serangkaian instruksi yang ditulis untuk melakukan sesuatu yang spesifik pada komputer. Komputer membutuhkan program agar bisa menjalankan tugas dan fungsinya. Hal ini dilakukan dengan cara mengeksekusi serangkaian instruksi program yang telah dibuat pada prosesor.

Bahasa pemrograman atau seringkali disebut dengan bahasa komputer ataupun bahasa pemrograman komputer merupakan bahasa formal yang digunakan untuk memberikan perintah kepada komputer agar menjalankan fungsi tertentu. Bahasa pemrograman ini seakan menjadi perantara antara manusia dengan mesin agar dapat berkomunikasi secara terstruktur. Dengan adanya bahasa pemrograman yang terdiri dari kumpulan dari aturan sintaks digunakan untuk mendefinisikan program komputer, manusia dapat menulis perintah yang mudah dipahami oleh komputer. Pada akhirnya, komputer dapat membaca dan menjalankan perintah dengan benar.

Perkembangan bahasa pemrograman dimulai sejak awal ditemukan pada komputer yakni sekitar tahun 1940-an. Bahasa pertama yang muncul sangat sederhana dengan fungsi yang terbatas. Adanya perkembangan zaman membuat bahasa pemrograman menjadi lebih canggih dan saat ini telah ada ratusan bahasa pemrograman yang memiliki fungsi dan kelebihan masing-masing.

Bahasa Go (Go-Lang)

Bahasa Go, atau yang biasa dikenal dengan bahasa Golang adalah bahasa pemrograman yang modern dan relatif baru dikembangkan oleh Google pada tahun 2007. Proyek Go dimulai dengan kolaborasi antara 3 tokoh yang sangat berpengaruh dalam dunia komputasi juga merupakan insinyur dari Google, yakni Robert Griesemer, Rob Pike, dan Ken Thompson. Bahasa Golang dirancang dengan tujuan mereka untuk mengatasi persoalan yang sering dihadapi Google, selain itu memudahkan pengguna dalam menuliskan kode, dan juga kemampuan kompilasi yang tinggi dalam pengembangan perangkat lunak di era yang terus berkembang.

Bahasa Go pertama kali diumumkan ke publik pada bulan November 2009. Namun masih perlu proses pengembangan dan penyempurnaan. Setelah itu versi stabil pertamanya dirilis pada bulan Maret 2012. Dengan ini, Google berharap Go dapat menjadi bahasa pemrograman yang efisien, mudah dipelajari dan sebagai penyedia layanan untuk mengembangkan aplikasi berskala besar. Sejak saat itu, Go telah mengalami pertumbuhan pesat dan menjadi salah satu bahasa pemrograman yang populer di kalangan pengusaha teknologi ternama dan developer profesional.

Selama beberapa tahun terakhir, Go telah menjadi salah satu bahasa pemrograman yang paling populer dan terkenal. Perusahaan besar seperti *Google*, *Uber*, *Dropbox*, dan *Twitch*

menggunakan Go dalam proyek-proyek industri mereka. Bahasa ini sangat efektif digunakan pada berbagai platform dan lingkungan, terutama untuk pengembangan layanan yang berbasis *cloud*, *mikroservis*, serta perangkat lunak yang membutuhkan performa tinggi dan juga stabilitas dalam jangka panjang. Alasan utama Go dikembangkan adalah untuk menyempurnakan keterbatasan bahasa pemrograman lainnya dalam hal pengelolaan dependensi, performa kompilasi, dan dukungan terhadap pemrosesan paralel (*concurrency*).

Kegunaan dan Lingkup Pemakaian Bahasa Go

Bahasa Go (Golang) adalah bahasa pemrograman yang dirancang untuk efisiensi, kecepatan kompilasi, dan kemudahan dalam penulisan kode, terutama dalam pengembangan sistem dan aplikasi berskala besar. Bahasa ini secara luas digunakan dalam berbagai bidang industri karena kemampuannya menangani beban kerja tinggi dengan performa yang stabil. Beberapa bidang utama penggunaan Go antara lain adalah pengembangan *backend* dan layanan web, sistem *cloud*, *microservices*, pemrosesan data, otomasi DevOps, dan pengembangan aplikasi jaringan.

Dalam pengembangan *backend*, Go sering digunakan untuk membuat layanan RESTful API menggunakan *framework* seperti Gin dan Echo. Di bidang DevOps, Go menjadi dasar bagi alat-alat penting seperti Docker dan Terraform, yang membantu dalam pengelolaan infrastruktur dan deployment. Untuk pemrosesan data, Go dipilih dalam pembuatan sistem streaming dan pengolahan log karena kecepatannya dalam menangani data dalam jumlah besar. Dalam pengembangan aplikasi web, *framework* seperti Fiber dan Beego memudahkan pembuatan antarmuka pengguna yang responsif. Sedangkan dalam pengembangan *microservices*, Go menjadi bahasa utama dalam platform seperti Istio dan Consul yang mengatur komunikasi antar layanan secara efisien.

B. INSTALASI

Apa saja yang harus di download?

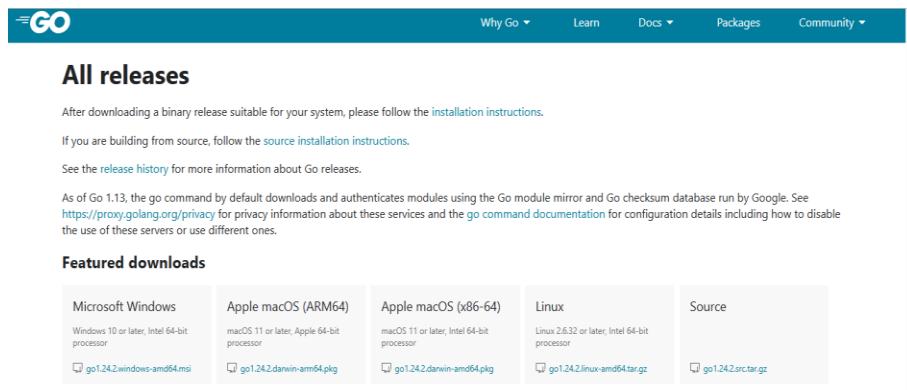
Berikut ini adalah file atau program yang perlu di-download sebelum memulai:

- a. Installer Bahasa Go. Download dari situs resmi: <https://go.dev/dl/>. Lalu pilih file installer sesuai sistem operasi (misal: .msi untuk Windows).
- b. Visual Studio Code (atau editor lain). Download dari : <https://code.visualstudio.com/>
- c. Ekstensi Go untuk VS Code. Bisa diinstal langsung dari VS Code (di menu Ektensi), cari “Go”.
- d. Git (Opsional). Jika ingin mengelola kode versi atau menggunakan GitHub.

Cara Instalasi Bahasa Go

Berikut ini adalah langkah-langkah instalasi bahasa pemrograman Go:

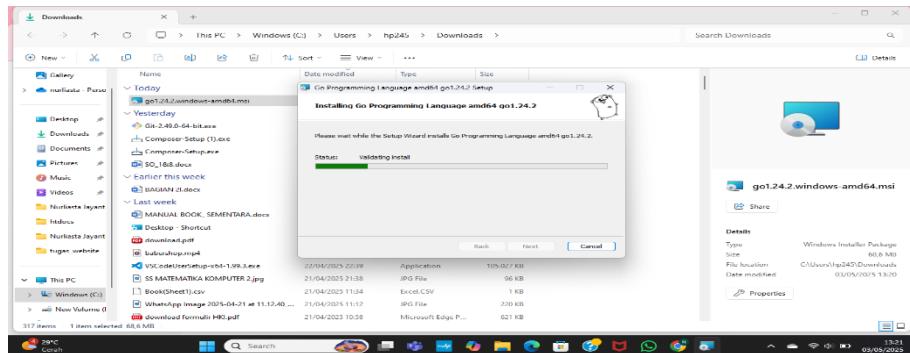
- Unduh Installer. Buka <https://go.dev/dl/>. Pilih versi Go yang sesuai dengan sistem operasi (misalnya go1.22.0.windows-amd.msi untuk Windows)



Gambar 1.1 Unduh Installer

- Jalankan Installer

- Pada Windows: Klik dua kali file .msi. Lalu ikuti panduan instalasi, tekan Next terus hingga selesai



Gambar 1.2 Menjalankan Installer di Windows

- Pada MacOS/ Linux gunakan file .pkg (macOS) atau .tar.gz (Linux), kemudian ekstrak dan pasang melalui terminal

- Verifikasi Instalasi

- Buka Command Prompt/ Terminal

- Ketik perintah berikut:

```
go version
```

- Jika berhasil, akan muncul versi Go yang terinstal, misalnya:

```
go version go1.22.0 windows/amd64
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go version  
go version go1.24.2 windows/amd64
```

Gambar 1.3 Mengetahui Versi Go

Biasanya installer otomatis menambahkan Go ke dalam PATH. Namun jika tidak, tambahkan folder c:\Go\bin ke dalam Environment Variables > PATH

e. Tes Program Pertama

- Buat file `hallo.go` dengan isi seperti contoh program berikut:

```
1. package main  
2.  
3. import "fmt"  
4.  
5. func main() {  
6.     fmt.Println("satu = ", 1)  
7. }
```

- Jalankan lewat terminal:

```
go run hallo.go
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run hello.go  
satu = 1
```

Gambar 1.4 Tes Program Pertama

BAB 2

PEMROGRAMAN DASAR GO

A. SINTAKS DASAR

Penamaan

Penamaan dalam pemrograman bukan sekedar estetika, tetapi juga konsistensi keterbacaan, dan pemeliharaan kode. Dalam Golang, konversi penamaan sangat penting karena Go mengandalkan pendekatan minimalis tanpa banyak konfigurasi. Oleh karena itu, setiap pengembang (terutama pemula) perlu memahami aturan-aturan penamaan dasar ini.

a. Penamaan File dalam Go.

Go memiliki aturan sederhana namun penting, terutama agar modul dan package dikenali serta diimpor dengan benar.

Aturan Umum:

- Gunakan huruf kecil semua (*lowercase*). Contoh: `user.go`, `helloworld.go`
- Bisa juga menggunakan `snake_case`, `camelCase` dan `kebab-case`
- Untuk file pengujian, gunakan akhiran `*_test.go`. Contoh: `user_test.go`

b. Penamaan Package

Package dalam Go adalah kumpulan file Go yang dikelompokkan berdasarkan fungsinya.

Penamaan package harus:

- Ditulis dengan huruf kecil semua. Contoh: `package user`, `package payment`
- Sesuai dengan nama folder tempat package berada.
- Nama package sebaiknya mencerminkan tujuan atau isi dari package tersebut.
- Tidak menggunakan *underscore*, kapitalisasi, atau tanda pemisah lainnya.

Contoh Program:

```
1. package UserPkg      // Salah: menggunakan kapitalisasi
2. package my_package   // Salah: menggunakan underscore
```

c. Penamaan *Identifiers*.

Identifiers mencakup variabel, struct, interface, dan konstanta. Dalam Go, ada dua gaya penulisan utama:

- *camelCase*: untuk variabel lokal dan parameter fungsi. Contoh Program:

```
1. var userName string      // Variabel lokal
2. func calculateTotal     // Fungsi local
```

- *PascalCase*: untuk item yang dieksport (*exported*), agar bisa diakses dari package lain. Contoh Program:

```
1. func GetUserData () { }    // Fungsi dieksport
```

Tipe Data

Dalam bahasa pemrograman Go, tipe data merupakan elemen dasar yang menentukan jenis nilai yang dapat disimpan dan diolah oleh program. Go mendukung berbagai tipe data, termasuk tipe data numerik (*int*, *float*), *boolean* (*bool*), *string* (*string*), serta tipe data kompleks seperti *array*, *struct*, dan *interface*.

a. *Integer*

Terdapat banyak jenis tipe data integer yang masing-masing memiliki rentang dan penggunaan memori yang berbeda. Semakin besar ukuran tipe data tersebut, semakin besar pula memori yang dibutuhkan. Tipe data *integer* di Go diklasifikasikan menjadi dua, yaitu *Signed Integer* dan *Unsigned Integer*.

• *Signed Integer*

Signed Integer adalah tipe data bilangan bulat yang mampu merepresentasikan nilai negatif maupun positif. Tipe data ini menggunakan satu bit paling signifikan untuk menyatakan tanda (positif atau negatif), sehingga rentang nilainya terbagi secara simetris antara bilangan negatif dan positif. Perhatikan tabel berikut:

| Tipe Data | Nilai Minimum | Nilai Maksimum |
|--------------|----------------------|---------------------|
| int8 | -128 | 127 |
| int16 | -32768 | 32767 |
| int32 | -2147483648 | 2147483647 |
| int64 | -9223372036854775808 | 9223372036854775807 |

Tabel 2.1 *Signed Integer*

• *Unsigned Integer*

Unsigned Integer adalah tipe data bilangan bulat yang hanya merepresentasikan nilai-nilai non-negatif (positif dan 0). Berbeda dengan *singed integer*, tipe ini tidak menggunakan bit untuk menyatakan tanda, sehingga seluruh bit yang tersedia digunakan untuk menyimpan nilai. Akibatnya, tipe data ini memiliki rentang positif yang lebih luas dibandingkan *Signed Integer* dengan ukuran bit yang sama.

| Tipe Data | Nilai Minimum | Nilai Maksimum |
|---------------|---------------|----------------------|
| uint8 | 0 | 255 |
| uint16 | 0 | 65535 |
| uint32 | 0 | 4294967295 |
| uint64 | 0 | 18446744073709551615 |

Tabel 2.2 *Unsigned Integer*

Pada Tipe data Integer, terdapat kata Ganti sebagai berikut ini:

| Tipe Data | Nilai Minimum |
|-----------|----------------|
| byte | uint8 |
| rune | int32 |
| int | minimal int32 |
| uint | minimal uint32 |

Tabel 2.3 Kata Ganti di Integer

b. Floating Point

Floating Point adalah tipe data yang digunakan untuk merepresentasikan bilangan riil, yaitu bilangan yang memiliki bagian desimal atau pecahan.

| Tipe Data | Nilai Minimum | Nilai Maksimum |
|-----------|-------------------------|----------------------|
| float32 | 1.18×10^{-38} | 3.4×10^{38} |
| float64 | 2.23×10^{-308} | 3.4×10^{38} |

Tabel 2.4 Floating Point

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main () {
6.     fmt.Println("Lima = ", 5)
7.     fmt.Println("Enam = ", 6)
8.     fmt.Println("Dua koma empat = ", 2.4)
9. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run numbers.go
Lima = 5
Enam = 6
Dua koma empat = 2.4
```

Gambar 2.1 Tipe Data Integer dan Float

c. Boolean

Boolean adalah tipe data yang memiliki dua nilai logika, yaitu benar (*true*) dan salah (*false*). Dalam Go, tipe data *boolean* direpresentasikan dengan kata kunci '*bool*' .

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     fmt.Println("Benar = ", true)
7.     fmt.Println("Salah = ", false)
8. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run bool.go
Benar = true
Salah = false
```

Gambar 2.2 Tipe Data Boolean

d. **String**

String adalah tipe data yang digunakan untuk menyimpan urutan karakter. Nilai *string* ditulis diantara tanda kutip ganda (""), dan direpresentasikan dengan kata kunci *string*.

Contoh program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     fmt.Println("Halo!")
7.     fmt.Println("Belajar Go itu seru!")
8. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run strg.go
Halo!
Belajar Go itu seru!
```

Gambar 2.3 Tipe Data String

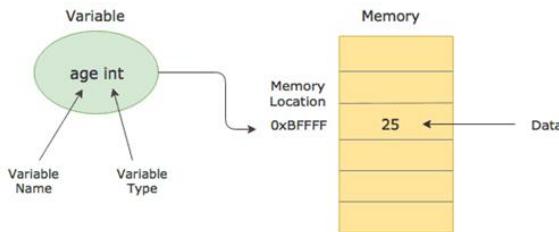
Variabel dan Konstanta

a. Variabel

Setiap program memerlukan ruang untuk menyimpan data. Data ini akan ditempatkan pada lokasi tertentu di dalam memori. RAM (*Random Access Memory*) pada komputer terdiri atas jutaan sel memori, di mana masing-masing sel memiliki ukuran sebesar 1 byte. Dalam program, data disimpan menggunakan variabel. Variabel adalah wadah untuk menyimpan data agar dapat digunakan dan diakses kembali di berbagai bagian kode. Dalam bahasa Go, satu variabel hanya dapat menyimpan satu jenis tipe data. Jika diperlukan penyimpanan untuk berbagai tipe data, maka harus dibuat beberapa variabel terpisah. Deklarasi variabel menggunakan kata kunci *var*, diikuti oleh nama variabel dan tipe datanya. Saat Anda menulis pernyataan kode seperti berikut:

```
1. var age int = 25;
```

Maka ketika program dijalankan, secara internal akan direpresentasikan oleh sistem operasi dengan cara berikut:



Gambar 2.4 Representasi Variabel

Sumber: CalliCoder

Namun, di bahasa pemrograman Go, kata kunci var saat membuka variabel tidak selalu wajib. Selama variabel langsung diinisialisasikan dengan sebuah nilai, kita dapat menggunakan sintaks pendek berupa operator := tanpa perlu tipe data secara eksplisit. Operator := memungkinkan deklarasi dan inisialisasi variabel dalam satu langkah, dan tipe datanya akan ditentukan secara otomatis berdasarkan nilai yang diberikan.

Contoh Program:

```

1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var nama string = "Budi"
7.     umur := 18
8.     fmt.Println("Nama saya:", nama)
9.     fmt.Println("Umur saya:", umur)
10. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run variabel1.go
Nama saya: Budi
Umur saya: 18
```

Gambar 2.5 Deklarasi Satu Variabel

Bahasa Go sendiri mendukung deklarasi lebih dari satu variabel secara bersamaan. Hal ini dilakukan dengan menuliskan beberapa nama variabel yang dipisahkan oleh tanda koma (,). Selain deklarasi, inisialisasi nilai untuk masing-masing variabel juga dapat dilakukan secara serentak dalam satu baris kode.

Contoh Program:

```

1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     kota, provinsi, kodePos := "Bandung", "Jawa Barat", 40123
7.
8.     fmt.Println("Kota:", kota)
9.     fmt.Println("Provinsi:", provinsi)
10.    fmt.Println("Kode Pos:", kodePos)
11. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run variabel2.go
Kota: Bandung
Provinsi: Jawa Barat
Kode Pos: 40123
```

Gambar 2.6 Deklarasi Banyak Variabel

b. Konstanta

Konstanta adalah variabel yang nilainya tetap dan tidak dapat diubah setelah pertama kali diberikan. Dalam bahas Go, deklarasi konstanta serupa dengan variabel, namun menggunakan kata kunci const sebagai pengganti var. Berbeda dengan variabel, konstanta harus langsung diinisialisasi saat dideklarasikan, dan tidak dapat diubah nilainya.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     const pi = 3.14
7.     const aplikasi = "GoApp"
8.
9.     fmt.Println("Nama Aplikasi:", aplikasi)
10.    fmt.Println("Nilai Pi:", pi)
11. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run konstanta.go
Nama Aplikasi: GoApp
Nilai Pi: 3.14
```

Gambar 2.7 Konstanta

Contoh berikut merupakan *error* yang akan terjadi jika kita mencoba mengubah nilai sebuah konstanta setelah dideklarasikan. Karena konstanta bersifat tetap, Go tidak mengizinkan perubahan terhadapnya:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     const pi = 3.14
7.     fmt.Println("Nilai awal pi:", pi)
8.
9.     pi = 3.14159
10.    fmt.Println("Nilai pi setelah diubah:", pi)
11. }
```

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run konstantaErr.go
# command-line-arguments
.\konstantaErr.go:9:5: cannot assign to pi (neither addressable nor a map index expression)
```

Gambar 2.8 Program Konstanta Error

Input dan Output

a. Mengambil Input

Untuk mengambil input pada Go, kita dapat menggunakan paket fmt dengan fungsi seperti `Scan`, `Scanln()` atau `Scanf()`. Perbedaan antara ketiganya yaitu:

- `Scan` digunakan untuk membaca *input* yang dipisahkan oleh spasi
- `Scanln()` sama seperti `Scan`, tetapi berhenti di akhir baris
- `Scanf()` untuk membaca *input* dengan format tertentu (seperti `Scanf` di C)

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var a, b int
7.     fmt.Print("Masukkan dua bilangan bulat (pisahkan dengan spasi): ")
8.     fmt.Scanln(&a, &b)
9.     fmt.Println("Jumlahnya adalah:", a+b)
10. }
```

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run ambilInput.go
Masukkan dua bilangan bulat (pisahkan dengan spasi): 5 9
Jumlahnya adalah: 14
```

Gambar 2.9 Mengambil Input

b. Menampilkan Output

Untuk menampilkan *output* ke layar pada Go, kita dapat menggunakan paket fmt, yaitu `Print()`, `PrintLn()`, dan `Printf()`. Fungsi `Print()` mencetak data ke layar tanpa menambahkan baris baru setelahnya. Sementara itu, `PrintLn()` akan mencetak data dan otomatis pindah ke baris berikutnya. Jika ingin menampilkan *output* dengan format tertentu, seperti menyisipkan nilai variabel ke dalam *string*, kita bisa menggunakan `Printf()`.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     nama := "Dina"
7.     umur := 19
8.
9.     fmt.Print("Halo ")
10.    fmt.Println(nama)
11.    fmt.Printf("Umur: %d tahun\n", umur)
12. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run tampilOutput.go
Halo Dina
Umur: 19 tahun
```

Gambar 2.10 Menampilkan Output

Komentar

Komentar digunakan untuk menambahkan catatan pada kode program, memberikan penjelasan mengenai bagian tertentu dari kode, atau menonaktifkan sementara baris kode yang tidak diperlukan (*remark*). Komentar tidak akan diproses saat program dikompilasi atau dijalankan. Go menyediakan dua jenis komentar yaitu *Inline* dan *Multi Inline*.

a. Komentar *Inline*

Komentar jenis ini diawali dengan tanda double slash (//), diikuti dengan isi komentar. Komentar semacam ini hanya berlaku untuk satu baris saja. Jika ingin menulis komentar lebih dari satu baris, maka setiap barisnya harus diawali kembali dengan //.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     // Inisialisasi variabel
7.     nama := "Dino"
8.
9.     // Menampilkan nama ke layar
10.    fmt.Println("Halo,", nama)
11.
12.    // fmt.Println("Baris ini dinonaktifkan dan tidak akan dieksekusi")
13. }
```

Mari coba jalankan kode di atas secara langsung. Buatlah file program baru di dalam folder proyek, bisa menggunakan proyek yang sudah ada atau membuat yang baru. Selanjutnya salin dan tempelkan kode tersebut ke dalam file, lalu jalankan programnya.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run singleKomen.go
Halo, Dino
```

Gambar 2.11 Komentar *Inline*

Hasil yang akan muncul di layar hanyalah teks `Halo, Dino` karena seluruh baris yang diawali tanda // akan diabaikan oleh *compiler* dan tidak dijalankan.

b. Komentar *Multi Inline*

Go juga mendukung komentar *Multi Inline* yang diawali dengan /* dan diakhiri dengan */. Komentar jenis ini digunakan ketika ingin menulis penjelasan yang lebih panjang atau mencakup beberapa baris sekaligus.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     /*
7.     fmt.Println("Baris ini tidak akan dijalankan")
8.     fmt.Println("Karena berada dalam komentar multi inline")
9.     */
10.    fmt.Println("Hanya baris ini yang tampil")
11. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run multiKomen.go
Hanya baris ini yang tampil
```

Gambar 2.12 Komentar Multi Inline

B. OPERATOR

Dalam bahasa Go, operator adalah simbol-simbol khusus yang digunakan untuk melakukan operasi terhadap satu atau lebih operand—operand ini bisa berupa variabel, nilai konstan, ekspresi, atau hasil evaluasi lainnya. Operator merupakan bagian penting dari bahasa pemrograman karena memungkinkan kita untuk mengolah data dan mengontrol alur logika program.

Operator di Go bekerja pada berbagai tipe data seperti *integer*, *float*, *boolean*, dan lainnya, tergantung pada jenis operator yang digunakan. Misalnya, operator aritmatika digunakan untuk menghitung nilai numerik, operator relasional digunakan untuk membandingkan dua nilai, dan operator logika digunakan untuk menyusun ekspresi *boolean* yang kompleks.

Operator Assignment

Operator *assignment* (penugasan) adalah operator yang digunakan untuk memberikan atau memperbarui nilai suatu variabel. Dalam bahasa Go, operator *assignment* bukan hanya terbatas pada tanda sama dengan (=), tetapi juga mencakup bentuk kombinasi yang menggabungkan operasi matematika dengan penugasan, seperti +=, -=, *=, dan sebagainya. Secara umum, *assignment* operator digunakan dalam berbagai konteks: untuk menginisialisasi nilai, memperbarui *state* suatu variabel di dalam perulangan, atau sebagai bagian dari proses perhitungan dalam fungsi dan logika program lainnya.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
```

```

6.     var x int = 10
7.     fmt.Println("Nilai awal x:", x)
8.
9.     x += 5 // sama dengan x = x + 5
10.    fmt.Println("Setelah x += 5:", x)
11.
12.    x -= 2 // sama dengan x = x - 2
13.    fmt.Println("Setelah x -= 2:", x)
14.
15.    x *= 3 // sama dengan x = x * 3
16.    fmt.Println("Setelah x *= 3:", x)
17.
18.    x /= 4 // sama dengan x = x / 4
19.    fmt.Println("Setelah x /= 4:", x)
20.
21.    x %= 3 // sama dengan x = x % 3
22.    fmt.Println("Setelah x %= 3:", x)
23. }

```

Pertama, `x += 5` berarti menambahkan 5 ke nilai x saat ini (yang semula 10), sehingga hasilnya menjadi 15. Hasil ini dicetak ke layar. Lalu, `x -= 2` akan mengurangi nilai x (yang sekarang 15) sebesar 2, menghasilkan 13. Setelah itu, `x *= 3` akan mengalikan x (13) dengan 3, menghasilkan 39. Selanjutnya, `x /= 4` akan membagi nilai x (39) dengan 4. Karena tipe data x adalah *integer*, maka hasil pembagian ini akan dibulatkan ke bawah secara otomatis (*integer division*), sehingga hasilnya menjadi 9. Kemudian, `x %= 3` akan menghitung sisa pembagian dari `x` (9) dengan 3, dan hasilnya adalah 0 karena 9 habis dibagi

Output :

```

PS C:\A\UM\VS CODE\6_Go\src> go run oprAssignment.go
Nilai awal x: 10
Setelah x += 5: 15
Setelah x -= 2: 13
Setelah x *= 3: 39
Setelah x /= 4: 9
Setelah x %= 3: 0

```

Gambar 2.13 Operator Assignment

Operator Aritmatika

Operator aritmatika dalam bahasa Go adalah simbol-simbol yang digunakan untuk melakukan operasi matematika dasar antara dua operand. Operator ini umum digunakan dalam hampir semua program, baik itu untuk melakukan perhitungan sederhana, membuat logika dalam algoritma, atau menghitung nilai dalam struktur data. Go menyediakan operator aritmatika yang mencakup operasi penjumlahan, pengurangan, perkalian, pembagian, dan modulus.

Operator aritmatika bekerja pada tipe data numerik, seperti `int`, `float32`, dan `float64`. Saat dua nilai beroperasi melalui operator aritmatika, hasilnya juga akan bertipe numerik. Go secara eksplisit mewajibkan bahwa operand harus bertipe sama; tidak seperti beberapa bahasa lain yang mengizinkan konversi otomatis (*implicit type conversion*), Go akan menghasilkan error jika tipe tidak konsisten, sehingga konversi harus dilakukan secara eksplisit.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     a := 15
7.     b := 4
8.
9.     fmt.Println("a =", a)
10.    fmt.Println("b =", b)
11.
12.    fmt.Println("Penjumlahan: a + b =", a + b)
13.    fmt.Println("Pengurangan: a - b =", a - b)
14.    fmt.Println("Perkalian : a * b =", a * b)
15.    fmt.Println("Pembagian : a / b =", a / b)
16.    fmt.Println("Modulus     : a %% b =", a % b)
17. }
```

Baris pertama dari *output* hanya mencetak nilai awal dari *a* dan *b*, yaitu 15 dan 4, sebagai konteks sebelum dilakukan operasi. Kemudian, program mencetak hasil dari penjumlahan *a* + *b*, yang menghasilkan 19. Setelah itu, hasil dari pengurangan *a* - *b* adalah 11, karena 15 dikurangi 4 sama dengan 11. Pada baris berikutnya, dilakukan perkalian *a* * *b*, yang memberi hasil 60.

Operasi pembagian menggunakan operator / pada dua bilangan bulat (int). Dalam bahasa Go, jika kedua operand adalah *integer*, maka hasilnya juga akan berupa *integer*. Jadi, 15 / 4 bukan 3.75, melainkan 3, karena angka di belakang koma akan dibuang (dibulatkan ke bawah). Untuk mendapatkan hasil pecahan, variabel harus dikonversi menjadi tipe *float64*.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run oprAritmatika.go
a = 15
b = 4
Penjumlahan: a + b = 19
Pengurangan: a - b = 11
Perkalian : a * b = 60
Pembagian : a / b = 3
Modulus     : a %% b = 3
```

Gambar 2.14 Operator Aritmatika

Operator Relasional

Operator relasional adalah operator yang digunakan untuk membandingkan dua nilai. Hasil dari operasi ini bukan berupa angka, melainkan sebuah nilai *boolean*, yaitu *true* (benar) atau *false* (salah). Dalam bahasa Go, operator relasional memainkan peran penting dalam pengambilan keputusan, seperti dalam kondisi *if*, *for*, atau ekspresi logika lainnya.

Go mendukung enam jenis operator relasional utama: sama dengan (==), tidak sama dengan (!=), lebih besar dari (>), lebih kecil dari (<), lebih besar atau sama dengan (>=), dan lebih kecil atau sama dengan (<=). Semua operator ini dapat digunakan pada tipe data numerik seperti int, float64, dan juga pada tipe string untuk perbandingan alfabetikal.

Operator relasional digunakan untuk mengevaluasi kondisi yang menentukan alur eksekusi program. Misalnya, saat menjalankan kode hanya jika suatu nilai lebih besar dari nilai lain, atau saat memeriksa apakah dua nilai tidak sama, maka operator relasional yang digunakan.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     a := 10
7.     b := 7
8.
9.     fmt.Println("a =", a)
10.    fmt.Println("b =", b)
11.
12.    fmt.Println("Apakah a == b ?", a == b)
13.    fmt.Println("Apakah a != b ?", a != b)
14.    fmt.Println("Apakah a > b ?", a > b)
15.    fmt.Println("Apakah a < b ?", a < b)
16.    fmt.Println("Apakah a >= b ?", a >= b)
17.    fmt.Println("Apakah a <= b ?", a <= b)
18. }
```

Membandingkan dua buah nilai dan menghasilkan nilai *boolean* (*true* atau *false*) sebagai hasilnya. Dalam konteks ini, dua buah variabel *a* dan *b* masing-masing diisi dengan nilai 10 dan 7, lalu keduanya dibandingkan menggunakan beberapa jenis operator relasional.

Operator `==` digunakan untuk mengecek apakah nilai *a* sama dengan nilai *b*. Karena 10 tidak sama dengan 7, maka hasilnya adalah *false*. Sebaliknya, operator `!=` memeriksa apakah *a* tidak sama dengan *b*, dan karena memang tidak sama, hasilnya adalah *true*. Selanjutnya, `a > b` digunakan untuk melihat apakah nilai *a* lebih besar dari *b*, dan dalam hal ini bernilai *true* karena 10 memang lebih besar dari 7.

Operator `<` akan bernilai *false* karena 10 tidak lebih kecil dari 7. Lalu, `a >= b` akan bernilai *true* karena 10 memang lebih besar atau sama dengan 7. Sebaliknya, `a <= b` bernilai *false* karena 10 tidak lebih kecil atau sama dengan 7.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run oprRelasional.go
a = 10
b = 7
Apakah a == b ? false
Apakah a != b ? true
Apakah a > b ? true
Apakah a < b ? false
Apakah a >= b ? true
Apakah a <= b ? false
```

Gambar 2.15 Operator Relasional

Operator Logika

Operator logika dalam bahasa Go digunakan untuk menggabungkan dua atau lebih ekspresi logika (kondisi) dan mengevaluasi hasilnya sebagai true atau false. Operator ini sangat penting dalam pengambilan keputusan, terutama dalam struktur kontrol seperti if, for, dan switch, di mana kita sering perlu mengevaluasi lebih dari satu syarat sekaligus dalam sekali proses dengan efisiensi dan optimalisasi logika. Tiga operator logika utama yang digunakan dalam bahasa Go adalah:

- **AND (&&)**: Menghasilkan *true* hanya jika kedua operand bernilai *true*.
- **OR (||)**: Menghasilkan *true* jika setidaknya salah satu operand bernilai *true*.
- **NOT (!)**: Digunakan untuk membalik nilai *boolean*, *true* menjadi *false*, dan sebaliknya.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     a := 8
7.     b := 5
8.     // AND: Kedua kondisi harus benar
9.     if a > 5 && b < 10 {
10.         fmt.Println("a lebih besar dari 5 DAN b lebih kecil dari 10")
11.     }
12.
13.     // OR: Salah satu kondisi cukup benar
14.     if a > 5 || b < 3 {
15.         fmt.Println("a lebih besar dari 5 ATAU b lebih kecil dari 3")
16.     }
17.
18.     // NOT: Membalikkan logika dari kondisi
19.     if !(a < 5) {
20.         fmt.Println("a tidak lebih kecil dari 5")
21.     }
22. }
```

Pada blok pertama, digunakan operator `&&` untuk menguji apakah `a` lebih besar dari 5 dan `b` lebih kecil dari 10. Karena kedua pernyataan ini benar (`8 > 5` dan `5 < 10`), maka ekspresi `if` dievaluasi *true*, dan program mencetak "a lebih besar dari 5 DAN b lebih kecil dari 10".

Blok kedua menggunakan operator `||` untuk menguji apakah `a` lebih besar dari 5 atau `b` lebih kecil dari 3. Meskipun `b` tidak lebih kecil dari 3, nilai `a > 5` sudah benar, sehingga cukup untuk membuat ekspresi keseluruhan bernilai *true*. Maka, kalimat "a lebih besar dari 5 ATAU b lebih kecil dari 3" juga akan dicetak.

Terakhir, program menggunakan operator `!` untuk membalikkan hasil dari ekspresi `a < 5`. Karena `a` bernilai 8, maka ekspresi `a < 5` adalah *false*. Namun, karena operator `!` digunakan, nilai tersebut dibalik menjadi *true*. Hasilnya, pernyataan "a tidak lebih kecil dari 5" juga dieksekusi dan ditampilkan.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run oprLogika.go
a lebih besar dari 5 DAN b lebih kecil dari 10
a lebih besar dari 5 ATAU b lebih kecil dari 3
a tidak lebih kecil dari 5
```

Gambar 2.16 Operator Logika

Operator Bitwise

Operator *bitwise* adalah operator yang digunakan untuk memanipulasi data dalam bentuk bit secara langsung. Dalam sistem komputasi, semua data disimpan dalam bentuk biner (0 dan 1). Oleh karena itu, operator *bitwise* memungkinkan kita untuk melakukan operasi logika pada tingkat bit yang sangat efisien dan cepat.

Bahasa Go menyediakan beberapa operator *bitwise* utama, yang digunakan terutama pada tipe data bilangan bulat (int, int32, uint8, dan sebagainya). Operator ini sangat penting dalam pemrograman sistem, jaringan, keamanan, dan berbagai algoritma optimasi.

Operator *bitwise* dalam Go meliputi:

- **& (AND)**: Menghasilkan 1 hanya jika kedua bit bernilai 1.
- **| (OR)**: Menghasilkan 1 jika salah satu atau kedua bit bernilai 1.
- **^ (XOR)**: Menghasilkan 1 jika hanya satu dari dua bit yang bernilai 1.
- **&^ (AND NOT)**: Operator unik di Go. Menghapus bit yang aktif di operand kanan dari operand kiri.
- **<< (left shift)**: Menggeser bit ke kiri sebanyak n posisi (setara dengan perkalian 2^n).
- **>> (right shift)**: Menggeser bit ke kanan sebanyak n posisi (setara dengan pembagian 2^n)

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var a uint = 5 // 0101 dalam biner
7.     var b uint = 3 // 0011 dalam biner
8.
9.     fmt.Println("a & b =", a & b) //0101 & 0011 = 0001 → 1
10.    fmt.Println("a | b =", a | b) //0101 | 0011 = 0111 → 7
11.    fmt.Println("a ^ b =", a ^ b) //0101 ^ 0011 = 0110 → 6
12.    fmt.Println("a &^ b =", a &^ b)//0101 &^ 0011 = 0100 → 4
13.    fmt.Println("a << 1 =", a << 1) // 0101 << 1 = 1010 → 10
14.    fmt.Println("a >> 1 =", a >> 1) // 0101 >> 1 = 0010 → 2
15. }
```

Dalam contoh ini, dua variabel a dan b masing-masing disimpan sebagai nilai 5 dan 3. Representasi biner dari 5 adalah 0101, dan 3 adalah 0011. Operasi seperti a & b akan

menghasilkan nilai 1 karena hanya bit terakhir yang sama-sama bernilai 1. $a | b$ akan menghasilkan 7 karena seluruh bit yang aktif di salah satu operand tetap aktif di hasil. Demikian pula, $a \wedge b$ hanya menghasilkan 1 untuk bit-bit yang berbeda. Operasi $a \& \wedge b$ akan menghapus bit-bit dari a yang juga aktif di b . Untuk pergeseran, $a >> 1$ menggandakan nilai a , dan $a >> 1$ membaginya dua.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run oprBitwise.go
a & b = 1
a | b = 7
a ^ b = 6
a &^ b = 4
a << 1 = 10
a >> 1 = 2
```

Gambar 2.17 Operator Bitwise

C. STRUKTUR KONTROL

Percabangan

Percabangan merupakan salah satu struktur kontrol pada alur program yang digunakan dalam pembuatan keputusan berdasarkan suatu kondisi. Metode ini memungkinkan program hanya bisa dieksekusi hanya jika kondisi tertentu terpenuhi, atau memilih antara beberapa jalur eksekusi berdasarkan nilai atau hasil evaluasi logika. Nilai yang bertipe *bool* merupakan acuan blok kode mana yang akan dieksekusi, bisa berasal dari variabel, ataupun operasi perbandingan. Dalam bahasa pemrograman GO, terdapat dua expression utama yakni *if expression* dan *switch expression*.

a. *If expression*

If expression merupakan bentuk logika percabanga yang digunakan untuk menentukan alur program berdasarkan hasil evaluasi suatu kondisi. Dalam *if expression* nilai yang dihasilkan berupa *boolean* (*true* atau *false*). Jika suatu kondisi bernilai *true* maka blok kode yang ada didalam *if* akan dijalankan. Namun jika kondisi tersebut bernilai *false* maka blok kode akan dilewati dan program akan melanjutkan untuk mengeksekusi blok kode pada *else if* atau *else*.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     hari := "senin"
7.
8.     if hari == "rabu" {
9.         fmt.Println("Menggunakan seragam batik.")
10.    } else if hari == "senin" {
11.        fmt.Println("Menggunakan seragam putih.")
12.    } else if hari == "selasa" {
```

```

13.     fmt.Println("Menggunakan seragam biru.")
14. } else {
15.     fmt.Println("Menggunakan seragam bebas.")
16. }
17. }
```

Pada contoh di atas menggunakan *expression if, else if* dan *else* untuk menentukan jenis seragam berdasarkan nilai variable hari yaitu senin. Program akan memeriksa kondisi *if* satu persatu, karena nilai hari pada kondisi pertama bukan merupakan *value* dari variable yang telah dideklarasikan sebelumnya maka program akan mengecek ke kondisi selanjutnya. Setelah kondisi terpenuhi pada pernyataan ke dua maka blok kode tersebut akan dijalankan dan program akan berhenti dan mencetak “Menggunakan seragam putih.”

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run ifExpression.go
Menggunakan seragam putih.
```

Gambar 2.18 If Expression

b. Switch expression

Switch merupakan percabangan dengan banyak kasus, pernyataan ini digunakan ketika terdapat banyak kasus yang ingin diperiksa. Setiap kasus akan memeriksa nilai suatu ekspresi dan akan dieksekusi jika ekspresi cocok dengan kasus tersebut. Ketika sebuah kasus terpenuhi, maka pengecekan tidak akan dilanjutkan ke kasus selanjutnya, meskipun terdapat *keyword break*. Konsep ini berkebalikan dengan *switch* pada pemrograman lain (yang ketika sebuah kasus terpenuhi, maka akan tetap dilanjut untuk mengecek kasus selanjutnya kecuali ada *keywoad break*).

Contoh Program:

```

1. package main
2.
3. import "fmt"
4.
5. func main() {
6. matkul := "Matek"
7.
8. switch matkul {
9.     case "Pemweb":
10.    fmt.Println("Memakai ruangan digedung B11")
11.    case "Matkom":
12.    fmt.Println("Memakai ruangan digedung A20")
13.    case "Strukdat":
14.    fmt.Println("Memakai ruangan digedung B12")
15.    case "Basdat", "SO":
16.    fmt.Println("Menggunakan ruangan digedung A19")
17.    default:
18.    fmt.Println("Matakuliah tidak terdaftar")
19. }
20. }
```

Pada contoh di atas *switch* digunakan untuk menentukan ruangan kuliah berdasarkan matakuliah yang tersimpan dalam variabel matkul. Karena nilai dari variabel adalah “Matek”,

maka program akan mencocokkan dengan salah satu *case*. Jika tidak ada *case* yang cocok, maka program akan menjalankan bagian *default*. Karena tidak terdapat *case* yang cocok maka program akan mencetak “Matakuliah tidak terdaftar”.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run switchExpression.go
Matakuliah tidak terdaftar
```

Gambar 2.19 Switch Expression

Perulangan

Perulangan (*looping*) adalah struktur kontrol yang digunakan untuk mengeksekusi kode program secara berulang selama suatu kondisi masih terpenuhi atau selama masih ada data yang bisa diakses. Perulangan digunakan untuk menghindari penulisan kode yang berulang-ulang, sehingga menjadikan kode lebih efisien, rapi dan mudah untuk dikelola. *looping* bekerja dengan cara mengevaluasi kondisi, jika kondisi tersebut bernilai *true*, maka blok perulangan akan dijalankan. Lalu setelah blok selesai, kembali ke langkah pertama. Namun, jika kondisi bernilai *false*, maka perulangan berhenti dan program akan melanjutkan ke bagian setelah perulangan. Pada bahasa GO hanya terdapat satu jenis perulangan utama, yaitu *for loop*, namun meski demikian fleksibilitas yang tinggi sehingga bisa digunakan seperti *while* atau *do-while* dalam bahasa lain.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     counter := 1
7.
8.     for counter <= 5 {
9.         fmt.Println("Perulangan ke", counter)
10.        counter++
11.    }
12. }
```

Pada contoh terdapat variabel *counter* yang diinisialisasikan dengan nilai 1, *for* bekerja selama nilainya kurang dari atau sama dengan 5 maka program akan mencetak “Perulangan ke” diikuti dengan nilai *counter*. Kemudian nilai *counter* akan ditambah satu setiap iterasi hingga iterasi ke 6 program akan berhenti mencetak karena sudah lebih dari 5, sehingga *output* yang dihasilkan hanya sampai iterasi ke 5.

Selain itu, dalam *for loop* terdapat *statement* yang mencakup dua *statement* lainnya yakni *init statement* dan *post statement*. *Init statement* merupakan *statement* yang ditempatkan sebelum kondisi *for* di eksekusi yang cocok digunakan untuk inisialisasi variabel. Pertama-tama *init statement* akan dieksekusi hanya sekali kemudian kondisinya akan dicek lalu blok kode dijalankan. Selanjutnya *post statement* yang akan selalu dieksekusi disetiap akhir perulangan. Berbeda dengan *init statement*, *post* akan dieksekusi setiap perulangan selesai dijalankan.

Contoh Program:

```
1. package main
2. import "fmt"
3. func main() {
4.     for counter := 1; counter <= 5; counter++ {
5.         fmt.Println("Perulangan ke", counter)
6.     }
7. }
```

Pada contoh diatas `counter := 1` merupakan *init statemen* dan `counter++` merupakan *post statement*. Cara kerjanya tidak jauh berbeda dengan contoh sebelumnya hanya saja penulisannya dibuat lebih singkat. Pada contoh ini *for* ditulis langsung mencakup inisialisasi, kondisi dan penambahan nilai dalam satu baris.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run perulangan.go
Perulangan ke 1
Perulangan ke 2
Perulangan ke 3
Perulangan ke 4
Perulangan ke 5
```

Gambar 2.20 Perulangan dengan Satu dan Tiga Statement

Break* dan *Continue

Break dan *Continue* merupakan *control statements* yang digunakan dalam perulangan untuk mengatur bagaimana suatu *loop* berjalan. *Break* digunakan untuk menghentikan secara paksa sebuah perulangan bahkan jika kondisi perulangan masih bernilai *true*. Ketika *break* dieksekusi di dalam *for*, maka seluruh perulangan akan langsung berhenti dan program akan melanjutkan ke baris kode setelah perulangan.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     for a := 0; a < 10; a++ {
7.         if a == 6 {
8.             break
9.         }
10.        fmt.Println("a =", a)
11.    }
12. }
```

Pada contoh diatas deklarasi variabel `a` diinisialisasikan dengan nilai 0, kemudian dilakukan perulangan sebanyak `a < 10`. Di dalam blok perulangan terdapat kondisi `if a == 6`, yang jika terpenuhi maka akan mengeksekusi *break*. Perintah `fmt.Println("a =", a)` terletak di

luar *if* sehingga setiap nilai a dari 0 hingga 5 akan dicetak sebelum perulangan dihentikan. Saat a mencapai nilai 6, kondisi *if* jadi benar dan perulangan selesai tanpa mencetak a = 6.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run break.go
a = 0
a = 1
a = 2
a = 3
a = 4
a = 5
```

Gambar 2.21 Break

Sedangkan *Continue* digunakan untuk melewati sisa kode iterasi saat ini dan langsung lanjut ke iterasi berikutnya. Jika kondisi *continue* terpenuhi, maka baris setelahnya yang berada didalam *loop* tidak akan di eksekusi, tapi *loop* tetap berjalan untuk nilai berikutnya.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     for a := 0; a < 10; a++ {
7.         if a%2 == 0 {
8.             continue
9.         }
10.
11.         fmt.Println("a =", a)
12.     }
13. }
```

Kondisi `a%2 == 0` di dalam *for* berfungsi untuk memeriksa apakah nilai a merupakan bilangan genap menggunakan operasi modulus (%). Jika kondisi tersebut bernilai *true*, maka pernyataan *continue* akan dijalankan. Saat *continue* dijalankan program akan langsung melompat ke iterasi berikutnya, maka dari itu angka yang dicetak adalah bilangan ganjil dari 0 hingga kurang dari 10.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run breakContinue.go
a = 1
a = 3
a = 5
a = 7
a = 9
```

Gambar 2.22 Break dan Continue

D. PENGANTAR ARRAY, SLICE DAN MAP

Array

Array adalah kumpulan data bertipe sama, yang disimpan dalam sebuah variabel. Array memiliki kapasitas yang nilainya ditentukan pada saat pembuatan, menjadikan data yang disimpan di array tersebut jumlahnya tidak boleh melebihi yang sudah dialokasikan. *Default* nilai tiap elemen array pada awalnya tergantung dari tipe datanya. Jika int maka tiap element *zero value*-nya adalah 0 , jika bool maka false , jika string maka "" (string kosong) dan seterusnya. Setiap elemen array memiliki indeks berupa angka yang merepresentasikan posisi urutan elemen tersebut. Berikut adalah beberapa cara menginisialisasi sebuah Array.

- Inisialisasi Nilai Array Masing-masing

```
1. var names [3]string
2. names[0] = "Budi"
3. names[1] = "Doni"
4. names[2] = "Eko"
5. // names[2] = "Salah"
6.
7. fmt.Println(names[0])
8. fmt.Println(names[1])
9. fmt.Println(names[2])
```

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run array1.go
Budi
Doni
Eko
```

Gambar 2.23 Inisialisasi Nilai Array Masing-Masing

Variabel names kita buat sebagai `array string` dengan jumlah nilai yang dapat disimpan adalah 3 slot. Salah satu cara mengisi slot di array dapat dilihat pada kode diatas, dengan mengakses *index* array ke berapa lalu mengisinya dengan sebuah *value*. Mengingat *index* array dimulai dari 0, maka *index* akhir sebuah array adalah n-1. Bisa dilihat, kita membuat kita membuat 3 slot array lalu *index* akhir yang dapat kita akses adalah 3-1 = 2. Sehingga saat kita mencoba mengakses `names[3]` akan terjadi error.

- Inisialisasi Nilai Array saat Deklarasi

Pengisian elemen array bisa dilakukan pada saat deklarasi variabel. Caranya dengan menuliskan data elemen dalam kurung kurawal setelah tipe data, dengan pembatas antar elemen adalah tanda koma (,).

Contoh Program:

```
1. // Gaya Horizontal
2. var fruits1 = [4]string{"Apel", "Jeruk", "Melon", "Strawberry"}
3.
4. // Gaya Vertikal
5. var fruits2 = [4]string{
```

```

6.     "Anggur",
7.     "Lemon",
8.     "Mangga",
9.     "Semangka",
10.    }
11.    fmt.Println(fruits1)
12.    fmt.Println(fruits2)

```

Output:

```

PS C:\A\UM\VS CODE\6_Go\src> go run array2.go
[Apel Jeruk Melon Strawberry]
[Anggur Lemon Mangga Semangka]

```

Gambar 2.24 Inisialisasi Nilai Array saat Deklarasi

Pada kode di atas, kita membuat 2 `array string` dan langsung mengisi masing-masing 4 nilai awal dari array tersebut. Keempat nilai tersebut dapat dipisahkan oleh tanda koma. Dengan 2 gaya saat kita mengisi nilai array yaitu array gaya horizontal dan vertikal.

- Inisialisasi Nilai Awal Array Tanpa Jumlah Elemen

Deklarasi array yang nilainya diset di awal, boleh tidak dituliskan jumlah lebar array -nya, cukup ganti dengan tanda 3 titik (...). Metode penulisan ini membuat kapasitas array otomatis dihitung dari jumlah elemen array yang ditulis.

```

1. var numbers = [...]int{2, 4, 6, 8}

```

Variabel `numbers` secara otomatis kapasitas elemennya adalah 4.

Slice

Slice adalah *reference* elemen array. Slice bisa dibuat, atau bisa juga dihasilkan dari manipulasi sebuah array ataupun slice lainnya. Karena slice merupakan data *reference*, menjadikan perubahan data di tiap elemen slice akan berdampak pada slice lain yang memiliki alamat memori yang sama.

- Inisialisasi Slice

Cara pembuatan slice mirip seperti pembuatan array, bedanya tidak perlu mendefinisikan jumlah elemen ketika awal deklarasi. Pengaksesan nilai elemennya juga sama.

Contoh Program:

```

1. var names = []string{"Nisa", "Mulan", "Sofia"}
2. fmt.Println(names[0]) // "Nisa"

```

Perbedaan antara slice dan array dapat terlihat saat mendeklarasikan variabel. Jika jumlah elemen tidak disebutkan dalam deklarasi, maka variabel tersebut dianggap sebagai slice.

```

1. var fruits1 = []string{"Apel", "Anggur", "Alpukat"} // ini Slice
2. var fruits2 = [3]string{"Blueberry", "Belimbing", "Bengkuang"} // ini Array
3. var fruits3 = [...]string{"Ceri", "Cokelat", "Cranberry"} // ini Array

```

- Hubungan Array dan Slice

Sebenarnya slice dan array tidak bisa dibedakan karena merupakan sebuah kesatuan. Array adalah kumpulan nilai atau elemen, sedang slice adalah referensi tiap elemen tersebut. Slice bisa dibentuk dari array yang sudah didefinisikan, caranya dengan memanfaatkan teknik 2 *index* untuk mengambil elemen-nya.

Teknik 2 *index* adalah mengambil bagian dari array dengan 2 *index* yang terdiri dari *index* atas dan *index* bawah dari sebuah array. Kedua *index* ini yang akan menjadikan bagian dari slice.

Berikut contoh program pembuatan Slice dari Array:

```
1. var fruits = [8]string{
2.     "Apel",
3.     "Jeruk",
4.     "Melon",
5.     "Strawberry",
6.     "Bengkuang",
7.     "Mangga",
8.     "Anggur",
9.     "Semangka",
10.    }
11. var sliceFruits = fruits[2:6]
12.
13. fmt.Println(fruits)           // Array
14. fmt.Println(sliceFruits)      // Slice
```

Bisa dilihat dari kode di atas, kita memiliki array dengan nama `fruits` berjumlah 8 nilai. Lalu slice bernama `sliceFruits` yang mengambil bagian array `fruits` dari *index* 2 sampai 6 (dari `"Melon"`sampai `"Anggur"`). Sehingga slice ini menyimpan kumpulan data dari bagian sebuah array.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run arrayNSlice.go
[Apel Jeruk Melon Strawberry Bengkuang Mangga Anggur Semangka]
[Melon Strawberry Bengkuang Mangga]
```

Gambar 2.25 Penggunaan Array dan Slice

Map

Dalam bahasa pemrograman Go, Map adalah struktur data yang digunakan untuk menyimpan kumpulan pasangan *key-value*, di mana setiap kunci bersifat unik dan digunakan untuk mengakses nilai yang sesuai. Map memungkinkan pencarian data secara efisien dan mendukung operasi seperti penambahan, penghapusan, serta pemeriksaan keberadaan kunci. Dengan menggunakan Map, *programmer* dapat mengelola data secara terorganisir tanpa harus memikirkan urutan penyimpanannya, menjadikannya ideal untuk berbagai aplikasi seperti *caching*, konfigurasi, dan penyimpanan *lookup* tabel.

Berikut adalah beberapa cara membuat Map:

- Mendeklarasi Map

```
1. var myMap1 map[string]int
```

Deklarasi Map tanpa inisialisasi berarti bahwa variabel `myMap1` dibuat tetapi belum memiliki alokasi memori untuk menyimpan data. Map yang dideklarasikan seperti ini akan bernilai `nil` sampai diinisialisasi menggunakan `make()`.

- Membuat Map dengan *Keyword Make*

```
1. myMap2 := make(map[string]int)
```

Untuk menghindari Map bernilai `nil`, Go menyediakan fungsi bawaan `make()`, yang secara otomatis mengalokasikan memori sehingga Map siap digunakan. Dalam contoh di atas, sebuah Map dibuat dengan kunci bertipe `string` dan nilai bertipe `int`. Dengan cara ini, Map bisa langsung diisi tanpa risiko *error* terkait alokasi memori.

- Membuat Map dan Langsung Menginisialisasi Nilai Awal

```
1. myMap3 := map[string]int{  
2.     "satu": 1,  
3.     "dua": 2,  
4. }
```

Cara ini memungkinkan Map dibuat dan langsung diisi dengan data awal menggunakan sintaks literal. Seperti pada contoh diatas, Map diinisialisasi dengan beberapa pasangan *key-value* sekaligus. Metode ini berguna untuk deklarasi yang lebih ringkas dan memudahkan pembacaan kode saat data awal sudah diketahui.

E. ERROR HANDLING

Error handling dalam Go adalah teknik yang digunakan untuk menangani situasi yang tidak diinginkan atau tak terduga saat eksekusi program berlangsung. Dengan penanganan *error* yang baik, program dapat tetap berjalan dengan stabil tanpa langsung berhenti akibat kesalahan. Go menyediakan berbagai cara untuk menangani *error*, termasuk mekanisme pengecekan manual, *wrapping error* untuk pelacakan yang lebih baik, serta fitur *defer*, *panic*, dan *recover* untuk situasi yang lebih kompleks.

Pendekatan Menangani Error

Error dalam Go biasanya ditangani dengan memeriksa nilai *error* yang dikembalikan oleh suatu operasi. Dengan pendekatan ini, program bisa menghindari *crash* mendadak. Selain itu, membantu dalam mengidentifikasi sumber masalah dengan lebih jelas, sehingga *debugging* menjadi lebih mudah. Dengan menangani *error* secara eksplisit, program dapat memberikan pesan yang lebih informatif kepada pengguna atau sistem lain yang menggunakannya.

Contoh Program:

```
1. package main
2.
3. import (
4.     "errors"
5.     "fmt"
6. )
7.
8. func main() {
9.     var pembilang, penyebut int = 10, 0
10.    var hasil int
11.    var err error
12.
13.    if penyebut == 0 {
14.        err = fmt.Errorf("tidak bisa membagi dengan nol")
15.    } else {
16.        hasil = pembilang / penyebut
17.    }
18.
19.    if err != nil {
20.        fmt.Println("Terjadi error:", err)
21.    } else {
22.        fmt.Println("Hasil pembagian:", hasil)
23.    }
24. }
```

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run menanganiErr.go
Terjadi error: tidak bisa membagi dengan nol
```

Gambar 2.26 Pendekatan Menangani Error

Pada contoh diatas, program menangani *error* dengan memeriksa apakah nilai yang digunakan dalam pembagian adalah nol. Jika iya, *error* dibuat menggunakan `fmt.Errorf()`, sehingga pembagian tidak dilakukan dan pesan *error* ditampilkan. Jika tidak ada *error*, program akan mencetak hasil pembagian.

Membungkus Error (*Error Wrapping*)

Error wrapping digunakan untuk memberikan konteks tambahan pada *error* yang terjadi, sehingga lebih mudah untuk melacak penyebabnya.

Contoh Program:

```
1. package main
2.
3. import (
4.     "errors"
5.     "fmt"
6. )
7.
8. func main() {
9.     var file string = ""
10.    var err error
11. }
```

```

12.     if file == "" {
13.         err = fmt.Errorf("Gagal membaca file: %w", errors.New("nama file
   kosong"))
14.     }
15.
16.     if err != nil {
17.         fmt.Println("Terjadi error!", err)
18.     }
19. }
```

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run errWrapping.go
Terjadi error! Gagal membaca file: nama file kosong
```

Gambar 2.27 Error Wrapping

Untuk *error wrapping*, program memeriksa apakah variabel file kosong. Jika kosong, program membuat *error* baru dengan `fmt.Errorf()` dan menggunakan `%w` untuk menyertakan *error* asli. Ini membantu dalam pelacakan *error* dengan memberikan informasi lebih spesifik mengenai penyebabnya. Jika *error* terjadi, pesan *error* akan ditampilkan.

Pengantar Defer, Panic, dan Recover

Go memiliki mekanisme *defer* untuk menangani *error* yang bersifat kritis. *panic* menghentikan eksekusi, tetapi sebelum itu terjadi, kode yang telah ditunda dengan *defer* tetap akan dieksekusi. *recover* bisa menangkapnya. Selain itu, kombinasi ketiganya memungkinkan pengelolaan *error* yang lebih fleksibel, sehingga program tetap dapat menangani kondisi yang tidak terduga tanpa langsung terhenti.

Contoh Program:

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. func main() {
8.     // Menangani panic dengan defer dan recover
9.     defer func() {
10.         if r := recover(); r != nil {
11.             fmt.Println("Panic tertangkap:", r)
12.         }
13.     }
14.
15.     fmt.Println("Sebelum panic")
16.
17.     // Menyebabkan panic
18.     panic("Terjadi kesalahan fatal!")
19.
20.     fmt.Println("Setelah panic (tidak akan dieksekusi)")
21. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run def_pan_rec.go
Sebelum panic
Panic tertangkap: Terjadi kesalahan fatal!
```

Gambar 2.27 Pengantar Defer, Panic dan Recover

Mekanisme defer, panic, dan recover digunakan dalam penanganan *error* yang lebih kritis. defer memastikan bahwa jika panic terjadi, fungsi anonim yang berisi `recover()` akan dieksekusi. `panic("Terjadi kesalahan fatal!")` langsung menghentikan eksekusi program, tetapi berkat `recover()`, panic bisa ditangkap dan program tetap berjalan tanpa langsung berhenti.

Hal ini memungkinkan program untuk tetap memberikan respons yang sesuai daripada langsung berhenti tanpa peringatan. Dengan menggunakan pendekatan ini, pengembang dapat menangani skenario yang tidak terduga dengan lebih baik dan memberikan pesan *error* yang lebih informatif kepada pengguna.

Bab 3

FUNGSI

A. PENGENALAN FUNGSI

Fungsi dalam bahasa Go merupakan suatu blok kode yang dirancang untuk melakukan tugas tertentu secara struktur agar bisa digunakan berulang kali. Dalam pengembangan sebuah program fungsi sangat membantu pembagian program menjadi bagian-bagian kecil yang modular, sehingga mudah dipahami, diuji dan dirawat. Dalam bahasa Go konsep fungsi hampir sama dengan bahasa pemrograman lainnya, namun dengan beberapa karakteristik khas, seperti kemampuan untuk mengembalikan banyak nilai dalam satu waktu.

Fungsi dideklarasikan menggunakan kata kunci `func` yang diikuti dengan nama fungsi, daftar parameter dalam tanda kurung, dan tipe nilai kembalian. Kemudian bagian isi dari fungsi diletakkan dalam blok kurung kurawal (`{}`). Setelah itu fungsi dieksekusi dengan cara memanggilnya dengan kata kunci nama fungsinya diikuti kurung buka dan kurung tutup.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func cetakBilangan(n int) {
6.     for i := 1; i <= n; i++ {
7.         fmt.Println("Bilangan ke-", i)
8.     }
9. }
10.
11. func main() {
12.     cetakBilangan(5)
13. }
```

Deklarasi fungsi dilakukan sebelum `main`, fungsi yang dibuat yaitu `cetakBilangan` yang menerima satu parameter dengan tipe data integer yang diberi nama `n`. Fungsi ini menggunakan `for` untuk mencetak “Bilangan ke-” yang diikuti dengan angka 1 hingga `n`. Kemudian dalam fungsi `main`, fungsi `cetakBilangan` dipanggil dengan argumen 5.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run pengenalanFungsi.go
Bilangan ke- 1
Bilangan ke- 2
Bilangan ke- 3
Bilangan ke- 4
Bilangan ke- 5
```

Gambar 3.1 Pengenalan Fungsi

Dalam fungsi, sering dikenalkan dengan istilah parameter. Parameter fungsi merupakan variable yang digunakan untuk menerima nilai dari luar saat fungsi tersebut dipanggil. Parameter ini didefinisikan dalam tanda kurung setelah nama fungsi dan memiliki tipe data tertentu. Dengan menggunakan parameter, fungsi dapat menjadi lebih fleksible dan dapat digunakan kembali dengan berbagai nilai input.

Secara default, bahasa pemrograman Go menggunakan mekanisme *pass-by-value*, yang berarti nilai dari suatu argumen disalin ke dalam parameter fungsi. Oleh karena itu, perubahan pada parameter di dalam fungsi tidak akan mempengaruhi nilai asli dari argumen di luar fungsi. Namun, agar perubahan pada parameter mempengaruhi argumen asli, dapat menggunakan *pointer* sebagai parameter. Dengan itu, fungsi akan menerima alamat memori dari argumen dan memungkinkan perubahan langsung pada nilai aslinya.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func angkaTerbesar(a int, b int) {
6.     if a > b {
7.         fmt.Println(a, "adalah angka terbesar")
8.     } else if a < b {
9.         fmt.Println(b, "adalah angka terbesar")
10.    } else {
11.        fmt.Println("Kedua angka memiliki nilai yang sama")
12.    }
13. }
14.
15. func main() {
16.     angkaTerbesar(10, 4)
17. }
```

Fungsi `angkaTerbesar` menerima dua parameter yakni `a` dan `b` dengan tipe data *integer*. Kemudian dilakukan pemeriksaan menggunakan `if` untuk melakukan seleksi kondisi. Lalu pada fungsi `main`, `angkaTerbesar(10, 4)` dipanggil untuk menjalankan fungsi tersebut dengan nilai `a=10` dan `b=4`.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run fungsi2Param.go
10 adalah angka terbesar
```

Gambar 3.2 Fungsi dengan 2 Parameter

B. FUNGSI RETURN SINGLE DAN RETURN MULTIPLE VALUE

Return value merupakan Fungsi yang mengembalikan nilai dengan menggunakan kata kunci `return`. Terdapat dua jenis pengembalian nilai yakni *singel return value* dan *multiple return value*. Pada *singel value* nilai yang dikembalikan hanya satu, baik itu berupa angka, *string*, *boolean*, maupun tipe data lainnya. Contohnya, sebuah fungsi penjumlahan yang menerima dua bilangan dan mengembalikan hasilnya: `func tambah(a int, b int) int { return a + b }`. Di sini, fungsi hanya mengembalikan satu nilai bertipe `int`, yaitu hasil dari `a + b`. Penggunaan satu *return value* ideal untuk operasi yang menghasilkan satu hasil akhir yang pasti.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func luasPersegi(sisi int) int {
6.     return sisi * sisi
7. }
8.
9. func main() {
10.    hasil := luasPersegi(5)
11.    fmt.Println("Luas persegi adalah:", hasil)
12. }
```

Pada program diatas terdapat deklarasi fungsi `luasPersegi` yang menerima satu parameter bertipe `integer` dengan nama `sisi` dan mengembalikan hasil perkalian `sisi * sisi` yang merupakan rumus menghitung luas persegi. Fungsi ini kemudian dipanggil didalam `main` di mana `luasPersegi` dipanggil dengan argumen 5. Nilai hasil pehitungannya kemudian disimpan kedalam variabel `hasil` dan dicetak menggunakan perintah `fmt.Println` seperti output dibawah ini:

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run returnSingle.go
Luas persegi adalah: 25
```

Gambar 3.3 Fungsi Single Return

Sementara itu, *multiple value* memungkinkan sebuah fungsi mengembalikan dua atau lebih nilai sekaligus. Dalam situasi ketika ingin mengembalikan hasil utama bersamaan dengan informasi tambahan seperti error, *multi value* sangat berguna. Contohnya dalam operasi file, kita bisa punya fungsi `func bukaFile(nama string) (File, error)`, yang mengembalikan objek `File` jika berhasil dan objek `error` jika terjadi kesalahan.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func bagiDanSisa(a int, b int) (int, int) {
6.     hasil := a / b
7.     sisa := a % b
8.     return hasil, sisa
9. }
10.
10. func main() {
11.     hasil, sisa := bagiDanSisa(10, 3)
12.     fmt.Println("Hasil pembagian:", hasil)
13.     fmt.Println("Sisa bagi:", sisa)
14. }
```

Pada program diatas terdapat fungsi `bagiDanSisa` yang menerima dua parameter bertipe `integer`, yaitu `a` dan `b`. Fungsi ini menghitung hasil pembagian bulat dari `a / b` dan sisa bagi `a % b` lalu mengembalikannya sebagai dua nilai sekaligus. Pada blok kode `return`, (`hasil`) akan mengembalikan nilai ke parameter pertama kemudian (`sisa`) akan mengembalikan nilai ke parameter kedua. Di dalam main, fungsi `bagiDanSisa` dipanggil dengan argumen 10 dan 3, kemudian hasil dan sisa pembagian tersebut disimpan ke dalam variable `hasil` dan `sisa`. Kemudian program akan mencetak nilai hasil pembagian 3 dan sisa bagi 1 menggunakan perintah `fmt.Println`.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run multiReturn.go
Hasil pembagian: 3
Sisa bagi: 1
```

Gambar 3.4 Fungsi Multiple Return

C. FUNGSI REKURSIF

Fungsi rekursif adalah suatu fungsi yang memanggil dirinya sendiri dalam proses eksekusinya. Teknik ini digunakan untuk memecahkan masalah dengan cara membaginya menjadi submasalah yang lebih kecil dan serupa dengan masalah awal. Agar fungsi rekursif bekerja dengan benar, harus terdapat *base case* atau kasus dasar, yaitu kondisi yang menjadi titik berhenti dari pemanggilan berulang. Jika tidak ada *base case*, maka fungsi akan terus memanggil dirinya sendiri tanpa akhir, yang dikenal sebagai *infinite recursion* (rekursi tak berujung).

Contoh program:

```
1. package main
2.
3. import "fmt"
4.
5. func faktorial(n int) int {
6.     if n == 0 {
7.         return 1 // base case
8.     }
9.     return n * faktorial(n-1) // rekursif
10.
11. func main() {
12.     fmt.Println("Faktorial dari 5 adalah:", faktorial(5))
13. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run rekursif.go
Faktorial dari 5 adalah: 120
```

Gambar 3.5 Fungsi Rekursif

Dari *output* terlihat bahwa program berhasil mencetak hasil faktorial dari angka 5, yaitu 120. Nilai ini diperoleh melalui fungsi rekursif faktorial yang akan terus memanggil dirinya sendiri hingga mencapai kondisi dasar saat n sama dengan 0. Pada kondisi ini, fungsi mengembalikan nilai 1, kemudian hasilnya dikalikan bertahap hingga kembali ke nilai awal.

Namun, jika sebuah fungsi rekursif tidak memiliki kondisi dasar (*base case*), maka pemanggilan fungsi akan terus berlangsung tanpa henti dan menyebabkan program mengalami error (*stack overflow*). Berikut contoh kode program yang tidak memiliki *base case*.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func tanpaAkhir(n int) int {
6.     fmt.Println("Memanggil dengan n =", n)
7.     return n + tanpaAkhir(n-1) // Tidak ada base case
8. }
9.
10. func main() {
11.     hasil := tanpaAkhir(5)
12.     fmt.Println("Hasil:", hasil)
13. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run rekursif.go
Memanggil dengan n = 5
Memanggil dengan n = 4
Memanggil dengan n = 3
Memanggil dengan n = 2
Memanggil dengan n = 1
Memanggil dengan n = 0
Memanggil dengan n = -1
Memanggil dengan n = -2
Memanggil dengan n = -3
Memanggil dengan n = -4
Memanggil dengan n = -5
Memanggil dengan n = -6
Memanggil dengan n = -7
Memanggil dengan n = -8
Memanggil dengan n = -9
```

Gambar 3.6 Fungsi Rekursif Tanpa Batas (Tidak Ada Base Case)

Dapat dilihat dari *Output*, kode tersebut akan menyebabkan *stack overflow* karena fungsi `tanpaAkhir(5)` dipanggil secara rekursif tanpa kondisi berhenti (*base case*). Setiap pemanggilan akan memanggil dirinya sendiri terus-menerus dengan nilai `n` yang semakin kecil.

D. DEFER, PANIC dan RECOVER

Defer

Dalam Go, terdapat sebuah pernyataan khusus yang disebut defer. Pernyataan ini digunakan untuk menunda eksekusi suatu fungsi, hingga fungsi tempat pernyataan defer tersebut berada selesai dijalankan atau telah mencapai titik pengembalian nilai (*return*). Artinya, fungsi yang dipanggil dengan defer tidak akan langsung dieksekusi pada saat baris kode tersebut dibaca, melainkan akan dieksekusi tepat sebelum fungsi induknya berakhir, tidak peduli bagaimana fungsi tersebut berakhir.

Contoh Program:

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. func main() {
8.     fmt.Println("Mulai")
9.     defer fmt.Println("Baris ini dieksekusi terakhir")
10.    fmt.Println("Sedang berjalan")
11. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run defer.go
Mulai
Sedang berjalan
Baris ini dieksekusi terakhir
```

Gambar 3.7 Pengenalan Defer

Dari *output* tersebut terlihat bahwa meskipun pernyataan defer muncul sebelum baris ketiga ("Sedang berjalan"), namun eksekusinya ditunda hingga fungsi *main* benar-benar akan berakhir. Fungsi yang dipanggil menggunakan defer *statement* juga akan selalu dijalankan.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func study() {
6.     fmt.Println("Semangat Belajar Golang!")
7. }
8.
9. func main() {
10.    defer study()
11.    div := 0
12.    result := 3 / div
13.    fmt.Println(result)
14. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run defer.go
Semangat Belajar Golang!
panic: runtime error: integer divide by zero

goroutine 1 [running]:
main.main()
    C:/A/UM/VS CODE/6_Go/src/defer.go:16 +0x30
exit status 2
```

Gambar 3.8 Penggunaan Defer

Panic

Dalam Go, terdapat fungsi bawaan yang disebut *panic*, yang berfungsi untuk menghentikan jalannya program secara langsung. Ketika fungsi *panic* dipanggil, program akan segera masuk menjalankan proses *unwinding*, yaitu membatalkan kumpulan eksekusi fungsi (*stack*) yang sedang berjalan. Namun demikian, sebelum program benar-benar dihentikan secara paksa, Go akan terlebih dahulu mengeksekusi semua pernyataan *defer* yang telah didaftarkan dalam fungsi-fungsi yang masih berada di dalam *stack*. Berikut adalah contoh *panic* fungsi.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func hello() {
6.     fmt.Println("Hello Sahabat!")
7. }
8.
9. func main() {
10.    defer hello()
11.
12.    panic("Panic dipanggil")
13.    fmt.Println("Tidak akan dieksekusi")
14. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run panic.go
Hello Sahabat!
panic: Panic dipanggil

goroutine 1 [running]:
main.main()
    C:/A/UM/VS CODE/6_Go/src/panic.go:15 +0x3e
exit status 2
```

Gambar 3.9 Panic

Dari *output* tersebut terlihat bahwa meskipun program dihentikan secara tiba-tiba menggunakan *panic*, fungsi *hello()* tetap dijalankan terlebih dahulu karena dipanggil menggunakan *defer* yang akan memastikan bahwa fungsi yang ditandainya akan dieksekusi tepat sebelum fungsi induknya keluar, baik keluar secara normal maupun karena *error*. Oleh

karena itu baris `fmt.Println("Tidak akan dieksekusi")` tidak dijalankan karena program sudah dihentikan oleh panic, tetapi fungsi `hello()` tetap muncul dalam *output*.

Recover

Recover adalah sebuah fungsi bawaan dalam Go yang digunakan untuk menangani kondisi panic secara terkontrol. Fungsi ini memungkinkan program untuk “pulih” dari keadaan panic, sehingga alur eksekusi tidak langsung dihentikan secara mendadak. Dengan kata lain, recover berfungsi untuk menangkap nilai yang dikirim oleh panic, dan mencegah program dari penghentian yang tidak diinginkan. Berikut adalah contoh recover Fungsi.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func checkSudahBelajar(sudahBelajar bool) {
6.     if sudahBelajar == false {
7.         panic("Ups, kamu belum belajar")
8.     }
9.     fmt.Println("Mantap, sudah belajar")
10. }
11.
12. func selesai() {
13.     er := recover()
14.
15.     if er != nil {
16.         fmt.Println("Ada eror:", er)
17.     }
18.
19.     fmt.Println("Selesai")
20. }
21.
22. func main() {
23.     defer selesai()
24.     checkSudahBelajar(false)
25. }
```

Output:

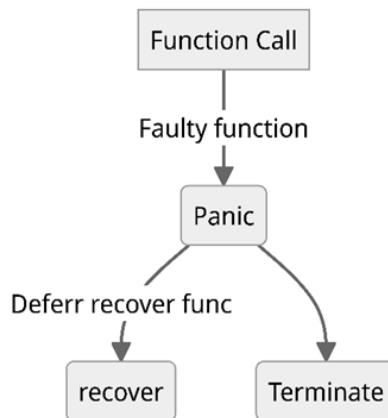
```
PS C:\A\UM\VS CODE\6_Go\src> go run recover.go
Ada eror: Ups, kamu belum belajar
Selesai
```

Gambar 3.10 Recover

Dari contoh tersebut, dapat dilihat bahwa meskipun fungsi `checkSudahBelajar` memicu kondisi panic saat menerima nilai `false`, program tidak langsung berhenti secara tiba-tiba. Hal ini disebabkan oleh adanya pernyataan `defer selesai()` dalam fungsi `main()`, yang memastikan bahwa fungsi selesai tetap dijalankan sebelum program berakhir. Di dalam fungsi tersebut, mekanisme `recover()` digunakan untuk menangkap pesan kesalahan yang ditimbulkan oleh panic.

Integrasi Penggunaan *Defer*, *Panic* dan *Recover*

Dalam bahasa Go, defer, panic, dan recover merupakan tiga konsep yang saling terintegrasi untuk menangani eksekusi akhir dan kondisi kesalahan (*error*) dalam program. Ketiganya bekerja secara berurutan dan saling melengkapi, terutama dalam konteks pemulihan dari *runtime error* tanpa harus menghentikan seluruh program secara paksa.



Gambar 3.11 Integrasi Defer, Panic dan Recover

Sumber: AdiTechSavvys Blogs on Golang

Contoh Program:

```
1. package main
2.
3. import (
4.     "fmt"
5.
6.     func tanganiPanic() {
7.         if pesan := recover(); pesan != nil {
8.             fmt.Println("Terjadi kesalahan:", pesan)
9.         }
10.    }
11.
12. func bagi(a, b int) {
13.     defer tanganiPanic()
14.     defer fmt.Println("Fungsi selesai dijalankan.")
15.
16.     fmt.Printf("Membagi %d dengan %d\n", a, b)
17.     if b == 0 {
18.         panic("Tidak bisa dibagi dengan nol!")
19.     }
20.     hasil := a / b
21.     fmt.Println("Hasil:", hasil)
22. }
23.
24. func main() {
25.
26.     bagi(10, 2)
27.
28.     fmt.Println()
29.
30.     bagi(5, 0)
31.     fmt.Println("\nProgram selesai tanpa crash.")
32. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run def_pan_rec.go
Membagi 10 dengan 2
Hasil: 5
Fungsi selesai dijalankan.

Membagi 5 dengan 0
Fungsi selesai dijalankan.
Terjadi kesalahan: Tidak bisa dibagi dengan nol!

Program selesai tanpa crash.
```

Gambar 3.12 Integrasi Penggunaan Defer, Panic dan Recover

Dari *output* tersebut terlihat bahwa meskipun terjadi *error* karena pembagian dengan nol dan program mengalami panic, baris `Fungsi selesai dijalankan`, tetap muncul. Hal ini karena baris tersebut ditempatkan menggunakan defer, yang menjamin akan tetap dijalankan sebelum fungsi keluar, baik karena selesai secara normal maupun karena panic.

Selain itu, fungsi `tanganiPanic()` juga dijalankan sebelum keluar karena dipanggil dengan defer. Fungsi ini menggunakan `recover()` untuk menangkap panic, sehingga program tidak langsung berhenti secara paksa, melainkan melanjutkan eksekusi ke baris selanjutnya. Baris `fmt.Println("\nProgram selesai tanpa crash.")` tetap dijalankan karena panic berhasil ditangani.

Dengan demikian, meskipun ada *error* dalam salah satu proses pembagian, program tetap berjalan secara normal setelahnya karena adanya kombinasi defer, panic, dan recover

BAB 4

POINTER

A. OPERATOR ADDRESS

Dalam bahasa Go, operator & disebut sebagai operator *address*. Operator ini digunakan untuk mengambil alamat memori dari suatu variabel. Ketika sebuah variabel diawali dengan &, maka hasilnya adalah alamat tempat variabel tersebut disimpan di dalam memori. Alamat ini bisa disimpan dalam sebuah *pointer* agar dapat digunakan kembali dalam proses lain, seperti pemanggilan fungsi atau manipulasi nilai secara langsung melalui alamat.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var a int = 42
7.     fmt.Println("Alamat variabel a:", &a)
8. }
```

Kode di atas merupakan program sederhana dalam bahasa Go yang mendeklarasikan sebuah variabel *integer* bernama *a* dengan nilai 42, lalu mencetak alamat memori tempat variabel *a* disimpan menggunakan operator &. Fungsi *fmt.Println* digunakan untuk menampilkan hasil ke layar, dan bagian *&a* berfungsi untuk mengambil alamat memori dari variabel *a*, bukan nilainya. Dengan demikian, *output* dari program ini adalah alamat memori dari *a*, yang berguna untuk memahami konsep *pointer* atau pengaksesan data melalui alamat memori di bahasa Go.

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run address.go
Alamat variabel a: 0xc00000a0d8
```

Gambar 4.1 Operator Address

B. PENGENALAN POINTER

Setelah memahami bagaimana mendapatkan alamat suatu variabel dengan operator &, selanjutnya kita masuk pada konsep *pointer* itu sendiri. *Pointer* adalah variabel yang menyimpan alamat memori dari variabel lain. Dalam bahasa Go, *pointer* dideklarasikan dengan menambahkan simbol * di depan tipe data, misalnya *int berarti *pointer* ke *integer*.

Pointer memiliki dua operasi penting:

1. Menyimpan alamat variabel lain: Misalnya *p* = *&x*.
2. Mengakses atau mengubah nilai di alamat tersebut: Dilakukan dengan operator *dereferensi* *, misalnya **p* = 20.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var x int = 10
7.     var p *int
8.
9.     p = &x
10.
11.    fmt.Println("Alamat memori x:", p)
12.    fmt.Println("Nilai x melalui pointer:", *p)
13.
14.    *p = 20
15.    fmt.Println("Nilai x setelah diubah melalui pointer:", x)
16. }
```

Program ini menunjukkan konsep dasar penggunaan pointer dalam bahasa Go. Pertama, variabel `x` dideklarasikan dengan tipe `int` dan diberi nilai awal `10`. Kemudian, variabel `p` dideklarasikan sebagai pointer ke integer. Setelah itu, `p` menyimpan alamat memori dari `x`, sehingga memungkinkan akses ke nilai `x` melalui pointer. Dengan menggunakan operator dereferensi `*p`, nilai yang tersimpan di alamat memori dapat diakses dan diubah. Ketika nilai `*p` diubah menjadi `20`, nilai `x` juga ikut berubah karena keduanya merujuk pada alamat yang sama. Sehingga pointer dapat digunakan untuk mengubah nilai dari variabel yang ditunjuk tanpa harus langsung mengakses variabel tersebut. Pointer berguna untuk mengelola memori secara efisien dan sering digunakan dalam struktur data yang kompleks.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run pointer.go
Alamat memori x: 0xc00000a0d8
Nilai x melalui pointer: 10
Nilai x setelah diubah melalui pointer: 20
```

Gambar 4.2 Pengenalan Pointer

C. POINTER DI FUNGSI

Konsep *pointer* memungkinkan kita untuk membuat program yang dapat mengubah isi variabel di lokasi lain. Hal ini sangat berguna ketika kita ingin menghemat penggunaan memori (dengan tidak membuat salinan variabel), atau saat membuat fungsi yang bisa mengubah nilai aslinya tanpa harus mengembalikannya. Contoh sederhananya adalah saat kita ingin menukar dua nilai dengan fungsi. Jika kita hanya mengirim nilai (*pass by value*), maka perubahan tidak akan berdampak pada variabel aslinya. Tapi jika kita kirim alamatnya (*pass by reference* menggunakan *pointer*), maka perubahan akan langsung berlaku.

Pass By Value

Pass by value adalah metode pengiriman parameter ke dalam fungsi di mana nilai dari argumen yang dikirim akan disalin ke parameter fungsi. Artinya, fungsi akan bekerja dengan salinan (*copy*) dari data yang diberikan, bukan data aslinya. Oleh karena itu, perubahan yang dilakukan terhadap parameter di dalam fungsi tidak akan memengaruhi nilai variabel asli di luar fungsi.

Go secara default menggunakan *pass by value* untuk semua jenis data, termasuk variabel primitif seperti `int`, `float`, `bool`, maupun struktur data seperti `struct`. Jika kamu ingin agar fungsi bisa memodifikasi nilai asli dari variabel, maka kamu perlu mengirimkan alamat memori dari variabel tersebut, yaitu menggunakan pointer.

Contoh Program :

```
1. package main
2.
3. import "fmt"
4.
5. func ubahNilai(x int) {
6.     x = x + 10
7.     fmt.Println("Nilai di dalam fungsi:", x)
8. }
9.
10. func main() {
11.     nilai := 5
12.     ubahNilai(nilai)
13.     fmt.Println("Nilai setelah fungsi dipanggil:", nilai)
14. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run passByValue.go
Nilai di dalam fungsi: 15
Nilai setelah fungsi dipanggil: 5
```

Gambar 4.3 Pass By Value

Pada contoh kode di atas, kita mendefinisikan sebuah fungsi bernama `ubahNilai` yang menerima satu parameter bertipe `int` bernama `x`. Di dalam fungsi tersebut, kita menambahkan 10 ke dalam nilai `x`, lalu mencetak nilai `x` setelah diubah. Fungsi ini bertujuan untuk menunjukkan bahwa nilai variabel dapat dimanipulasi di dalam fungsi. Namun, penting untuk dicatat bahwa parameter `x` di dalam fungsi hanyalah salinan dari nilai variabel asli yang dikirim dari fungsi `main`. Dalam fungsi `main`, kita mendeklarasikan variabel `nilai` dan memberinya nilai 5. Kemudian kita memanggil fungsi `ubahNilai` dengan mengirimkan `nilai` sebagai argumen. Nilai 5 dari `nilai` disalin ke parameter `x` di dalam fungsi, sehingga setiap perubahan yang terjadi terhadap `x` tidak berdampak pada variabel `nilai` di luar fungsi. Setelah fungsi dijalankan, saat kita mencetak `nilai`, hasilnya tetap 5, membuktikan bahwa perubahan yang dilakukan di dalam fungsi tidak mempengaruhi nilai asli. Hal ini menggambarkan dengan jelas bagaimana mekanisme *pass by value* bekerja dalam bahasa Go.

Pass By Reference

Pass by reference adalah sebuah cara untuk *passing* parameter ke dalam fungsi di mana yang dikirimkan ke dalam fungsi atau method adalah alamat memori dari parameter. Dengan demikian, fungsi tidak bekerja pada salinan nilai, melainkan langsung terhadap variabel aslinya. Artinya, jika ada perubahan yang dilakukan pada parameter di dalam fungsi atau method, maka perubahan tersebut akan langsung memengaruhi nilai dari parameter asli yang berada di luar fungsi.

Mekanisme ini sangat berguna ketika kita ingin melakukan modifikasi langsung terhadap data yang dikirim ke dalam fungsi, tanpa perlu mengembalikannya melalui nilai *return*. Bahasa Go tidak menyediakan *pass by reference* secara eksplisit seperti bahasa C++, tetapi mendukungnya melalui penggunaan *pointer*, yaitu variabel yang menyimpan alamat memori dari variabel lain. Dengan memberikan *pointer* sebagai parameter ke dalam fungsi, kita bisa mengakses dan mengubah nilai asli dari variabel tersebut.

Penggunaan *pass by reference* juga sangat efisien dalam hal performa, terutama saat berhadapan dengan data berukuran besar seperti *array* atau struktur kompleks (*struct*), karena tidak perlu menyalin seluruh isi data ke dalam fungsi. Dengan hanya mengirim alamat memori, pemrosesan menjadi lebih ringan dan hemat memori. Maka dari itu, memahami *pointer* sebagai sarana untuk *pass by reference* sangat penting dalam pengembangan program Go yang efisien dan optimal.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. func ubahNilai(x *int) {
6.     *x = *x + 10
7.     fmt.Println("Nilai di dalam fungsi:", *x)
8. }
9.
10. func main() {
11.     nilai := 5
12.     ubahNilai(&nilai)
13.     fmt.Println("Nilai setelah fungsi dipanggil:", nilai)
14. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run passByRef.go
Nilai di dalam fungsi: 15
Nilai setelah fungsi dipanggil: 15
```

Gambar 4.4 Pass By Reference

Dalam kode tersebut, kita mendeklarasikan sebuah fungsi bernama `ubahNilai` yang menerima parameter bertipe `*int`, yaitu *pointer* ke tipe `int`. Artinya, parameter `x` akan berisi alamat memori dari variabel bertipe `int`, bukan nilainya langsung. Saat fungsi dipanggil dari

fungsi `main`, kita mengirimkan `&nilai`, yaitu alamat dari variabel `nilai`. Dengan begitu, fungsi `ubahNilai` bekerja langsung terhadap variabel `nilai`, bukan salinannya.

Di dalam fungsi `ubahNilai`, kita menggunakan `*x` untuk mengakses nilai yang ada di alamat yang ditunjuk oleh `x`. Lalu kita menambahkan 10 ke nilai tersebut. Misalnya, awalnya `nilai` bernilai 5, maka setelah `*x = *x + 10`, nilainya menjadi 15. Karena perubahan ini dilakukan langsung terhadap memori `nilai`, maka ketika program kembali ke fungsi `main`, nilai `nilai` benar-benar telah berubah menjadi 15. Hal ini membuktikan bahwa dengan menggunakan *pointer*, kita bisa mengubah nilai variabel dari dalam fungsi.

Penggunaan *pass by reference* ini sangat penting dalam berbagai situasi, seperti ketika kita ingin membuat fungsi yang memperbarui beberapa nilai sekaligus, atau ketika menangani struktur data besar tanpa perlu menyalin seluruh isi data.

D. POINTER DI METHOD

Dalam bahasa Go, *method* dapat memiliki *receiver* berupa nilai (*value receiver*) atau *pointer* (*pointer receiver*). Penggunaan *pointer receiver* memungkinkan method untuk:

1. Mengubah nilai asli dari *receiver*

Dengan *pointer receiver*, method dapat memodifikasi data asli dari objek yang menerima method tersebut.

2. Menghindari penyalinan data besar

Pointer receiver menghindari penyalinan seluruh data *struct* saat method dipanggil, yang lebih efisien untuk *struct* berukuran besar.

3. Konsistensi dalam method set

Jika suatu *type* memiliki method dengan *pointer receiver*, maka untuk konsistensi, sebaiknya semua method untuk *type* tersebut juga menggunakan *pointer receiver*.

Contoh Program:

```
1. package main
2.
3. import "fmt"
4.
5. type Mahasiswa struct {
6.     Nama string
7.     Umur int
8. }
9.
10. func (m *Mahasiswa) TambahUmur() {
11.     m.Umur += 1
12. }
13.
14. func main() {
```

```
15.     mhs := Mahasiswa{Nama: "Safira", Umur: 20}
16.     fmt.Println("Sebelum:", mhs.Umur)
17.     mhs.TambahUmur()
18.     fmt.Println("Sesudah:", mhs.Umur)
19. }
```

Pada contoh di atas, *struct* Mahasiswa memiliki *field* Nama dan Umur. Method *TambahUmur()* menggunakan *receiver* **Mahasiswa*, artinya ia dapat mengakses dan mengubah nilai asli dari objek Mahasiswa.

Saat method *TambahUmur()* dipanggil, ia menambahkan umur sebesar 1 secara langsung pada data mhs, bukan salinan. Ini mungkin karena *receiver*-nya berbentuk alamat (**Mahasiswa*), sehingga perubahan terjadi langsung di memori objek yang dipanggil. Dengan kata lain, *pointer receiver* memungkinkan method mengakses alamat memori dari *instance* yang dipanggil, bukan hanya menyalin isinya, sehingga sangat berguna ketika ingin memodifikasi isi *struct* dari dalam method.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run pointer.go
Sebelum: 20
Sesudah: 21
```

Gambar 4.5 Pointer di Method

BAB 5

MODULE DAN PACKAGE

A. PENGENALAN MODULE

Dalam bahasa pemrograman Go, module adalah unit independen dari kode yang berisi kumpulan paket dan dikelola menggunakan `go mod`. Modul memungkinkan pengelolaan dependensi dengan cara yang lebih terstruktur dan memudahkan pembagian serta penggunaan kode dalam proyek yang lebih besar. Dengan adanya `go mod`, pengembang dapat mengelola versi dependensi eksternal dan memastikan proyek tetap stabil tanpa konflik versi.

Sementara itu, package adalah kumpulan kode yang memiliki fungsi tertentu dan dapat digunakan kembali dalam program. Package memungkinkan organisasi kode yang lebih baik, sehingga program menjadi modular dan mudah dikelola. Go memiliki berbagai package bawaan, tetapi pengembang juga bisa membuat custom package sesuai kebutuhan. Mengimpor package dengan tepat membantu memanfaatkan fitur yang telah tersedia tanpa harus menulis ulang logika dari awal.

Selain itu, module dalam Go juga bertindak sebagai unit pengelolaan versi yang memungkinkan proyek untuk menentukan dependensi spesifik berdasarkan versi yang diinginkan. Dengan pendekatan ini, module memberikan fleksibilitas bagi pengembang untuk berbagi dan menggunakan kembali kode secara efisien, serta memungkinkan kolaborasi yang lebih baik dalam ekosistem Go.

B. PENGELOLAAN MODULE

Go menggunakan `go mod` sebagai sistem manajemen modul untuk mengelola dependensi dan versi paket secara otomatis. Modul memungkinkan proyek tetap konsisten tanpa harus menyimpan semua dependensi secara manual di dalam repositori. Dengan `go mod`, pengembang dapat menginisialisasi, membersihkan, dan mengunduh dependensi yang diperlukan untuk proyek.

```
go mod init myProject1  
go mod tidy  
go mod download
```

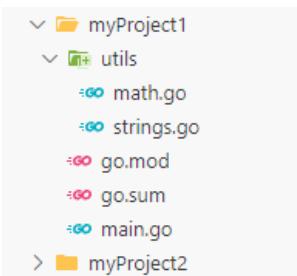
Gambar 5.1 Pengelolaan Module

- `go mod init myproject1` membuat modul baru dengan nama `myproject1`, memungkinkan pengelolaan dependensi.
- `go mod tidy` membersihkan daftar dependensi, hanya menyisakan yang benar-benar digunakan dalam proyek.
- `go mod download` mengunduh semua dependensi yang tercantum dalam `go.mod` agar siap digunakan.

C. STRUKTUR DIREKTORI GO

Struktur direktori yang baik dalam proyek Go memastikan bahwa kode mudah dikelola dan diorganisir. Biasanya, proyek terdiri dari file utama (`main.go`), modul (`go.mod`), dan berbagai package tambahan dalam folder terpisah. Dengan struktur yang jelas, pengembang dapat dengan mudah mengakses dan menggunakan kode tanpa kehilangan arah.

Contoh:



Gambar 5.2 Contoh Direktori di Go

- `go.mod` menyimpan informasi modul dan dependensi.
- `go.sum` berisi checksum dari semua dependensi.
- `main.go` adalah file utama tempat eksekusi program dimulai.
- `utils/` adalah folder package yang menyimpan beberapa file dengan fungsi yang lebih spesifik.

D. MENGGUNAKAN CUSTOM PACKAGE

Pengembang dapat membuat package sendiri agar fungsi tertentu dapat digunakan kembali dalam program tanpa harus menulis ulang kode. Dengan pendekatan ini, kode menjadi lebih modular dan mudah di-maintain.

Contoh Program :

File: `utils/math.go`

```
1. package utils
2.
3. func Tambah(a, b int) int {
4.     return a + b
5. }
```

File: `main.go`

```
1. package main
2.
```

```
3. import (
4.     "fmt"
5.     "myproject/utils"
6. )
7.
8. func main() {
9.     hasil := utils.Tambah(3, 5)
10.    fmt.Println("Hasil penjumlahan:", hasil)
11. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src\myProject1> go run main.go
Hasil penjumlahan: 8
```

Gambar 5.3 Menggunakan Custom Package

Package `utils` berisi fungsi sederhana `Tambah()` yang menerima dua angka dan mengembalikan hasil penjumlahan. Di dalam `main.go`, package `utils` diimpor agar fungsinya bisa digunakan dalam program utama. Dengan cara ini, kode untuk operasi matematika dasar bisa ditempatkan dalam package khusus, sehingga meningkatkan modularitas proyek.

E. MENGIMPOR PACKAGE STANDAR

Go menyediakan banyak package standar yang dapat langsung digunakan tanpa harus diinstal terlebih dahulu. Beberapa di antaranya adalah `fmt` untuk input *output*, `math` untuk operasi matematika, `time` untuk manajemen waktu dan lainnya.

Contoh Program:

```
1. import (
2.     "fmt"
3.     "math"
4.     "time"
5. )
6.
7. fmt.Println("Akar dari 16:", math.Sqrt(16))
8. fmt.Println("Waktu saat ini:", time.Now())
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run pckgImport.go
Akar dari 16: 4
Waktu saat ini: 2025-05-17 00:19:49.9487253 +0700 +07 m=+0.001478101
```

Gambar 5.4 Mengimpor Package Standar

Pada contoh ini, program menggunakan beberapa package standar untuk berbagai keperluan. `math.Sqrt(16)` menghitung akar kuadrat dari angka 16, sedangkan `time.Now()` digunakan untuk menampilkan waktu saat ini. Dengan mengimpor package standar, pengembang dapat memanfaatkan fitur bawaan Go tanpa harus membuat ulang fungsionalitas tersebut dari awal.

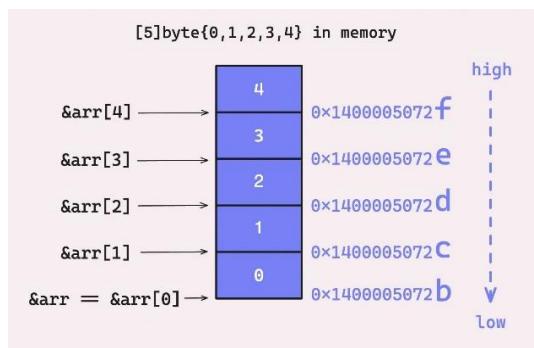
BAB 6

STRUKTUR DATA BAWAAN

A. ARRAY

Dalam bahasa pemrograman Go (Golang), array adalah struktur data statis yang menyimpan kumpulan elemen dengan tipe data yang sama. Ukuran array ditentukan saat deklarasi dan tidak dapat diubah setelahnya. Array digunakan untuk menyimpan data dalam jumlah tetap yang diketahui sebelumnya, seperti daftar nilai, nama, atau angka. Perbedaan utama antara array dan slice di Go adalah bahwa array memiliki ukuran tetap dan dimasukkan sebagai nilai, sedangkan slice bersifat dinamis dan direferensikan.

Alokasi memori untuk array dilakukan secara berurutan di memori, sehingga setiap elemen array memiliki alamat yang berdekatan. Karena sifatnya yang statis, array dialokasikan di stack atau heap tergantung pada scope penggunaannya, yang berpengaruh pada efisiensi penggunaan memori.



Gambar 6.1 Ilustrasi Array dengan Alokasi Memori
Sumber: VictoriaMetrics

Kelebihan :

- Struktur Sederhana dan Statis. Array memiliki struktur data yang mudah dipahami dan digunakan, cocok untuk data yang jumlahnya sudah pasti dari awal.
- Akses Cepat via Indeks. Elemen dalam array dapat diakses langsung menggunakan indeks dengan waktu akses konstan ($O(1)$), menjadikannya sangat efisien untuk membaca dan menulis data.

Kekurangan:

- Ukuran Tetap (Tidak Dinamis). Setelah array dideklarasikan, ukurannya tidak bisa diubah. Ini tidak fleksibel jika jumlah data yang disimpan bisa berubah-ubah.
- Kurang Efisien untuk Operasi Kompleks. Untuk keperluan seperti menambah atau menghapus data di tengah, array kurang efisien dibandingkan slice.
- Tidak *Reusable* untuk Data Bertipe Berbeda. Array di Go harus memiliki satu tipe data yang seragam, sehingga tidak bisa digunakan untuk menyimpan berbagai jenis data dalam satu array.

Contoh Program yang Mengimplementasikan Array:

```
1. package main
2.
3. import (
4.     "fmt"
5.     "os"
6.     "sort"
7. )
8.
9. func main() {
10.     const jumlahSiswa = 3
11.
12.     var nama [jumlahSiswa]string
13.     var tugas [jumlahSiswa]float64
14.     var uts [jumlahSiswa]float64
15.     var uas [jumlahSiswa]float64
16.     var nilaiAkhir [jumlahSiswa]float64
17.     var status [jumlahSiswa]string
18.
19.     fmt.Println("==== INPUT DATA 3 SISWA ===")
20.     for i := 0; i < jumlahSiswa; i++ {
21.         fmt.Printf("\nSiswa ke-%d:\n", i+1)
22.         fmt.Print("Nama : ")
23.         fmt.Scanln(&nama[i])
24.         fmt.Print("Nilai Tugas : ")
25.         fmt.Scanln(&tugas[i])
26.         fmt.Print("Nilai UTS : ")
27.         fmt.Scanln(&uts[i])
28.         fmt.Print("Nilai UAS : ")
29.         fmt.Scanln(&uas[i])
30.
31.         nilaiAkhir[i] = tugas[i]*0.3 + uts[i]*0.3 + uas[i]*0.4
32.
33.         if nilaiAkhir[i] >= 70 {
34.             status[i] = "LULUS"
35.         } else {
36.             status[i] = "TIDAK LULUS"
37.         }
38.     }
39.
40.     file, err := os.Create("hasil_nilai.txt")
41.     if err != nil {
42.         fmt.Println("Gagal menyimpan file:", err)
43.         return
44.     }
45.     defer file.Close()
46.
47.     header := fmt.Sprintf("%-5s %-10s %-6s %-6s %-6s %-10s %-15s\n",
48.         "No", "Nama", "Tgs", "UTS", "UAS", "Akhir", "Status")
49.     fmt.Println("\n==== REKAP NILAI SISWA ===")
50.     fmt.Print(header)
51.     file.WriteString("==== REKAP NILAI SISWA ===\n")
52.     file.WriteString(header)
53.
54.     for i := 0; i < jumlahSiswa; i++ {
55.         baris := fmt.Sprintf("%-5d %-10s %-6.1f %-6.1f %-6.1f %-10.1f %-15s\n",
56.             i+1, nama[i], tugas[i], uts[i], uas[i], nilaiAkhir[i], status[i])
57.         fmt.Print(baris)
```

```

57.     file.WriteString(baris)
58. }
59.
60. type siswaData struct {
61.     nama      string
62.     nilaiAkhir float64
63. }
64.
65. var ranking []siswaData
66. for i := 0; i < jumlahSiswa; i++ {
67.     ranking = append(ranking, siswaData{nama: nama[i], nilaiAkhir:
68.         nilaiAkhir[i]})  

69. }
70. sort.Slice(ranking, func(i, j int) bool {
71.     return ranking[i].nilaiAkhir > ranking[j].nilaiAkhir
72. })
73.
74. fmt.Println("\n==== PERINGKAT SISWA ===")
75. file.WriteString("\n==== PERINGKAT SISWA ===\n")
76. for i, s := range ranking {
77.     baris := fmt.Sprintf("Peringkat %d: %-10s (%.1f)\n", i+1, s.nama,
78.     s.nilaiAkhir)
79.     fmt.Print(baris)
80.     file.WriteString(baris)
81. }
82. fmt.Println("\n✓ Data berhasil disimpan ke file 'hasil_nilai.txt'")
83. }

```

Penjelasan Kode Tiap Bagian

- Mengimpor pustaka yang dibutuhkan

```

3. import (
4.     "fmt"
5.     "os"
6.     "sort"
7. )

```

Program menggunakan tiga paket bawaan Go. `fmt` digunakan untuk *input/output*, `os` untuk pembuatan dan penulisan file, dan `sort` untuk mengurutkan nilai siswa berdasarkan nilai akhir.

- Deklarasi variabel

```

10. const jumlahSiswa = 3
11.
12. var nama [jumlahSiswa]string
13. var tugas [jumlahSiswa]float64
14. var uts [jumlahSiswa]float64
15. var uas [jumlahSiswa]float64
16. var nilaiAkhir [jumlahSiswa]float64
17. var status [jumlahSiswa]string

```

`jumlahSiswa` ditentukan sebanyak tiga. Selanjutnya, program membuat *array* tetap untuk menyimpan informasi nama, nilai tugas, UTS, UAS, nilai akhir, dan status kelulusan masing-masing siswa.

- Input data dan perhitungan nilai akhir

```
19. fmt.Println("==== INPUT DATA 3 SISWA ===")
20. for i := 0; i < jumlahSiswa; i++ {
21.     fmt.Printf("\nSiswa ke-%d:\n", i+1)
22.     fmt.Print("Nama : ")
23.     fmt.Scanln(&nama[i])
24.     fmt.Print("Nilai Tugas : ")
25.     fmt.Scanln(&tugas[i])
26.     fmt.Print("Nilai UTS : ")
27.     fmt.Scanln(&uts[i])
28.     fmt.Print("Nilai UAS : ")
29.     fmt.Scanln(&uas[i])
30.
31.     nilaiAkhir[i] = tugas[i]*0.3 + uts[i]*0.3 + uas[i]*0.4
32.
33.     if nilaiAkhir[i] >= 70 {
34.         status[i] = "LULUS"
35.     } else {
36.         status[i] = "TIDAK LULUS"
37.     }
38. }
```

Program meminta *input* dari pengguna berupa nama siswa dan tiga jenis nilai. Nilai akhir dihitung dengan rumus pembobotan: tugas 30%, UTS 30%, dan UAS 40%. Berdasarkan nilai akhir, status kelulusan ditentukan.

- Pembuatan file hasil_nilai.txt

```
40. file, err := os.Create("hasil_nilai.txt")
41. if err != nil {
42.     fmt.Println("Gagal menyimpan file:", err)
43.     return
44. }
45. defer file.Close()
```

Program membuat file teks untuk menyimpan hasil rekap. Jika file tidak bisa dibuat, program akan menampilkan pesan error dan berhenti. Perintah `defer` digunakan agar file ditutup otomatis setelah fungsi `main` selesai dijalankan.

- Menulis header tabel ke layar dan file

```
47. header := fmt.Sprintf("%-5s %-10s %-6s %-6s %-6s %-10s %-15s\n",
48.     "No", "Nama", "Tgs", "UTS", "UAS", "Akhir", "Status")
49. Zmt.Println("\n==== REKAP NILAI SISWA ===")
50. fmt.Print(header)
51. file.WriteString("==== REKAP NILAI SISWA ===\n")
```

Baris ini menampilkan *header* tabel yang merinci kolom nomor, nama, nilai tugas, UTS, UAS, nilai akhir, dan status. *Header* dicetak di terminal dan ditulis ke dalam file.

- Menampilkan dan menyimpan data tiap siswa

```
53. for i := 0; i < jumlahSiswa; i++ {  
54.     baris := fmt.Sprintf("%-5d %-10s %-6.1f %-6.1f %-6.1f %-10.1f %-15s\n",  
55.         i+1, nama[i], tugas[i], uts[i], uas[i], nilaiAkhir[i], status[i])  
56.     fmt.Println(baris)  
57.     file.WriteString(baris)  
58. }
```

Program mencetak data lengkap setiap siswa ke layar dan menyimpannya ke file hasil_nilai.txt dalam format tabel.

- Struct* untuk menyimpan nama dan nilai akhir

```
60. type siswaData struct {  
61.     nama        string  
62.     nilaiAkhir float64  
63. }
```

Struct siswaData dibuat untuk menyimpan pasangan nama dan nilai akhir siswa, yang akan digunakan dalam proses perankingan.

- Menampilkan dan menyimpan peringkat siswa

```
74. fmt.Println("\n==== PERINGKAT SISWA ===")  
75. file.WriteString("\n==== PERINGKAT SISWA ===\n")  
76. for i, s := range ranking {  
77.     baris := fmt.Sprintf("Peringkat %d: %-10s (%.1f)\n", i+1, s.nama,  
    s.nilaiAkhir)  
78.     fmt.Println(baris)  
79.     file.WriteString(baris)  
80. }
```

Setelah urutan ditentukan, program mencetak dan menyimpan hasil peringkat ke layar dan file. Format yang digunakan jelas dan mudah dipahami.

Output :

```
PS C:\A\UM\VS CODE\6_Go\src> go run array.go
== INPUT DATA 3 SISWA ==

Siswa ke-1:
Nama : Putra
Nilai Tugas : 90
Nilai UTS : 85
Nilai UAS : 85

Siswa ke-2:
Nama : Putri
Nilai Tugas : 87
Nilai UTS : 85
Nilai UAS : 90

Siswa ke-3:
Nama : Salsa
Nilai Tugas : 69
Nilai UTS : 65
Nilai UAS : 67

== REKAP NILAI SISWA ==
No Nama Tgs UTS UAS Akhir Status
1 Putra 90.0 85.0 85.0 86.5 LULUS
2 Putri 87.0 85.0 90.0 87.6 LULUS
3 Salsa 69.0 65.0 67.0 67.0 TIDAK LULUS

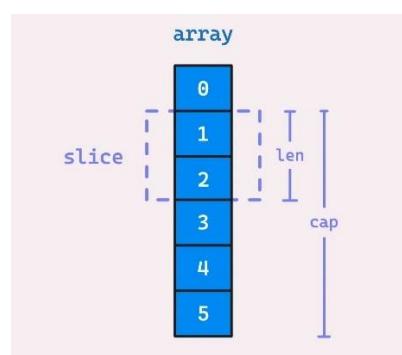
== PERINGKAT SISWA ==
Peringkat 1: Putri (87.6)
Peringkat 2: Putra (86.5)
Peringkat 3: Salsa (67.0)

 Data berhasil disimpan ke file 'hasil_nilai.txt'
```

Gambar 6.2 Contoh Pengimplementasian Array

B. SLICE

Dalam struktur data, slice tidak hanya dipahami sebagai potongan data dari array atau *list*, tetapi juga mencerminkan cara pengelolaan memori, efisiensi manipulasi data, dan kemampuan pewarisan data tanpa perlu melakukan duplikasi. Dalam beberapa bahasa pemrograman seperti Go dan Python, slice merupakan abstraksi penting yang memungkinkan pemrogram bekerja dengan bagian dari struktur data yang lebih besar secara efisien.



Gambar 6.3 Ilustrasi Slice pada Array

Sumber: VictoriaMetrics

Kelebihan :

- Efisiensi memori: Slice memungkinkan akses sebagian data tanpa menyalin keseluruhan struktur.
- Waktu akses cepat: Karena slice mempertahankan indeks asli array, akses elemen tetap cepat ($O(1)$).
- Fleksibilitas: Slice bisa digunakan untuk manipulasi data secara dinamis tanpa mendefinisikan struktur baru.

Kekurangan :

- *Side effects*: Di beberapa bahasa (seperti Go), perubahan pada slice memengaruhi *array* asli.
- Manajemen kapasitas kompleks: Jika tidak hati-hati, operasi *append* bisa menyebabkan *overhead* alokasi ulang.
- Slice mudah menimbulkan bug jika indeks tidak dikontrol dengan baik.

Contoh Program yang Mengimplementasikan Slice:

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type Buah struct {
8.     Nama string
9.     Warna string
10.    Berat float64
11. }
12.
13. func tampilanBuah(buahSlice []Buah) {
14.     fmt.Println("Daftar Buah:")
15.     for i, buah := range buahSlice {
16.         fmt.Printf("%d. %s (Warna: %s, Berat: %.2f g)\n", i+1, buah>Nama,
17.                     buah.Warna, buah.Berat)
18.     }
19.     fmt.Printf("Panjang Slice : %d\n", len(buahSlice))
20.     fmt.Printf("Kapasitas Slice: %d\n\n", cap(buahSlice))
21. }
22. func hapusBuah(slice []Buah, nama string) []Buah {
23.     for i, b := range slice {
24.         if b>Nama == nama {
25.             return append(slice[:i], slice[i+1:]...)
26.         }
27.     }
28.     fmt.Println("Buah tidak ditemukan:", nama)
29.     return slice
30. }
31.
32. func main() {
```

```

33.     buah := []Buah{
34.         {"Apel", "Merah", 150.0},
35.         {"Jeruk", "Oranye", 200.5},
36.         {"Mangga", "Kuning", 300.0},
37.     }
38.
39.     tampilkanBuah(buah)
40.     buahBaru := Buah{"Pisang", "Kuning", 120.0}
41.     buah = append(buah, buahBaru)
42.     fmt.Println("Setelah Menambahkan Pisang:")
43.     tampilkanBuah(buah)
44.     buah = hapusBuah(buah, "Jeruk")
45.     fmt.Println("Setelah Menghapus Jeruk:")
46.     tampilkanBuah(buah)
47. }
```

Penjelasan kode program:

```

13. func tampilkanBuah(buahSlice []Buah) {
14.     fmt.Println("Daftar Buah:")
15.     for i, buah := range buahSlice {
16.         fmt.Printf("%d. %s (Warna: %s, Berat: %.2f g)\n", i+1, buah>Nama,
17.                     buah.Warna, buah.Berat)
18.     }
19.     fmt.Printf("Panjang Slice : %d\n", len(buahSlice))
20.     fmt.Printf("Kapasitas Slice: %d\n\n", cap(buahSlice))
```

Struct ini mewakili setiap buah dengan nama, warna, dan berat. Membuat slice berisi beberapa buah yang langsung diisi nilai struct. Selain itu juga, menampilkan isi slice dengan format yang rapi. juga mencetak panjang (len) dan kapasitas (cap) dari slice.

```

22. func hapusBuah(slice []Buah, nama string) []Buah {
23.     for i, b := range slice {
24.         if b>Nama == nama {
25.             return append(slice[:i], slice[i+1:]...)
26.         }
27.     }
```

Mencari buah berdasarkan nama dan menghapusnya dari slice menggunakan `append(slice[:i], slice[i+1:]...)`. `append()` digunakan untuk menambahkan elemen ke slice. Slice bersifat dinamis, jadi panjang dan kapasitas bisa berubah.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run slice.go
Daftar Buah:
1. Apel (Warna: Merah, Berat: 150.00 g)
2. Jeruk (Warna: Oranye, Berat: 200.50 g)
3. Mangga (Warna: Kuning, Berat: 300.00 g)
Panjang Slice : 3
Kapasitas Slice: 3

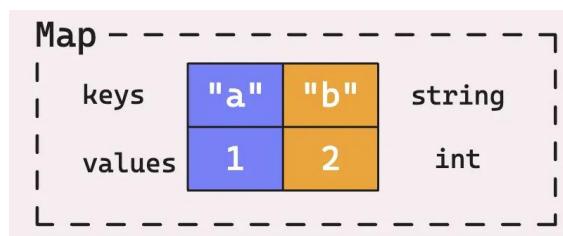
Setelah Menambahkan Pisang:
Daftar Buah:
1. Apel (Warna: Merah, Berat: 150.00 g)
2. Jeruk (Warna: Oranye, Berat: 200.50 g)
3. Mangga (Warna: Kuning, Berat: 300.00 g)
4. Pisang (Warna: Kuning, Berat: 120.00 g)
Panjang Slice : 4
Kapasitas Slice: 6

Setelah Menghapus Jeruk:
Daftar Buah:
1. Apel (Warna: Merah, Berat: 150.00 g)
2. Mangga (Warna: Kuning, Berat: 300.00 g)
3. Pisang (Warna: Kuning, Berat: 120.00 g)
Panjang Slice : 3
Kapasitas Slice: 6
```

Gambar 6.4 Pengimplementasian Slice

C. MAP

Map merupakan tipe data asosiatif yang terdiri dari pasangan *key* dan *value* dimana kata kuncinya bersifat unik, tidak boleh sama. Setiap data atau nilai yang disimpan dalam slice memiliki *key* yang berfungsi sebagai penanda atau pengidentifikasian untuk mengakses nilai yang terkait. Dalam map, indeks yang digunakan untuk mengakses nilai dapat ditentukan sendiri dengan tipe data tertentu dan indeks tersebut disebut sebagai *key*. Berbeda dengan Array dan Slice, jumlah data yang dimasukkan ke dalam map boleh sebanyak-banyaknya, asalkan kata kuncinya berbeda, jika kata kunci yang digunakan sama, maka secara otomatis data sebelumnya akan diganti dengan data yang baru.



Gambar 6.5 Map
Sumber: VictoriaMetrics

Kelebihan:

- Akses cepat karena menggunakan struktur hash dibalik layar.
- *Tipe key* dan *value* dapat ditentukan sesuai kebutuhan.
- Dapat menambah jumlah elemen secara dinamis.
- *built-in Fungsiality*.

Kekurangan:

- Map tidak menyimpan data terurut yang membuat urutan iterasi bisa berubah-ubah.
- Tidak bisa menggunakan Slice, Map dan fungsi sebagai *key*.
- Tidak aman untuk akses pararel.

Contoh Program yang Mengimplementasikan Map:

```
1. package main
2.
3. import "fmt"
4.
5. type Mahasiswa struct {
6.     Nama string
7.     NIM  string
8. }
9.
10. func tampilanNilai(data map[Mahasiswa]map[string]float64) {
11.     for mhs, nilaiMap := range data {
12.         fmt.Printf("Nama: %s | NIM: %s\n", mhs>Nama, mhs.NIM)
13.         for matkul, nilai := range nilaiMap {
14.             fmt.Printf(" - %s: %.2f\n", matkul, nilai)
15.         }
16.         fmt.Println()
17.     }
18. }
19.
20. func main() {
21.     nilaiMahasiswa := make(map[Mahasiswa]map[string]float64)
22.
23.     mhs1 := Mahasiswa{"Putra", "240535"}
24.     mhs2 := Mahasiswa{"Nadiyah", "240636"}
25.
26.     nilaiMahasiswa[mhs1] = map[string]float64{
27.         "Matematika": 87.5,
28.         "Fisika":      78.0,
29.     }
30.
31.     nilaiMahasiswa[mhs2] = map[string]float64{
32.         "Matematika": 92.0,
33.         "Bahasa":     88.5,
34.     }
35.
36.     tampilanNilai(nilaiMahasiswa)
37. }
```

Penjelasan Kode Tiap Bagian

- Membuat struct mahasiswa

```
5. type Mahasiswa struct {
6.     Nama string
7.     NIM  string
8. }
```

Kode ini mendefinisikan sebuah *struct* bernama `Mahasiswa` yang memiliki dua *field*, yaitu `Nama` dan `NIM` dengan tipe data `string`. Kemudian *struct* ini digunakan untuk merepresentasikan data mahasiswa secara terstruktur.

- Membuat fungsi untuk menampilkan nilai-nilai mahasiswa

```
9. func tampilanNilai(data map[Mahasiswa]map[string]float64) {
10.    for mhs, nilaiMap := range data {
11.        fmt.Printf("Nama: %s | NIM: %s\n", mhs>Nama, mhs.NIM)
12.        for matkul, nilai := range nilaiMap {
13.            fmt.Printf(" - %s: %.2f\n", matkul, nilai)
14.        }
15.        fmt.Println()
16.    }
17. }
```

Pada kode diatas fungsi menerima parameter berupa `tampilanNilai` dengan *key* bertipe `Mahasiswa` dan *value* yakni `map[string]float64`. Fungsi ini melakukan iterasi terhadap setiap mahasiswa, lalu menampilkan nama, nim serta daftar nilai mereka untuk tiap matakuliah. `for mhs, nilaiMap := range data` digunakan untuk mengambil setiap pasang *value* dari map `data`, di mana *key*-nya adalah *struct* `Mahasiswa` dan *value*-nya adalah map dari matakuliah ke nilai. Kemudian *for* didalam *for* yakni `for matkul, nilai := range nilaiMap` digunakan untuk membaca semua matakuliah beserta nilai yang dimiliki oleh mahasiswa tersebut.

- Fungsi main

```
20. func main() {
21.     nilaiMahasiswa := make(map[Mahasiswa]map[string]float64)
22.
23.     mhs1 := Mahasiswa{"Putra", "240535"}
24.     mhs2 := Mahasiswa{"Nadiyah", "240636"}
25.
26.     nilaiMahasiswa[mhs1] = map[string]float64{
27.         "Matematika": 87.5,
28.         "Fisika":      78.0,
29.     }
30.
31.     nilaiMahasiswa[mhs2] = map[string]float64{
32.         "Matematika": 92.0,
33.         "Bahasa":     88.5,
34.     }
35.
36.     tampilanNilai(nilaiMahasiswa)
37. }
```

Pada fungsi utama diatas dibuat sebuah map bernama `nilaiMahasiswa` yang memiliki *key* bertipe struct `Mahasiswa` dan *value* berupa map dari *string* ke *float64*, yang menyimpan nama mata kuliah dan nilai. Dua data mahasiswa, yaitu "Putra" dengan NIM "240535" dan "Nadiyah" dengan NIM "240636", kemudian dibuat dalam bentuk *struct*. Masing-masing mahasiswa dimasukkan ke dalam map `nilaiMahasiswa` bersama dengan data mata kuliah dan nilai yang bersangkutan. Setelah semua data disiapkan, fungsi `tampilkanNilai` dipanggil untuk menampilkan informasi tersebut ke layar. Fungsi ini menggunakan dua buah perulangan *for* yang pertama untuk mengambil setiap mahasiswa dan map nilainya, dan yang kedua untuk menampilkan daftar mata kuliah beserta nilai masing-masing mahasiswa dengan format yang rapi.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run map.go
Nama: Putra | NIM: 240535
- Matematika: 87.50
- Fisika: 78.00

Nama: Nadiyah | NIM: 240636
- Matematika: 92.00
- Bahasa: 88.50
```

Gambar 6.6 Pengimplementasian Map

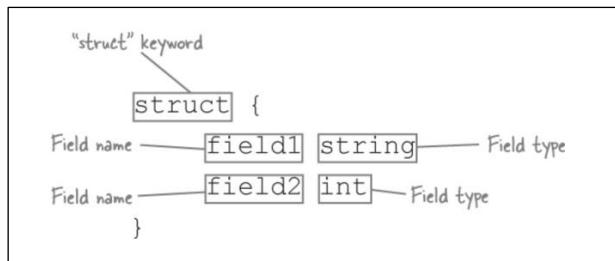
D. STRUCT

Pengenalan Struct

Dalam bahasa pemrograman Go tidak mengadopsi konsep *class* seperti pada bahasa pemrograman berparadigma *Object-Oriented Programming* (OOP) lainnya, seperti *Java* atau *C++*. Namun, Go tetap menyediakan kemampuan untuk membungkus data dan perilaku ke dalam satu entitas melalui fitur yang disebut *struct* (*structure*). *Struct* adalah tipe data bentukan (*custom data type*) yang digunakan untuk mengelompokkan beberapa variabel yang bisa berbeda-beda tipe datanya, ke dalam satu kesatuan logis dengan nama tertentu.

Deklarasi Struct

Struct dideklarasikan menggunakan kata kunci *type*, diikuti oleh nama *struct* yang ingin dibuat, lalu dituliskan kata *struct* dan diapit oleh kurung kurawal `{ }` yang membuat daftar properti atau atribut *struct* tersebut. Setiap atribut di dalam *struct* dituliskan dalam format `namaAtribut tipeData`. Go memungkinkan setiap atribut memiliki tipe data yang berbeda, sesuai dengan kebutuhan representasi datang diajukan.



Gambar 6.7 Deklarasi Struct

Sumber: DonOfDen

Berikut merupakan contoh deklarasi struct:

```

1. type Mahasiswa struct {
2.   Nama string
3.   NIM  string
4.   Usia int
5.   IPK  float32
6. }
  
```

Inisialisasi Struct (Dengan dan Tanpa new)

a. Inisialisasi Tanpa new

Cara ini adalah yang paling umum digunakan, yaitu dengan membuat variabel dari *struct* secara langsung:

```

1. type Mahasiswa struct {
2.   Nama string
3.   NIM  string
4. }
5.
6. var mhs1 Mahasiswa
7. mhs1>Nama = "Putra"
8. mhs1.NIM = "TIC2024001"
  
```

Dalam contoh di atas, variabel `mhs1` bertipe `Mahasiswa` akan dialokasikan di *stack*, dan nilai dari masing-masing atribut (*field*) dapat diberikan secara manual.

b. Inisialisasi dengan new

Alternatif lain adalah menggunakan fungsi *new*. Fungsi ini akan mengalokasikan memori untuk *struct* dan mengembalikan *pointer* ke *struct* yang baru dibuat.

Contoh Program:

```

1. mhs2 := new(Mahasiswa)
2. mhs2>Nama = "Zahra"
3. mhs2.NIM = "TIC2024002"
  
```

Pada contoh tersebut, `mhs2` bertipe `*Mahasiswa`, yaitu *pointer* ke *struct* `Mahasiswa`. Bahasa Go secara otomatis menangani *dereference pointer* ketika mengakses *field*, sehingga tidak diperlukan tanda asterisk (*) secara eksplisit untuk mengakses atribut.

Inisialisasi dengan Struct Literal

Struct literal merupakan cara singkat dan eksplisit untuk menginisialisasi objek struct beserta nilai awal dari *field*-nya. Pendekatan ini lebih ringkas dan sangat direkomendasikan karena meningkatkan keterbacaan serta mengurangi potensi kesalahan.

Contoh Program:

```
1. mhs3 := Mahasiswa{  
2.     Nama: "Nabila",  
3.     NIM:  "TIC2024003",  
4. }
```

Pada contoh tersebut, nilai diberikan langsung ke masing-masing *field* menggunakan sintaks (*field*: nilai). Pendekatan ini bersifat deklaratif dan memperjelas struktur data yang dibentuk.

Contoh Kode Program yang Mengimplementasikan Struct:

```
1. package main  
2.  
3. import (  
4.     "fmt"  
5. )  
6.  
7. type Mahasiswa struct {  
8.     Nama      string  
9.     NIM       string  
10.    Jurusan    string  
11.    NilaiUAS float64  
12.    NilaiUTS float64  
13. }  
14.  
15. func (m Mahasiswa) HitungRataRata() float64 {  
16.     return (m.NilaiUAS + m.NilaiUTS) / 2  
17. }  
18.  
19. func (m *Mahasiswa) UbahNama(namaBaru string) {  
20.     m>Nama = namaBaru  
21. }  
22.  
23. func CetakProfil(m Mahasiswa) {  
24.     fmt.Println("== Data Mahasiswa ==")  
25.     fmt.Println("Nama : ", m>Nama)  
26.     fmt.Println("NIM : ", m>NIM)  
27.     fmt.Println("Jurusan:", m>Jurusan)  
28. }  
29.  
30. func BuatMahasiswa(nama, nim, jurusan string, uts, uas float64) Mahasiswa {  
31.     return Mahasiswa{  
32.         Nama:      nama,  
33.         NIM:       nim,  
34.         Jurusan:   jurusan,  
35.         NilaiUTS:  uts,  
36.         NilaiUAS: uas,  
37.     }  
38. }  
39.
```

```

40. func main() {
41.     mhs1 := BuatMahasiswa("Aril Salwa Hartono", "TI2025005", "Teknik
        Informatika", 85, 90)
42.
43.     CetakProfil(mhs1)
44.     fmt.Printf("Rata-rata nilai: %.2f\n", mhs1.HitungRataRata())
45.
46.     mhs1.UbahNama("Aril Salwa H.")
47.
48.     fmt.Println("\nSetelah perubahan nama:")
49.     CetakProfil(mhs1)
50. }
```

Penjelasan Kode Tiap Bagian

- Definisi Struct Mahasiswa

```

7. type Mahasiswa struct {
8.     Nama      string
9.     NIM       string
10.    Jurusan   string
11.    NilaiUAS float64
12.    NilaiUTS float64
13. }
```

Struct `Mahasiswa` didefinisikan dengan beberapa *field* yang menyimpan data mahasiswa: nama, NIM, jurusan, serta nilai UTS dan UAS. Struct ini membentuk tipe data baru yang dapat digunakan untuk merepresentasikan objek mahasiswa.

- Fungsi `CetakProfil`

```

23. func CetakProfil(m c {
24.     fmt.Println("== Data Mahasiswa ==")
25.     fmt.Println("Nama : ", m>Nama)
26.     fmt.Println("NIM : ", m.NIM)
27.     fmt.Println("Jurusan:", m.Jurusan)
28. }
```

Fungsi ini mencetak data mahasiswa ke layar. Struct `Mahasiswa` dikirim sebagai parameter untuk ditampilkan, tetapi tidak dimodifikasi karena dikirim sebagai nilai biasa (*by value*).

- Fungsi `BuatMahasiswa`

```

30. func BuatMahasiswa(nama, nim, jurusan string, uts, uas float64) Mahasiswa {
31.     return Mahasiswa{
32.         Nama:      nama,
33.         NIM:       nim,
34.         Jurusan:   jurusan,
35.         NilaiUTS:  uts,
36.         NilaiUAS:  uas,
37.     }
38. }
```

Fungsi ini membuat dan mengembalikan objek `Mahasiswa` berdasarkan parameter yang diberikan. Ini memudahkan proses inisialisasi objek struct dengan data lengkap.

- Fungsi main

```

40. func main() {
41.     mhs1 := BuatMahasiswa("Aril Salwa Hartono", "TI2025005", "Teknik
        Informatika", 85, 90)
42.
43.     CetakProfil(mhs1)
44.     fmt.Printf("Rata-rata nilai: %.2f\n", mhs1.HitungRataRata())
45.
46.     mhs1.UbahNama("Aril Salwa H.")
47.
48.     fmt.Println("\nSetelah perubahan nama:")
49.     CetakProfil(mhs1)
50. }
```

Output:

```

PS C:\A\UM\VS CODE\6_Go\src> go run struct.go
== Data Mahasiswa ==
Nama : Aril Salwa Hartono
NIM : TI2025005
Jurusan: Teknik Informatika
Rata-rata nilai: 87.50

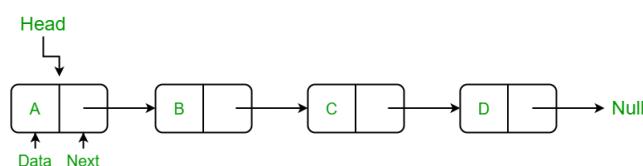
Setelah perubahan nama:
== Data Mahasiswa ==
Nama : Aril Salwa H.
NIM : TI2025005
Jurusan: Teknik Informatika
```

Gambar 6.8 Pengimplementasian Struct

E. LINKED LIST

Linked List adalah struktur data dinamis yang terdiri dari kumpulan *node* yang saling terhubung melalui referensi atau *pointer*. Setiap *node* dalam *Linked List* terdiri dari dua bagian utama: *data*, yang menyimpan nilai atau informasi, dan *pointer*, yang menunjuk ke *node* berikutnya dalam daftar. *Linked List* menyimpan elemen secara terpisah, di mana setiap *node* hanya mengetahui lokasi *node* berikutnya melalui referensi.

Linked List memungkinkan penyisipan dan penghapusan elemen dengan lebih fleksibel tanpa harus menggeser elemen lain. Hal ini membuatnya ideal untuk digunakan dalam sistem yang membutuhkan manipulasi data secara dinamis, seperti antrian, di mana elemen dapat ditambahkan dan dihapus dengan efisiensi tinggi tanpa memengaruhi seluruh struktur data.



Gambar 6.9 Linked List

Sumber: GeeksforGeeks

Kelebihan:

- Fleksibilitas dalam perubahan ukuran
- Efisien dalam penyisipan dan penghapusan
- Menghemat memori jika data dinamis

Kekurangan:

- Menggunakan lebih banyak memori
- Akses lebih lambat dibanding *array*
- Pengelolaan *pointer* lebih kompleks

Contoh Program yang Mengimplementasikan Linked List:

```
1. package main
2.
3. import "fmt"
4.
5. type Node struct {
6.     data int
7.     next *Node
8. }
9.
10. type LinkedList struct {
11.     head *Node
12. }
13.
14. func (list *LinkedList) Insert(value int) {
15.     newNode := &Node{data: value}
16.     if list.head == nil {
17.         list.head = newNode
18.     } else {
19.         temp := list.head
20.         for temp.next != nil {
21.             temp = temp.next
22.         }
23.         temp.next = newNode
24.     }
25. }
26.
27. func (list *LinkedList) Delete(value int) {
28.     if list.head == nil {
29.         fmt.Println("Linked List kosong")
30.         return
31.     }
32.
33.     if list.head.data == value {
34.         list.head = list.head.next
35.         return
36.     }
37.
38.     temp := list.head
39.     for temp.next != nil && temp.next.data != value {
40.         temp = temp.next
41.     }
42.
43.     if temp.next == nil {
44.         return
45.     }
46.
47.     if temp.next.data == value {
48.         temp.next = temp.next.next
49.     }
50.
51.     return
52. }
```

```

41.    }
42.
43.    if temp.next == nil {
44.        fmt.Println("Data tidak ditemukan dalam Linked List")
45.    } else {
46.        temp.next = temp.next.next
47.    }
48. }
49.
50. func (list *LinkedList) Display() {
51.     if list.head == nil {
52.         fmt.Println("Linked List kosong")
53.         return
54.     }
55.
56.     temp := list.head
57.     for temp != nil {
58.         fmt.Print(temp.data, " -> ")
59.         temp = temp.next
60.     }
61.     fmt.Println("nil")
62. }
63.
64. func main() {
65.     list := LinkedList{}
66.
67.     list.Insert(10)
68.     list.Insert(20)
69.     list.Insert(30)
70.
71.     fmt.Println("Isi Linked List sekarang:")
72.     list.Display()
73.
74.     list.Delete(20)
75.     fmt.Println("Isi Linked List setelah delete:")
76.     list.Display()
77. }

```

Penjelasan Kode Tiap Bagian

- Membuat Linked List

```

5. type Node struct {
6.     data int
7.     next *Node
8. }
9.
10. type LinkedList struct {
11.     head *Node
12. }

```

Kode diawali dengan pembuatan struktur `Node` yang terdiri dari dua bagian, yaitu `data` untuk menyimpan nilai dan `next`, yang merupakan *pointer* ke *node* berikutnya. Kemudian, ada struktur `LinkedList` yang memiliki variabel `head` sebagai titik awal dari daftar. Jika `head` bernilai `nil`, itu berarti `LinkedList` masih kosong.

- Membuat Fungsi untuk menambah elemen

```
14. func (list *LinkedList) Insert(value int) {  
15.     newNode := &Node{data: value}  
16.     if list.head == nil {  
17.         list.head = newNode  
18.     } else {  
19.         temp := list.head  
20.         for temp.next != nil {  
21.             temp = temp.next  
22.         }  
23.         temp.next = newNode  
24.     }  
25. }
```

Proses menambahkan elemen dilakukan dalam fungsi ini, di mana pertama-tama dibuat `Node` baru dengan nilai yang diberikan. Jika *Linked List* masih kosong, `Node` baru langsung dijadikan `head`, tetapi jika sudah ada elemen lain dalam daftar, program akan mencari *node* terakhir dan menambahkan `Node` baru setelahnya.

- Membuat fungsi untuk menghapus elemen

```
27. func (list *LinkedList) Delete(value int) {  
28.     if list.head == nil {  
29.         fmt.Println("Linked List kosong")  
30.         return  
31.     }  
32.  
33.     if list.head.data == value {  
34.         list.head = list.head.next  
35.         return  
36.     }  
37.  
38.     temp := list.head  
39.     for temp.next != nil && temp.next.data != value {  
40.         temp = temp.next  
41.     }  
42.  
43.     if temp.next == nil {  
44.         fmt.Println("Data tidak ditemukan dalam Linked List")  
45.     } else {  
46.         temp.next = temp.next.next  
47.     }
```

Untuk menghapus elemen, digunakan fungsi di atas ini. Jika `head` memiliki nilai yang ingin dihapus, maka `head` cukup dipindahkan ke `Node` harus menelusuri daftar untuk menemukan *node* sebelumnya, lalu mengubah *pointer next* agar menghubungkan *node* sebelum dan sesudah *node* yang dihapus. Jika tidak ditemukan, program akan mencetak pesan bahwa elemen tersebut tidak ada dalam daftar.

- Fungsi untuk menampilkan isi Linked List

```
50. func (list *LinkedList) Display() {  
51.     if list.head == nil {  
52.         fmt.Println("Linked List kosong")
```

```

53.     return
54. }
55.
56. temp := list.head
57. for temp != nil {
58.     fmt.Println(temp.data, " -> ")
59.     temp = temp.next
60. }
61. fmt.Println("nil")
62. }
```

Fungsi `Display()` ini digunakan untuk menampilkan isi *Linked List* dengan cara menelusuri setiap elemen dari `head` hingga `nil`. Saat setiap elemen ditemukan, program mencetaknya dalam format data → data → nil, menunjukkan bagaimana setiap *node* saling terhubung.

- Fungsi main

```

64. func main() {
65.     list := LinkedList{}
66.
67.     list.Insert(10)
68.     list.Insert(20)
69.     list.Insert(30)
70.
71.     fmt.Println("Isi Linked List sekarang:")
72.     list.Display()
73.
74.     list.Delete(20)
75.     fmt.Println("Isi Linked List setelah delete:")
76.     list.Display()
77. }
```

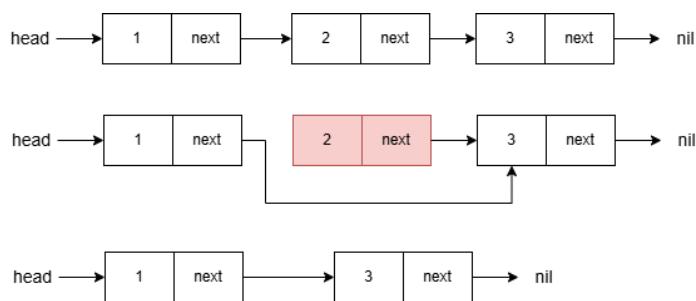
Output:

```

PS C:\A\UM\VS CODE\6_Go\src> go run linkedList.go
Isi Linked List sekarang:
10 -> 20 -> 30 -> nil
Isi Linked List setelah delete:
10 -> 30 -> nil
```

Gambar 6.10 Pengimplementasian *Linked List*

Berikut ilustrasi dari *Output* di atas:



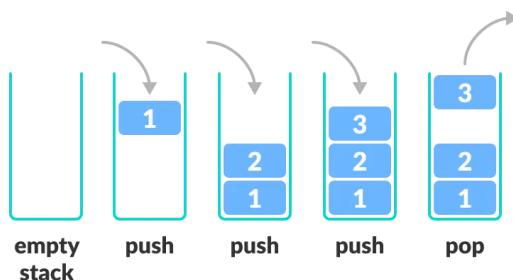
Gambar 6.11 Ilustrasi Output Program *Linked List*

BAB 7

STRUKTUR DATA LANJUTAN

A. STACK

Stack adalah struktur data linear yang fundamental dalam ilmu komputer, yang menyimpan dan mengelola data dengan prinsip *LIFO* (*Last In, First Out*), yaitu elemen yang terakhir dimasukkan akan menjadi yang pertama dikeluarkan. *Stack* hanya memungkinkan operasi penambahan (*push*) dan penghapusan (*pop*) dilakukan pada satu sisi yang disebut *top*. Struktur ini sering dianalogikan sebagai tumpukan barang, seperti piring atau buku, di mana pengambilan dan penambahan hanya dapat dilakukan dari atas. *Stack* banyak digunakan dalam berbagai proses komputasi penting seperti pengelolaan pemanggilan fungsi (*call stack*), algoritma rekursif, *undo-redo* dalam aplikasi, serta evaluasi ekspresi matematika. Karena kesederhanaannya, *stack* menjadi salah satu komponen dasar dalam pemahaman dan implementasi algoritma serta struktur data lanjutan.



Gambar 7.1 Ilustrasi Stack dengan Prinsip LIFO

Sumber: Programiz

Kelebihan:

- Struktur sederhana dan mudah diimplementasikan.
- Efisien untuk proses pengolahan data dengan pola *LIFO*.
- Mendukung operasi rekursi dan manajemen fungsi dalam pemrograman tingkat rendah.

Kekurangan:

- Akses terbatas, hanya dari bagian atas (*top*).
- Tidak cocok untuk pencarian secara acak atau memanipulasi elemen di tengah.
- Kapasitas ukuran *stack* terbatas dan rentan *overflow* jika tidak ditangani dengan benar.

Contoh Program yang Mengimplementasikan Stack:

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type Stack struct {
8.     items []int
9. }
```

```

10.
11. func NewStack() *Stack {
12.     return &Stack{items: []int{}}
13. }
14.
15. func (s *Stack) Push(item int) {
16.     s.items = append(s.items, item)
17.     fmt.Printf("Elemen %d ditambahkan ke stack\n", item)
18. }
19.
20. func (s *Stack) Pop() int {
21.     if len(s.items) == 0 {
22.         fmt.Println("Stack kosong!")
23.         return -1
24.     }
25.     top := s.items[len(s.items)-1]
26.     s.items = s.items[:len(s.items)-1]
27.     fmt.Printf("Elemen %d dihapus dari stack\n", top)
28.     return top
29. }
30. func (s *Stack) Display() {
31.     if len(s.items) == 0 {
32.         fmt.Println("Stack kosong!")
33.         return
34.     }
35.     fmt.Println("Isi stack (dari bawah ke atas):")
36.     for i, v := range s.items {
37.         fmt.Printf("[%d] %d\n", i, v)
38.     }
39. }
40. func main() {
41.     stack := NewStack()
42.
43.     stack.Push(10)
44.     stack.Push(20)
45.     stack.Push(30)
46.
47.
48.     stack.Display()
49.
50.     stack.Pop()
51.     stack.Display()
52. }
```

Penjelasan Kode Tiap Bagian

- Membuat Struktur Stack

```

7. type Stack struct {
8.     items []int
9. }
```

Kode ini mendefinisikan struktur `Stack` yang memiliki satu *field* bernama `items`, yaitu slice dari `int ([]int)`. Slice ini digunakan untuk menyimpan elemen-elemen di dalam `stack` secara dinamis.

- Fungsi untuk membuat Stack baru

```
11. func NewStack() *Stack {
12.     return &Stack{items: []int{}}
13. }
```

Fungsi `NewStack()` digunakan untuk membuat *stack* baru yang masih kosong. Fungsi ini mengembalikan *pointer* ke objek `Stack` dengan slice `items` yang diinisialisasi kosong.

- Menambah Elemen ke Stack (Push)

```
15. func (s *Stack) Push(item int) {
16.     s.items = append(s.items, item)
17.     fmt.Printf("Elemen %d ditambahkan ke stack\n", item)
18. }
```

Fungsi `Push` bertugas untuk menambahkan elemen ke atas *stack*. Data baru akan disisipkan di akhir slice `items` menggunakan fungsi `append`, yang sesuai dengan prinsip *LIFO*. Setelah ditambahkan, fungsi akan mencetak informasi bahwa elemen berhasil dimasukkan.

- Menghapus Elemen dari Stack (Pop)

```
20. func (s *Stack) Pop() int {
21.     if len(s.items) == 0 {
22.         fmt.Println("Stack kosong!")
23.         return -1
24.     }
25.     top := s.items[len(s.items)-1]
26.     s.items = s.items[:len(s.items)-1]
27.     fmt.Printf("Elemen %d dihapus dari stack\n", top)
28.     return top
29. }
```

Fungsi `Pop` menghapus elemen paling atas dari *stack*. Pertama, dicek apakah *stack* kosong (`if len(s.items) == 0 {}`), jika iya, maka akan menampilkan pesan peringatan dan mengembalikan nilai `-1` sebagai indikator kesalahan. Jika tidak kosong, elemen terakhir dari slice (paling atas di *stack*) diambil dan disimpan dalam variabel `top`, lalu slice dipotong satu elemen dari belakang menggunakan *slicing* (`s.items[:len(s.items)-1]`). Terakhir, elemen tersebut dikembalikan dan ditampilkan ke layar.

- Menampilkan Isi Stack (Display)

```
30. func (s *Stack) Display() {
31.     if len(s.items) == 0 {
32.         fmt.Println("Stack kosong!")
33.         return
34.     }
35.     fmt.Println("Isi stack (dari bawah ke atas):")
36.     for i, v := range s.items {
37.         fmt.Printf("[%d] %d\n", i, v)
38.     }
39. }
```

Fungsi `Display` menampilkan semua elemen yang ada di `stack`. Jika `stack` kosong, akan muncul pesan peringatan. Jika ada isi, maka akan dicetak ke layar dari elemen paling bawah hingga ke atas, lengkap dengan indeksnya.

- Fungsi main

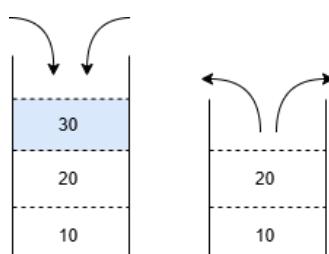
```
40. func main() {
41.     stack := NewStack()
42.
43.     stack.Push(10)
44.     stack.Push(20)
45.     stack.Push(30)
46.
47.     stack.Display()
48.
49.     stack.Pop()
50.     stack.Display()
51. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run stack.go
Elemen 10 ditambahkan ke stack
Elemen 20 ditambahkan ke stack
Elemen 30 ditambahkan ke stack
Isi stack (dari bawah ke atas):
[0] 10
[1] 20
[2] 30
Elemen 30 dihapus dari stack
Isi stack (dari bawah ke atas):
[0] 10
[1] 20
```

Gambar 7.2 Pengimplementasian Stack

Berikut ilustrasi dari *Output* di atas:



Gambar 7.3 Ilustrasi Output Program Stack

B. QUEUE

Queue (antrian) adalah salah satu struktur data linear dalam ilmu komputer yang berfungsi untuk menyimpan dan mengelola kumpulan data secara berurutan berdasarkan prinsip *FIFO* (*First In, First Out*), yaitu elemen yang pertama kali dimasukkan akan menjadi elemen

pertama yang dikeluarkan. *Queue* sangat berguna dalam berbagai aplikasi seperti sistem antrian pelanggan, manajemen proses dalam sistem operasi, pengiriman data dalam jaringan, hingga simulasi antrian dalam dunia bisnis. Elemen dalam *queue* hanya dapat ditambahkan dari satu ujung (*belakang/tail*) melalui operasi *enqueue*, dan dihapus dari ujung lainnya (*depan/head*) melalui operasi *dequeue*. Dengan cara ini, *queue* menjamin pengolahan data yang adil dan teratur sesuai urutan kedatangan.



Gambar 7.4 Ilustrasi Queue dengan Prinsip FIFO

Sumber: Programiz

Kelebihan:

- Struktur sederhana dan mudah diimplementasikan.
- Efisien dalam pengolahan data secara berurutan.
- Mendukung operasi *asynchronous* esperto penjadwalan.

Kekurangan:

- Tidak fleksibel dalam penyisipan dan penghapusan selain di ujung depan dan belakang.
- Dapat mengalami keterbatasan ukuran.
- Tidak cocok untuk pencarian elemen secara acak karena harus diakses secara linear.

Contoh Program yang Mengimplementasikan Queue:

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type Queue struct {
8.     items []int
9. }
10.
11. func NewQueue() *Queue {
12.     return &Queue{items: []int{}}
13. }
14.
15. func (q *Queue) Enqueue(item int) {
16.     q.items = append(q.items, item)
17.     fmt.Printf("Elemen %d ditambahkan ke queue\n", item)
18. }
19.
20. func (q *Queue) Dequeue() int {
21.     if len(q.items) == 0 {
22.         fmt.Println("Queue kosong!")
23.     }
  
```

```

24.     }
25.     front := q.items[0]
26.     q.items = q.items[1:]
27.     fmt.Printf("Elemen %d dihapus dari queue\n", front)
28.     return front
29. }
30.
31. func (q *Queue) Display() {
32.     if len(q.items) == 0 {
33.         fmt.Println("Queue kosong!")
34.         return
35.     }
36.     fmt.Println("Isi queue (dari depan ke belakang):")
37.     for i, v := range q.items {
38.         fmt.Printf("[%d] %d\n", i, v)
39.     }
40. }
41.
42. func main() {
43.     queue := NewQueue()
44.
45.     queue.Enqueue(10)
46.     queue.Enqueue(20)
47.     queue.Enqueue(30)
48.
49.     queue.Display()
50.
51.     queue.Dequeue()
52.
53.     queue.Display()
54. }

```

Penjelasan Kode Tiap Bagian

- Membuat Struktur Queue

```

7. type Queue struct {
8.     items []int
9. }

```

Kode ini mendefinisikan struktur data bernama `Queue` yang memiliki satu properti, yaitu `items`, sebuah *slice* bertipe `integer ([]int)`. *Slice* ini digunakan untuk menyimpan elemen-elemen dalam antrian.

- Membuat Queue Baru

```

11. func NewQueue() *Queue {
12.     return &Queue{items: []int{}}
13. }

```

Fungsi `NewQueue()` digunakan untuk membuat `queue` baru yang kosong. Ia mengembalikan *pointer* ke objek `Queue` dengan *slice* `items` yang diinisialisasi kosong. `Queue` ini akan digunakan sebagai tempat menyimpan data yang masuk ke antrian.

- Menambah Elemen ke Queue (Enqueue)

```
15. func (q *Queue) Enqueue(item int) {  
16.     q.items = append(q.items, item)  
17.     fmt.Printf("Elemen %d ditambahkan ke queue\n", item)  
18. }
```

Fungsi `Enqueue()` bertugas untuk menambahkan elemen baru ke akhir antrian. Elemen ditambahkan menggunakan fungsi bawaan `append`, yang menyisipkan nilai `item` ke ujung slice `items`. Proses ini sesuai dengan perilaku `queue` di mana penambahan elemen (`enqueue`) selalu terjadi di belakang antrian. Pesan juga dicetak ke layar untuk menunjukkan elemen yang berhasil ditambahkan.

- Menghapus Elemen dari Queue (Dequeue)

```
20. func (q *Queue) Dequeue() int {  
21.     if len(q.items) == 0 {  
22.         fmt.Println("Queue kosong!")  
23.         return -1  
24.     }  
25.     front := q.items[0]  
26.     q.items = q.items[1:]  
27.     fmt.Printf("Elemen %d dihapus dari queue\n", front)  
28.     return front  
29. }
```

Fungsi `Dequeue()` digunakan untuk menghapus dan mengambil elemen paling depan dari antrian. Jika `queue` kosong (dicek dengan `len(q.items) == 0`), maka fungsi mencetak pesan kesalahan dan mengembalikan nilai `-1`. Jika tidak kosong, elemen pertama (paling depan) disimpan ke variabel `front`, lalu `slice` diubah dengan menghapus elemen tersebut menggunakan `slicing` (`q.items = q.items[1:]`), dan elemen yang dihapus dikembalikan.

- Menampilkan Isi Queue

```
31. func (q *Queue) Display() {  
32.     if len(q.items) == 0 {  
33.         fmt.Println("Queue kosong!")  
34.         return  
35.     }  
36.     fmt.Println("Isi queue (dari depan ke belakang):")  
37.     for i, v := range q.items {  
38.         fmt.Printf("[%d] %d\n", i, v)  
39.     }  
40. }
```

Fungsi `Display()` digunakan untuk menampilkan semua isi antrian dari depan ke belakang. Jika `queue` kosong, akan ditampilkan pesan “`Queue kosong!`”. Jika tidak, program mencetak setiap elemen `queue` lengkap dengan indeksnya menggunakan perulangan `for range`. Fungsi ini berguna untuk memantau status `queue` setelah operasi `enqueue` atau `dequeue` dilakukan.

- Fungsi main

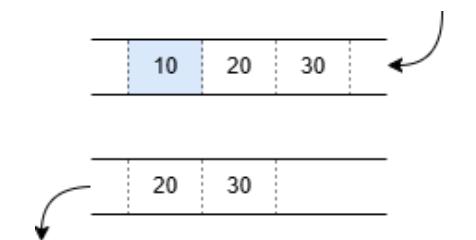
```
42. func main() {  
43.     queue := NewQueue()  
44.  
45.     queue.Enqueue(10)  
46.     queue.Enqueue(20)  
47.     queue.Enqueue(30)  
48.  
49.     queue.Display()  
50.  
51.     queue.Dequeue()  
52.  
53.     queue.Display()  
54. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run queue.go  
Elemen 10 ditambahkan ke queue  
Elemen 20 ditambahkan ke queue  
Elemen 30 ditambahkan ke queue  
Isi queue (dari depan ke belakang):  
[0] 10  
[1] 20  
[2] 30  
Elemen 10 dihapus dari queue  
Isi queue (dari depan ke belakang):  
[0] 20  
[1] 30
```

Gambar 7.5 Pengimplementasian Queue

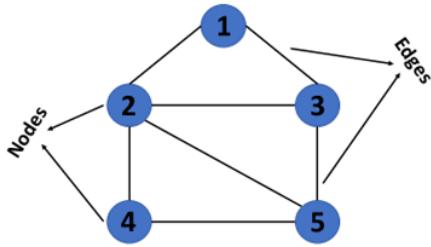
Berikut ilustrasi dari *Output* diatas:



Gambar 7.6 Ilustrasi Output Program Queue

C. GRAPH

Graph adalah struktur data yang terdiri dari sekumpulan simpul (*vertex*) dan sisi (*edge*) yang menghubungkan simpul-simpul tersebut. Struktur ini digunakan untuk merepresentasikan hubungan antar objek, seperti jaringan komputer, peta jalan, atau relasi sosial. *Graph* dapat bersifat berarah (arah koneksi antar simpul ditentukan) atau tak berarah (hubungan dua simpul bersifat dua arah). Selain itu, *Graph* juga bisa memiliki bobot pada setiap sisi, yang menunjukkan nilai tertentu dalam hubungan, seperti jarak antara kota atau biaya suatu koneksi. Karena fleksibilitasnya, *Graph* banyak digunakan dalam berbagai algoritma pencarian jalur, analisis jaringan, dan sistem rekomendasi.

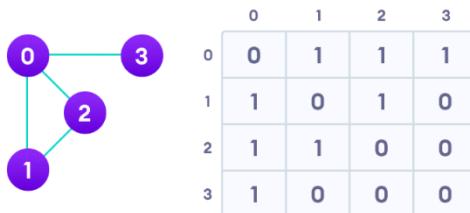


Gambar 7.7 Graph
Sumber: simplilearn

Di dalam Go, *Graph* dapat direpresentasikan menggunakan *Adjacency Matrix* dan *Adjacency List*, yang masing-masing memiliki kelebihan dan kekurangan tergantung pada kebutuhan aplikasi.

Graph dengan Adjacency Matrix

Adjacency Matrix adalah cara merepresentasikan *Graph* menggunakan matriks 2D, di mana setiap elemen menunjukkan apakah ada koneksi antara dua simpul. Jika *Graph* berarah, elemen $[i][j]$ dalam matriks menunjukkan ada sisi dari simpul i ke simpul j . Jika *Graph* tidak berarah, maka $[i][j]$ dan $[j][i]$ akan memiliki nilai yang sama. Representasi ini berguna untuk *Graph* kecil atau yang memiliki banyak hubungan antar simpul, karena aksesnya cepat melalui indeks array.



Gambar 7.8 Graph dengan Adjacency Matrix
Sumber: Programiz

Contoh Program yang Mengimplementasikan Graph Adjacency Matrix:

```

1. package main
2.
3. import "fmt"
4.
5. type Graph struct {
6.     vertices int
7.     matrix [][]int
8. }
9.
10. func NewGraph(size int) *Graph {
11.     matrix := make([][]int, size)
12.     for i := range matrix {
13.         matrix[i] = make([]int, size)
14.     }
15.     return &Graph{vertices: size, matrix: matrix}

```

```

16. }
17.
18. func (g *Graph) AddEdge(from, to int) {
19.     g.matrix[from][to] = 1
20.     g.matrix[to][from] = 1
21. }
22.
23. func (g *Graph) Display() {
24.     for _, row := range g.matrix {
25.         fmt.Println(row)
26.     }
27. }
28.
29. func main() {
30.     graph := NewGraph(4)
31.
32.     graph.AddEdge(0, 1)
33.     graph.AddEdge(1, 2)
34.     graph.AddEdge(2, 3)
35.     graph.AddEdge(3, 0)
36.     fmt.Println("Adjacency Matrix:")
37.     graph.Display()
38. }
```

Penjelasan Kode Tiap Bagian

- Membuat Graph

```

5. type Graph struct {
6.     vertices int
7.     matrix [][]int
8. }
```

Pada *Adjacency Matrix*, struktur *graph* terdiri dari dua bagian utama yaitu `vertices int` untuk menyimpan jumlah simpul dalam *Graph* dan `Matrix [][]int` yang merupakan sebuah matriks 2D yang merepresentasikan hubungan antar simpul. Nilai default dalam matriks adalah 0, yang berarti tidak ada koneksi antar simpul.

- Inisialisasi Graph

```

10. func NewGraph(size int) *Graph {
11.     matrix := make([][]int, size)
12.     for i := range matrix {
13.         matrix[i] = make([]int, size)
14.     }
15.     return &Graph{vertices: size, matrix: matrix}
16. }
```

Fungsi `NewGraph()` digunakan untuk membuat *Graph* dengan jumlah simpul tertentu. Jika menggunakan *Adjacency Matrix*, fungsi ini akan membuat matriks 2D berukuran *size x size* yang berisi 0 sebagai nilai *default*, menandakan tidak ada koneksi antar simpul.

- Menambahkan Edge ke Graph

```
18. func (g *Graph) AddEdge(from, to int) {
19.     g.matrix[from][to] = 1
20.     g.matrix[to][from] = 1
21. }
```

Fungsi `AddEdge()` digunakan untuk menambahkan koneksi antar simpul dalam *Graph*. Dalam *Adjacency Matrix*, nilai `1` dimasukkan ke indeks `[from][to]`, menunjukkan bahwa simpul `from` terhubung ke `to`. Jika *Graph* bersifat tak berarah, maka `to` juga akan terhubung kembali ke `from`.

- Menampilkan Graph

```
23. func (g *Graph) Display() {
24.     for _, row := range g.matrix {
25.         fmt.Println(row)
26.     }
27. }
```

Fungsi `Display()` digunakan untuk mencetak isi *Graph* dalam bentuk *Adjacency Matrix*. Setiap baris menunjukkan hubungan antar simpul dalam *Graph*, dengan angka `1` menandakan ada koneksi dan angka `0` menandakan tidak ada koneksi.

- Fungsi main

```
29. func main() {
30.     graph := NewGraph(4)
31.
32.     graph.AddEdge(0, 1)
33.     graph.AddEdge(1, 2)
34.     graph.AddEdge(2, 3)
35.     graph.AddEdge(3, 0)
36.     fmt.Println("Adjacency Matrix:")
37.     graph.Display()
38. }
```

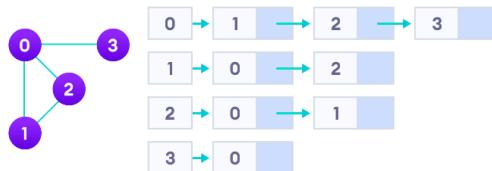
Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run graph_AdjMatrix.go
Adjacency Matrix:
[0 1 0 1]
[1 0 1 0]
[0 1 0 1]
[1 0 1 0]
```

Gambar 7.9 Graph dengan Adjacency Matrix

Graph dengan Adjacency List

Adjacency List adalah cara merepresentasikan *Graph* menggunakan daftar (*list*) yang menyimpan simpul-simpul yang terhubung dengan masing-masing simpul. Cara ini lebih efisien dalam penggunaan memori untuk *Graph* yang memiliki sedikit koneksi dibandingkan *Adjacency Matrix*, karena hanya menyimpan daftar koneksi yang diperlukan.



Gambar 7.10 Graph dengan Adjacency List
Sumber: Programiz

Contoh Program yang Mengimplementasikan Graph Adjacency List:

```

1. package main
2.
3. import "fmt"
4.
5. type Graph struct {
6.     vertices int
7.     adjList map[int][]int
8. }
9.
10. func NewGraph(size int) *Graph {
11.     return &Graph{vertices: size, adjList: make(map[int][]int)}
12. }
13.
14. func (g *Graph) AddEdge(from, to int) {
15.     g.adjList[from] = append(g.adjList[from], to)
16.     g.adjList[to] = append(g.adjList[to], from)
17. }
18.
19. func (g *Graph) Display() {
20.     for key, value := range g.adjList {
21.         fmt.Println(key, ":", value)
22.     }
23. }
24.
25. func main() {
26.     graph := NewGraph(4)
27.
28.     graph.AddEdge(0, 1)
29.     graph.AddEdge(1, 2)
30.     graph.AddEdge(2, 3)
31.     graph.AddEdge(3, 0)
32.
33.     fmt.Println("Adjacency List:")
34.     graph.Display()
35. }
```

Penjelasan Kode Tiap Bagian

- Membuat Graph

```

5. type Graph struct {
6.     vertices int
7.     adjList map[int][]int
8. }
```

Dalam *Adjacency List*, *Graph* terdiri dari `vertices int` yang menyimpan jumlah simpul dalam *Graph* dan `adjList map[int] []int` yang menggunakan map, di mana setiap simpul memiliki daftar simpul yang terhubung dengannya. Cara ini lebih hemat memori dibandingkan matriks.

- Inisialisasi Graph

```
10. func NewGraph(size int) *Graph {  
11.     return &Graph{vertices: size, adjList: make(map[int][]int)}  
12. }
```

Dalam *Adjacency List*, fungsi ini menginisialisasi map kosong (`adjList`), di mana setiap simpul akan memiliki daftar simpul yang terhubung dengannya. Representasi ini lebih hemat memori untuk *Graph* yang memiliki sedikit koneksi antar simpul.

- Menambahkan Edge ke Graph

```
14. func (g *Graph) AddEdge(from, to int) {  
15.     g.adjList[from] = append(g.adjList[from], to)  
16.     g.adjList[to] = append(g.adjList[to], from)  
17. }
```

Dalam *Adjacency List*, elemen `to` ditambahkan ke dalam daftar simpul yang terhubung dengan `from`, dan sebaliknya untuk *Graph* tak berarah. Cara ini lebih efisien dalam penggunaan memori dibandingkan matriks.

- Menampilkan Edge

```
19. func (g *Graph) Display() {  
20.     for key, value := range g.adjList {  
21.         fmt.Println(key, ":", value)  
22.     }  
23. }
```

Dalam *Adjacency List*, program mencetak daftar simpul yang terhubung dengan masing-masing simpul. Format ini lebih ringkas dibandingkan matriks, terutama untuk *Graph* dengan sedikit koneksi.

- Fungsi main

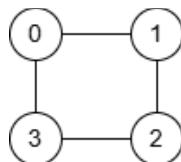
```
25. func main() {  
26.     graph := NewGraph(4)  
27.  
28.     graph.AddEdge(0, 1)  
29.     graph.AddEdge(1, 2)  
30.     graph.AddEdge(2, 3)  
31.     graph.AddEdge(3, 0)  
32.  
33.     fmt.Println("Adjacency List:")  
34.     graph.Display()  
35. }
```

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run graph_AdjList.go
Adjacency List:
2 : [1 3]
3 : [2 0]
0 : [1 3]
1 : [0 2]
```

Gambar 7.11 Graph dengan Adjacency List

Berikut ilustrasi *Output Graph* dengan *Adjacency Matrix* dan *List*:



Gambar 7.12 Ilustrasi Output kedua program Graph

D. TREE

Struktur data *tree* adalah bentuk struktur data *non-linear* yang digunakan untuk menyimpan data secara hierarkis. *Tree* terdiri dari *node* yang saling terhubung melalui relasi *parent-child*. Berikut kelebihan dan kekurangan *Tree*:

Kelebihan:

- Representasi data hierarkis yang alami
- Efisien untuk operasi hierarki dan rekursif
- Mudah diimplementasikan dengan *Pointer*

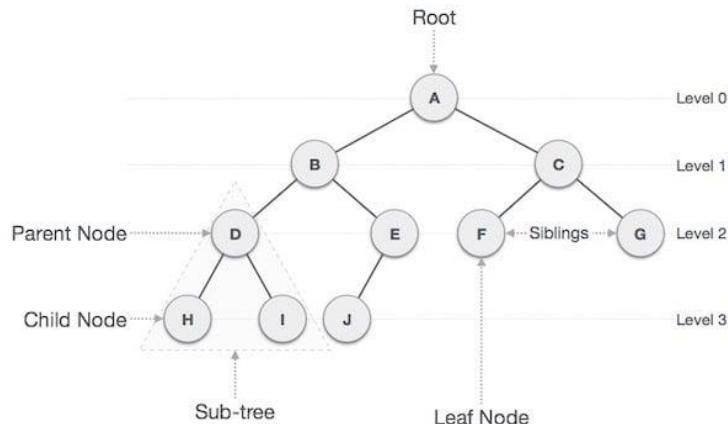
Kekurangan:

- Di dalam Go, *Tree* tidak tersedia sebagai struktur bawaan
- Potensi rekursi berlebihan (*Stack Overflow*)
- Manajemen *Pointer* bisa menjadi rumit

Beberapa istilah penting dalam *tree*:

- *Root* : *Node* paling atas dari *tree*; satu-satunya *node* yang tidak memiliki *parent*.
- *Parent* : *Node* yang memiliki satu atau lebih *node* anak (*child*).
- *Child* : *Node* yang diturunkan dari *node* lain.
- *Leaf*: *Node* yang tidak memiliki anak; disebut juga daun.
- *Siblings* : *Node-node* yang memiliki *parent* yang sama.

- *Subtree* : Setiap *node* dengan seluruh turunannya membentuk *subtree*.
- *Height* : Jarak terjauh dari *root* ke *leaf*, dihitung dalam jumlah *edge* (sisi).
- *Depth* : Jarak dari *root* ke node tertentu.
- *Level* : Menunjukkan posisi *node* berdasarkan kedalamannya; *root* berada di level 0.



Gambar 7.13 Tree
Sumber: Medium

Contoh Program yang Mengimplementasikan Tree:

```

1. package main
2.
3. import "fmt"
4.
5. type Node struct {
6.     Value     string
7.     Children []*Node
8. }
9.
10. func (n *Node) AddChild(child *Node) {
11.     n.Children = append(n.Children, child)
12. }
13.
14. func PrintTree(n *Node, level int) {
15.     if n == nil {
16.         return
17.     }
18.
19.     for i := 0; i < level; i++ {
20.         fmt.Print("  ")
21.     }
22.     fmt.Println(n.Value)
23.
24.     for _, child := range n.Children {
25.         PrintTree(child, level+1)
26.     }
27. }
28.
29. func main() {
30.
31.     root := &Node{Value: "A"}

```

```

32.
33.     b := &Node{Value: "B"}
34.     c := &Node{Value: "C"}
35.     d := &Node{Value: "D"}
36.
37.     root.AddChild(b)
38.     root.AddChild(c)
39.     root.AddChild(d)
40.
41.     e := &Node{Value: "E"}
42.     f := &Node{Value: "F"}
43.     b.AddChild(e)
44.     b.AddChild(f)
45.
46.     g := &Node{Value: "G"}
47.     c.AddChild(g)
48.
49.     h := &Node{Value: "H"}
50.     i := &Node{Value: "I"}
51.     j := &Node{Value: "J"}
52.     d.AddChild(h)
53.     d.AddChild(i)
54.     d.AddChild(j)
55.
56.     fmt.Println("Struktur Tree:")
57.     PrintTree(root, 0)
58. }

```

Penjelasan Kode Tiap Bagian

- Pendefinisan Struct Node

```

5. type Node struct {
6.     Value    string
7.     Children []*Node
8. }

```

Bagian pertama dari kode mendefinisikan struktur `Node` yang merepresentasikan setiap simpul dalam *tree*. *Struct* ini memiliki dua atribut utama: `Value`, yang menyimpan nilai dari *node* bertipe *string*, dan `Children`, yaitu *slice* dari *pointer* `Node`, yang menyimpan daftar anak dari *node* tersebut.

- Fungsi AddChild

```

10. func (n *Node) AddChild(child *Node) {
11.     n.Children = append(n.Children, child)
12. }

```

Fungsi `AddChild` adalah metode dari *struct* `Node` yang digunakan untuk menambahkan simpul anak ke *node* induk. Parameter dari fungsi ini adalah *pointer* ke `Node` baru yang akan ditambahkan sebagai anak. Implementasinya cukup sederhana, yaitu dengan menggunakan fungsi `append` untuk menambahkan anak ke *slice* `Children` milik *node* induk.

- Fungsi PrintTree

```
14. func PrintTree(n *Node, level int) {
15.     if n == nil {
16.         return
17.     }
18.
19.     for i := 0; i < level; i++ {
20.         fmt.Print(" ")
21.     }
22.     fmt.Println(n.Value)
23.
24.     for _, child := range n.Children {
25.         PrintTree(child, level+1)
26.     }
27. }
```

Fungsi `PrintTree` berfungsi untuk menampilkan struktur *tree* dalam bentuk hierarki bertingkat. Fungsi ini menggunakan pendekatan rekursif dengan parameter tambahan *level* untuk mencatat kedalaman *node* saat ini dalam *tree*. Untuk setiap pemanggilan, nilai *node* dicetak dengan indentasi berupa spasi “ “ yang dikalikan dengan *level*-nya, sehingga *node* pada level yang lebih dalam akan tampak menjorok ke kanan. Kemudian, fungsi ini akan memanggil dirinya sendiri untuk setiap anak dari *node* tersebut.

- Fungsi main

```
29. func main() {
30.
31.     root := &Node{Value: "A"}
32.
33.     b := &Node{Value: "B"}
34.     c := &Node{Value: "C"}
35.     d := &Node{Value: "D"}
36.
37.     root.AddChild(b)
38.     root.AddChild(c)
39.     root.AddChild(d)
40.
41.     e := &Node{Value: "E"}
42.     f := &Node{Value: "F"}
43.     b.AddChild(e)
44.     b.AddChild(f)
45.
46.     g := &Node{Value: "G"}
47.     c.AddChild(g)
48.
49.     h := &Node{Value: "H"}
50.     i := &Node{Value: "I"}
51.     j := &Node{Value: "J"}
52.     d.AddChild(h)
53.     d.AddChild(i)
54.     d.AddChild(j)
55.
56.     fmt.Println("Struktur Tree:")
57.     PrintTree(root, 0)
58. }
```

Fungsi main berperan sebagai titik awal untuk menguji struktur *tree* yang telah didefinisikan. Proses dimulai dengan pembuatan *node* induk (*root*) bernama “A” menggunakan `&Node{Value: "A"}`. Node ini menjadi titik utama atau akar dari seluruh struktur *tree*. Selanjutnya, tiga *node* anak bernama “B” dan “C” dan “D” ditambahkan ke *root* “A” menggunakan metode `AddChild`, sehingga *node* “A” kini memiliki tiga cabang utama. Untuk memperluas cabang:

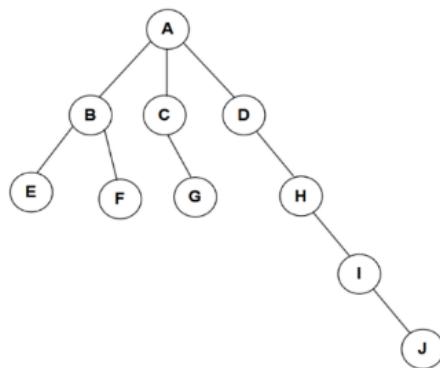
- Pada node “B”, ditambahkan dua anak yaitu “E” dan “F”. Proses ini dilakukan dengan memanggil `AddChild` langsung pada node b, sehingga “B” memiliki dua anak, yaitu “E” dan “F”
- Pada node “C” ditambahkan satu anak yaitu “G”. Ini dilakukan dengan `c.AddChild(g)`.
- Pada node “D”, ditambahkan tiga anak yaitu “H”, “I”, dan “J” menggunakan metode `d.AddChild(...)` secara berurutan.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run tree.go
Struktur Tree:
A
 B
  E
  F
 C
  G
 D
  H
  I
  J
```

Gambar 7.14 Pengimplementasian Tree

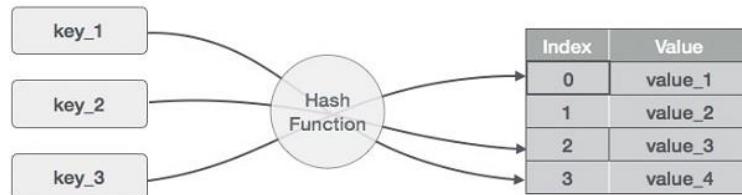
Berikut ilustrasi dari *Output* di atas:



Gambar 7.15 Ilustrasi Output program Tree

E. HASH

Hash adalah teknik pemetaan data dari domain yang besar ke domain yang lebih kecil (biasanya bilangan bulat terbatas) menggunakan fungsi deterministik bernama fungsi *hash* (*hash* Fungsi). Dalam struktur data, hashing digunakan untuk mempercepat pencarian dan penyimpanan, seperti di *hash table*, *hash map*, dan set. Dalam konteks keamanan digunakan untuk menjamin integritas data, seperti dalam digital *signature* dan *password hashing*.



Gambar 7.16 Hash Function

Sumber: Open Source For You

Kelebihan:

- Operasi sangat cepat rata-rata: pencarian, penyisipan, dan penghapusan hanya O(1).
- Tidak perlu traversal seperti *linked list* atau *tree*.
- Cocok untuk data tidak terurut namun sering diakses.

Kekurangan:

- Rentan terhadap kolisi.
- Tidak mempertahankan urutan data.
- Pemilihan fungsi hash yang buruk bisa membuat performa jatuh drastis.
- Tidak cocok untuk *range query* atau urutan terurut.

Contoh Program yang Mengimplementasikan Hash:

```
1. package main
2.
3. import (
4.     "bufio"
5.     "crypto/sha256"
6.     "fmt"
7.     "os"
8.     "strings"
9. )
10.
11. func generateSHA256Hash(data string) string {
12.     hash := sha256.Sum256([]byte(data))
13.     return fmt.Sprintf("%x", hash)
14. }
15. func compareHashes(hash1, hash2 string) bool {
16.     return hash1 == hash2
}
```

```

17. }
18.
19. func main() {
20.     reader := bufio.NewReader(os.Stdin)
21.     fmt.Println("== Program Hashing SHA-256 ==")
22.     fmt.Print("Masukkan data pertama: ")
23.     input1, _ := reader.ReadString('\n')
24.     input1 = strings.TrimSpace(input1)
25.     if input1 == "" {
26.         fmt.Println("Input tidak boleh kosong.")
27.         return
28.     }
29.     fmt.Print("Masukkan data kedua: ")
30.     input2, _ := reader.ReadString('\n')
31.     input2 = strings.TrimSpace(input2)
32.
33.     if input2 == "" {
34.         fmt.Println("Input tidak boleh kosong.")
35.         return
36.     }
37.     hash1 := generateSHA256Hash(input1)
38.     hash2 := generateSHA256Hash(input2)
39.     fmt.Println("\n== Hasil Hashing ==")
40.     fmt.Printf("Data 1      : %s\n", input1)
41.     fmt.Printf("Hash SHA-256: %s\n", hash1)
42.     fmt.Printf("Data 2      : %s\n", input2)
43.     fmt.Printf("Hash SHA-256: %s\n", hash2)
44.     fmt.Println("\nApakah kedua hash sama?")
45.     if compareHashes(hash1, hash2) {
46.         fmt.Println("Ya, hash sama.")
47.     } else {
48.         fmt.Println("Tidak, hash berbeda.")
49.     }
50. }
```

Penjelasan Kode Tiap Bagian

- Import Package

```

1. package main
2.
3. import (
4.     "bufio"
5.     "crypto/sha256"
6.     "fmt"
7.     "os"
8.     "strings"
9. )
```

Bagian ini mendefinisikan program sebagai `main`, yang artinya ini adalah titik masuk utama ketika program dijalankan. Lalu, beberapa package di-*import* untuk mendukung berbagai fungsionalitas. `bufio` digunakan untuk membaca input dari pengguna secara efisien. `crypto/sha256` menyediakan algoritma *hashing* SHA-256. `fmt` dipakai untuk mencetak output dan memformat teks. `os` digunakan untuk mengakses *input* dari terminal. Terakhir, `strings` membantu dalam manipulasi teks seperti menghapus spasi berlebih.

- Fungsi Hashing SHA-256

```
11. func generateSHA256Hash(data string) string {
12.     hash := sha256.Sum256([]byte(data))
13.     return fmt.Sprintf("%x", hash)
14. }
```

Fungsi ini bertugas menghasilkan nilai *hash* dari input *string*. Input *string* diubah terlebih dahulu menjadi *array byte*, karena fungsi SHA-256 bekerja dengan data *byte*. Hasil hash yang berupa *byte array* kemudian diubah ke dalam format *string* heksadesimal agar bisa ditampilkan dengan mudah.

- Fungsi Perbandingan Hash

```
15. func compareHashes(hash1, hash2 string) bool {
16.     return hash1 == hash2
17. }
```

Fungsi ini menerima dua *string hash* sebagai parameter dan membandingkannya secara langsung. Jika kedua *false* identik, maka fungsi akan mengembalikan nilai *true*, yang berarti kedua *input* yang di-*hash* sama nilainya. Jika tidak, hasilnya *false*.

- Inisialisasi Pembaca Input

```
20. reader := bufio.NewReader(os.Stdin)
```

Pada baris ini, dibuat sebuah objek *reader* dari package *bufio* yang akan membaca *input* dari *os.Stdin*, yaitu terminal. Ini memungkinkan pengguna untuk mengetikkan teks yang kemudian bisa diproses oleh program.

- Input dan Validasi Data Pertama

```
22. fmt.Print("Masukkan data pertama: ")
23. input1, _ := reader.ReadString('\n')
24. input1 = strings.TrimSpace(input1)
25. if input1 == "" {
26.     fmt.Println("Input tidak boleh kosong.")
27.     return
28. }
```

Program meminta pengguna untuk memasukkan data pertama, lalu membacanya sampai pengguna menekan Enter. Fungsi *TrimSpace* digunakan untuk menghapus spasi atau *newline* di awal dan akhir input. Jika ternyata input kosong, maka program menampilkan peringatan dan langsung berhenti.

- Input Data Kedua

```
29. fmt.Print("Masukkan data kedua: ")
30. input2, _ := reader.ReadString('\n')
31. input2 = strings.TrimSpace(input2)
```

Setelah data pertama diterima dan valid, program melanjutkan untuk meminta data kedua dari pengguna. Proses membaca dan membersihkan *inputnya* sama dengan data pertama, namun kali ini tidak ada validasi apakah kosong atau tidak.

- Proses Hashing

```
37.     hash1 := generateSHA256Hash(input1)
38.     hash2 := generateSHA256Hash(input2)
```

Kedua input yang sudah dibersihkan kemudian diproses menggunakan fungsi `generateSHA256Hash`. Masing-masing hasilnya disimpan dalam `hash1` dan `hash2`. Ini adalah tahap utama di mana *input* dikonversi menjadi nilai hash.

- Menampilkan Hasil Hashing

```
39.     fmt.Println("\n==== Hasil Hashing ===")
40.     fmt.Printf("Data 1      : %s\n", input1)
41.     fmt.Printf("Hash SHA-256: %s\n", hash1)
42.     fmt.Printf("Data 2      : %s\n", input2)
43.     fmt.Printf("Hash SHA-256: %s\n", hash2)
44.     fmt.Println("\nApakah kedua hash sama?")
```

Program kemudian menampilkan hasil *hash* untuk masing-masing data. *Output* diformat agar rapi dan mudah dibaca, menampilkan data asli dan hasil hashingnya dalam urutan yang jelas.

- Perbandingan Hash

```
45. if compareHashes(hash1, hash2) {
46.     fmt.Println("Ya, hash sama.")
47. } else {
48.     fmt.Println("Tidak, hash berbeda.")
49. }
50. }
```

Di bagian akhir, program menanyakan apakah kedua nilai hash yang dihasilkan sama. Fungsi `compareHashes` dipanggil untuk membandingkannya. Jika hasilnya sama, maka output akan menyatakan bahwa *hash* cocok, sebaliknya akan muncul pesan bahwa *hash* berbeda.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run hash.go
== Program Hashing SHA-256 ==
Masukkan data pertama: safira222
Masukkan data kedua: safira234

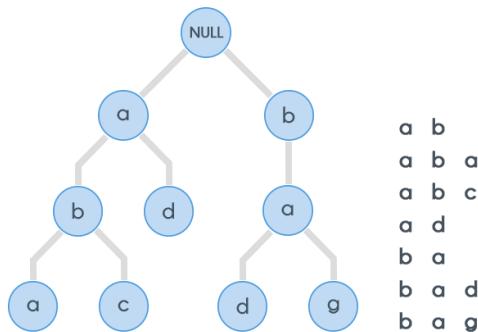
== Hasil Hashing ==
Data 1      : safira222
Hash SHA-256: 563d13fc...a9ec181112203aaebfe8cc34b40d3d5
Data 2      : safira234
Hash SHA-256: db19d71dce...ea59c0399ec627c449bd6

Apakah kedua hash sama?
Tidak, hash berbeda.
```

Gambar 7.17 Pengimplementasian Hash

F. TRIE

Struktur data *Trie* adalah struktur mirip pohon yang digunakan untuk menyimpan serangkaian *string* dinamis. *Trie*, juga dikenal sebagai *Prefix Tree*, adalah struktur data pohon yang digunakan secara efisien untuk menyimpan dan mengambil *string*, khususnya ketika berurusan dengan kumpulan kata yang memiliki awalan yang sama. Nama “*Trie*” berasal dari kata “*retrieval*” karena struktur ini sangat optimal untuk operasi pencarian *string* berdasarkan awalan (*prefix*). Setiap *node* dalam *trie* mewakili sebuah karakter. Kata-kata dibangun melalui jalur dari akar (*root*) ke daun, dan setiap *node* memiliki sekumpulan anak berupa *map* karakter ke *node* berikutnya.



Gambar 7.18 Trie
Sumber: OpenGenius IQ

Ciri khas utama *trie* dibandingkan struktur data *string* lain adalah:

- Efisien pencarian *prefix*: cocok untuk fitur *autocomplete*, *spell checking*, pencarian cepat.
- Menghemat waktu: pencarian kata dalam *trie* memiliki kompleksitas $O(n)$, di mana n adalah panjang kata.
- Ramah memori dalam konteks tertentu: meskipun penggunaan *map* atau *array* bisa besar, *trie* mampu menyimpan banyak kata dengan awalan sama secara kompak.

Kelebihan:

- Efisien untuk Pencarian *String* Berbasis *Prefix*
- Menghemat Memori untuk Data Berawalan Sama
- Kompleksitas Pencarian Tergantung Panjang Kata ($O(n)$)

Kekurangan:

- Struktur Lebih Kompleks Dibanding Array/*List*
- Konsumsi Memori Bisa Besar pada Skala Besar
- Kurang Efektif untuk Dataset Kecil

Contoh Program yang Mengimplementasikan Trie:

```
1. package main
2.
3. import "fmt"
4.
5. type TrieNode struct {
6.     children map[rune]*TrieNode
7.     isEndOfWord bool
8. }
9.
10. type Trie struct {
11.     root *TrieNode
12. }
13.
14. func NewTrie() *Trie {
15.     return &Trie{
16.         root: &TrieNode{children: make(map[rune]*TrieNode)},
17.     }
18. }
19.
20. func (t *Trie) Insert(word string) {
21.     current := t.root
22.     for _, char := range word {
23.         if _, exists := current.children[char]; !exists {
24.             current.children[char] = &TrieNode{children: make(map[rune]*TrieNode)}
25.         }
26.         current = current.children[char]
27.     }
28.     current.isEndOfWord = true
29. }
30.
31. func (t *Trie) Search(word string) bool {
32.     current := t.root
33.     for _, char := range word {
34.         if _, exists := current.children[char]; !exists {
35.             return false
36.         }
37.         current = current.children[char]
38.     }
39.     return current.isEndOfWord
40. }
41.
42. func (t *Trie) Autocomplete(prefix string) []string {
43.     current := t.root
44.     for _, char := range prefix {
45.         if _, exists := current.children[char]; !exists {
46.             return []string{}
47.         }
48.         current = current.children[char]
49.     }
50.
51.     results := []string{}
52.     t.collectWords(current, prefix, &results)
53.     return results
54. }
55.
56. func (t *Trie) collectWords(node *TrieNode, prefix string, results *[]string) {
57.     if node.isEndOfWord {
58.         *results = append(*results, prefix)
59.     }
60.     for char, child := range node.children {
61.         t.collectWords(child, prefix+string(char), results)
```

```

62.     }
63.   }
64.
65. func main() {
66.   trie := NewTrie()
67.   kata := []string{"go", "golang", "game", "gopher", "gopay", "gold", "gone"}
68.   for _, word := range kata {
69.     trie.Insert(word)
70.   }
71.
72.   fmt.Println("Autocomplete untuk 'go':")
73.   saran := trie.AutoComplete("go")
74.   for _, s := range saran {
75.     fmt.Println(" -", s)
76.   }
77. }
```

Penjelasan Kode Tiap Bagian

- Membuat struct TrieNode

```

5. type TrieNode struct {
6.   children map[rune]*TrieNode
7.   isEndOfWord bool
8. }
```

Struct `TrieNode` diatas digunakan untuk merepresentasikan simpul dalam struktur *trie*. Setiap simpul memiliki properti `children`, yaitu *map* dari karakter ke simpul anak, serta `isEndOfWord` yang menandakan apakah simpul ini adalah akhir dari sebuah kata.

- Membuat struct Trie utama

```

10. type Trie struct {
11.   root *TrieNode
12. }
```

Struct `Trie` hanya memiliki satu anggota, yaitu `root` yang merupakan akar dari pohon *trie*. Akar ini menjadi titik awal untuk semua operasi pencarian atau penyisipan kata.

- Fungsi konsutruktor untuk Trie

```

14. func NewTrie() *Trie {
15.   return &Trie{
16.     root: &TrieNode{children: make(map[rune]*TrieNode)},
17.   }
18. }
```

Fungsi `NewTrie()` membuat dan mengembalikan sebuah objek `Trie` baru dengan *node* akar yang telah diinisialisasi. Pada `root` ini *map* `children` telah siap digunakan.

- Fungsi untuk menyisipkan kata ke dalam trie

```

20. func (t *Trie) Insert(word string) {
21.   current := t.root
22.   for _, char := range word {
```

```

23.     if _, exists := current.children[char]; !exists {
24.         current.children[char] = &TrieNode{children: make(map[rune]*TrieNode)}
25.     }
26.     current = current.children[char]
27. }
28. current.isEndOfWord = true
29. }
```

Fungsi `Insert` diatas menelusuri *trie* karakter demi karakter. Jika karakter belum ada dalam `map children` dari simpul saat ini, maka simpul baru dibuat. Proses ini berlanjut hingga semua karakter dalam string `word` dimasukkan, lalu simpul terakhir ditandai sebagai akhir kata dengan `isEndOfWord = true`.

- Fungsi untuk mencari apakah sebuah kata ada dalam *trie*

```

31. func (t *Trie) Search(word string) bool {
32.     current := t.root
33.     for _, char := range word {
34.         if _, exists := current.children[char]; !exists {
35.             return false
36.         }
37.         current = current.children[char]
38.     }
39.     return current.isEndOfWord
40. }
```

Fungsi `Search` memeriksa apakah semua karakter dalam *string word* ada di jalur *trie*. Jika salah satu karakter tidak ditemukan, fungsi langsung mengembalikan `false`. Jika semua karakter ditemukan dan *node* terakhir merupakan akhir kata (`isEndOfWord==true`), maka fungsi mengembalikan `tr`.

- Fungsi Autocomplete berdasarkan prefix

```

42. func (t *Trie) Autocomplete(prefix string) []string {
43.     current := t.root
44.     for _, char := range prefix {
45.         if _, exists := current.children[char]; !exists {
46.             return []string{}
47.         }
48.         current = current.children[char]
49.     }
```

Fungsi ini digunakan untuk mencari semua kata yang diawali dengan *string prefix*. Pertama, *trie* ditelusuri untuk mencapai simpul akhir dari *prefix* tersebut. Jika *prefix* tidak ditemukan, maka *array* kosong dikembalikan. Jika ditemukan, fungsi akan memanggil `collectWords` untuk mengumpulkan seluruh kemungkinan kata yang memiliki awalan tersebut.

- Fungsi rekursif untuk mengumpulkan kata berdasarkan prefix

```

56. func (t *Trie) collectWords(node *TrieNode, prefix string, results *[]string) {
57.     if node.isEndOfWord {
58.         *results = append(*results, prefix)
59.     }
60.     for char, child := range node.children {
```

```
61.     t.collectWords(child, prefix+string(char), results)
62. }
63. }
```

Fungsi ini berjalan rekursif ke setiap anak simpul. Jika simpul saat ini merupakan akhir dari kata, *prefix* yang telah terbentuk ditambahkan ke dalam *slice* hasil. Fungsi ini akan menjelajahi seluruh jalur hingga semua kata valid terkumpul.

- Fungsi main

```
65. func main() {
66.     trie := NewTrie()
67.     kata := []string{"go", "golang", "game", "gopher", "gopay", "gold", "gone"}
68.     for _, word := range kata {
69.         trie.Insert(word)
70.     }
71.
72.     fmt.Println("Autocomplete untuk 'go':")
73.     saran := trie.AutoComplete("go")
74.     for _, s := range saran {
75.         fmt.Println(" -", s)
76.     }
77. }
```

Di dalam fungsi `main`, pertama-tama dibuat objek `trie` baru. Kemudian sejumlah kata disisipkan ke dalam *trie*. Setelah itu, fungsi `Autocomplete` digunakan untuk menampilkan semua kata yang diawali dengan `"go"`. Hasilnya ditampilkan di konsol.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run trie.go
Autocomplete untuk 'go':
- go
- golang
- gold
- gopher
- gopay
- gone
```

Gambar 7.19 Pengimplementasian Trie

BAB 8

IMPLEMENTASI

A. SISTEM ANTRIAN PADA RUMAH SAKIT

Pengelolaan antrian merupakan salah satu komponen krusial dalam sistem pelayanan publik, terutama di sektor kesehatan seperti rumah sakit. Ketidakteraturan dalam proses pelayanan pasien berpotensi menimbulkan berbagai permasalahan, seperti keterlambatan penanganan medis, penurunan kualitas layanan, hingga meningkatnya risiko keselamatan pasien. Oleh karena itu, dibutuhkan suatu sistem manajemen antrian yang efisien, adil, dan adaptif terhadap kondisi prioritas pasien.

Pada sub-bab ini menyajikan studi kasus yang mengilustrasikan penerapan struktur data dalam perancangan sistem manajemen antrian rumah sakit. Sistem yang dikembangkan harus mampu mengelompokkan pasien berdasarkan tingkat urgensi penanganan medis. Pasien dengan kondisi darurat harus memperoleh prioritas lebih tinggi dibanding pasien reguler, namun dengan tetap mempertimbangkan waktu kedatangan sebagai parameter sekunder. Untuk mengakomodasi kebutuhan ini, struktur data seperti *priority queue* atau *heap* menjadi pilihan yang tepat, mengingat kemampuannya dalam menyusun elemen berdasarkan tingkat prioritas secara efisien.

Contoh Program:

```
1. package main
2.
3. import (
4.     "container/heap"
5.     "fmt"
6.     "time"
7. )
8.
9. type Pasien struct {
10.     Nama         string
11.     Kondisi      string
12.     Prioritas    int
13.     WaktuDatang  time.Time
14. }
15.
16. type AntrianDarurat []*Pasien
17.
18. func (pq AntrianDarurat) Len() int { return len(pq) }
19.
20. func (pq AntrianDarurat) Less(i, j int) bool {
21.     return pq[i].Prioritas < pq[j].Prioritas
22. }
23.
24. func (pq AntrianDarurat) Swap(i, j int) {
25.     pq[i], pq[j] = pq[j], pq[i]
26. }
27.
28. func (pq *AntrianDarurat) Push(x any) {
29.     item := x.(*Pasien)
30.     *pq = append(*pq, item)
31. }
32.
```

```

33. func (pq *AntrianDarurat) Pop() any {
34.     old := *pq
35.     n := len(old)
36.     item := old[n-1]
37.     *pq = old[0 : n-1]
38.     return item
39. }
40.
41. type AntrianReguler []*Pasien
42. /
43. func (q *AntrianReguler) Enqueue(p *Pasien) {
44.     *q = append(*q, p)
45. }
46.
47. func (q *AntrianReguler) Dequeue() *Pasien {
48.     if len(*q) == 0 {
49.         return nil
50.     }
51.     item := (*q)[0]
52.     *q = (*q)[1:]
53.     return item
54. }
55.
56. func (q *AntrianReguler) IsEmpty() bool {
57.     return len(*q) == 0
58. }
59.
60. func LayananPasien(darurat *AntrianDarurat, reguler *AntrianReguler) {
61.     fmt.Println("Urutan Pelayanan Pasien:")
62.     for darurat.Len() > 0 || !reguler.IsEmpty() {
63.         if darurat.Len() > 0 {
64.             // Layani pasien darurat terlebih dahulu
65.             p := heap.Pop(darurat).(*Pasien)
66.             fmt.Printf("[DARURAT] %s (Prioritas: %d)\n", p.Nama, p.Prioritas)
67.         } else if !reguler.IsEmpty() {
68.             p := reguler.Dequeue()
69.             fmt.Printf("[REGULER] %s (Datang: %s)\n",
70.                         p.Nama, p.WaktuDatang.Format("15:04:05"))
71.         }
72.     }
73.
74. func main() {
75.     darurat := &AntrianDarurat{}
76.     reguler := &AntrianReguler{}
77.
78.     heap.Init(darurat)
79.
80.     heap.Push(darurat, &Pasien{Nama: "Pak Budi", Kondisi: "darurat", Prioritas: 1})
81.     heap.Push(darurat, &Pasien{Nama: "Bu Siti", Kondisi: "darurat", Prioritas: 2})
82.     heap.Push(darurat, &Pasien{Nama: "Pak Darto", Kondisi: "darurat", Prioritas: 0})
83.
84.     now := time.Now()
85.     reguler.Enqueue(&Pasien{Nama: "Andi", Kondisi: "reguler", WaktuDatang: now.Add(1 * time.Minute)})

```

```

86.     reguler.Enqueue(&Pasien{Nama: "Rina", Kondisi: "reguler", WaktuDatang:
87.         now.Add(2 * time.Minute)})
88.     reguler.Enqueue(&Pasien{Nama: "Sari", Kondisi: "reguler", WaktuDatang:
89.         now.Add(3 * time.Minute)})
90.     LayananPasien(darurat, reguler)
91. }

```

Penjelasan Kode Tiap Bagian

- Pendefinisian Struct Pasien

```

9. type Pasien struct {
10.    Nama      string
11.    Kondisi   string
12.    Prioritas int
13.    WaktuDatang time.Time
14. }

```

Struct Pasien digunakan untuk merepresentasikan data pasien. Setiap pasien memiliki `Nama`, `Kondisi` (darurat/reguler), `Prioritas` (hanya untuk darurat), dan `WaktuDatang` (hanya digunakan untuk pasien reguler). *Struct* ini akan digunakan dalam dua jenis antrian.

- Antrian Darurat sebagai *Priority Queue*

```

16. type AntrianDarurat []*Pasien
17.
18. func (pq AntrianDarurat) Len() int { return len(pq) }
19.
20. func (pq AntrianDarurat) Less(i, j int) bool {
21.     return pq[i].Prioritas < pq[j].Prioritas
22. }
23.
24. func (pq AntrianDarurat) Swap(i, j int) {
25.     pq[i], pq[j] = pq[j], pq[i]
26. }
27.
28. func (pq *AntrianDarurat) Push(x any) {
29.     item := x.(*Pasien)
30.     *pq = append(*pq, item)
31. }
32.
33. func (pq *AntrianDarurat) Pop() any {
34.     old := *pq
35.     n := len(old)
36.     item := old[n-1]
37.     *pq = old[0 : n-1]
38.     return item
39. }

```

Tipe AntrianDarurat merupakan slice dari pointer ke Pasien, dan diimplementasikan sebagai *priority queue* menggunakan package *container/heap*. Untuk mendukung struktur *heap*, beberapa method harus diimplementasikan:

1. `Len()`, `Less(i, j int)` dan `Swap(i, j int)` digunakan oleh package heap untuk mengatur posisi elemen.
2. `Push(x any)` dan `Pop() any` adalah metode yang dipanggil saat menambahkan atau menghapus elemen dari *heap*. Fungsi `Less` diatur agar pasien dengan angka prioritas lebih kecil (lebih darurat) diproses lebih dulu.

- Antrian Reguler sebagai Queue FIFO

```

41. type AntrianReguler []*Pasien
42.
43. func (q *AntrianReguler) Enqueue(p *Pasien) {
44.     *q = append(*q, p)
45. }
46.
47. func (q *AntrianReguler) Dequeue() *Pasien {
48.     if len(*q) == 0 {
49.         return nil
50.     }
51.     item := (*q)[0]
52.     *q = (*q)[1:]
53.     return item
54. }
55.
56. func (q *AntrianReguler) IsEmpty() bool {
57.     return len(*q) == 0
58. }
```

Tipe AntrianReguler juga berupa slice dari *pointer* ke Pasien, namun diperlukan sebagai antrian biasa. Ada tiga metode:

1. `Enqueue(p *Pasien)` untuk menambahkan pasien ke akhir antrian.
2. `Dequeue() *Pasien` untuk menghapus dan mengembalikan pasien dari awal antrian.
3. `IsEmpty()` untuk mengecek apakah antrian kosong.

- Fungsi LayananPasien

```

59. func LayananPasien(darurat *AntrianDarurat, reguler *AntrianReguler) {
60.     fmt.Println("Urutan Pelayanan Pasien:")
61.     for darurat.Len() > 0 || !reguler.IsEmpty() {
62.         if darurat.Len() > 0 {
63.             // Layani pasien darurat terlebih dahulu
64.             p := heap.Pop(darurat).(*Pasien)
65.             fmt.Printf("[DARURAT] %s (Prioritas: %d)\n", p.Nama, p.Prioritas)
66.         } else if !reguler.IsEmpty() {
67.             p := reguler.Dequeue()
68.             fmt.Printf("[REGULER] %s (Datang: %s)\n",
69.                         p.Nama, p.WaktuDatang.Format("15:04:05"))
70.         }
71.     }
```

Fungsi ini bertugas untuk melayani pasien dari dua jenis antrian. Selama masih ada pasien dalam antrian darurat atau reguler, fungsi akan melayani pasien. Prioritas diberikan ke antrian darurat terlebih dahulu. Jika antrian darurat kosong, maka pasien dari antrian reguler akan diproses berdasarkan urutan kedatangan. Informasi pasien yang dilayani ditampilkan ke layar.

- Fungsi main

```
72. func main() {
73.     darurat := &AntrianDarurat{}
74.     reguler := &AntrianReguler{}
75.
76.     heap.Init(darurat)
77.
78.     heap.Push(darurat, &Pasien{Nama: "Pak Budi", Kondisi: "darurat",
    Prioritas: 1})
79.     heap.Push(darurat, &Pasien{Nama: "Bu Siti", Kondisi: "darurat",
    Prioritas: 2})
80.     heap.Push(darurat, &Pasien{Nama: "Pak Darto", Kondisi: "darurat",
    Prioritas: 0})
81.
82.     now := time.Now()
83.     reguler.Enqueue(&Pasien{Nama: "Andi", Kondisi: "reguler", WaktuDatang:
    now.Add(1
        time.Minute)})
84.     reguler.Enqueue(&Pasien{Nama: "Rina", Kondisi: "reguler", WaktuDatang:
    now.Add(2
        time.Minute)})
85.     reguler.Enqueue(&Pasien{Nama: "Sari", Kondisi: "reguler", WaktuDatang:
    now.Add(3
        time.Minute)})
86.     LayananPasien(darurat, reguler)
87. }
```

Pada fungsi `main`, dua antrian (`darurat` dan `reguler`) diinisialisasi. Kemudian, antrian darurat diinisialisasi sebagai `heap` dan diisi dengan tiga pasien dengan prioritas berbeda. Lalu antrian reguler diisi dengan tiga pasien yang datang dengan selang waktu berbeda menggunakan `time.Now().Add(...)`. Setelah itu, fungsi `LayananPasien` dipanggil untuk memproses seluruh pasien.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run antrian_rumahSakit.go
Urutan Pelayanan Pasien:
[DARURAT] Pak Darto (Prioritas: 0)
[DARURAT] Pak Budi (Prioritas: 1)
[DARURAT] Bu Siti (Prioritas: 2)
[REGULER] Andi (Datang: 21:14:03)
[REGULER] Rina (Datang: 21:15:03)
[REGULER] Sari (Datang: 21:16:03)
```

Gambar 8.1 Output Sistem Antrian Rumah Sakit

B. SISTEM NAVIGASI KOTA SEDERHANA

Seiring dengan pertumbuhan pesat infrastruktur jalan di kota-kota besar, sistem navigasi yang efisien sangat dibutuhkan untuk membantu pengguna dalam menemukan jalur tercepat atau terpendek antara satu lokasi dengan lokasi lainnya. Permasalahan yang muncul adalah bagaimana merepresentasikan struktur jalan kota yang kompleks, mencari *route* optimal dalam waktu singkat, serta mengakomodasi kondisi jalan seperti satu arah, jarak, atau jalan yang tertutup. Sistem navigasi kota bertujuan untuk menyelesaikan masalah ini dengan memberikan solusi navigasi yang akurat dan *real-time*.

Untuk menyelesaikan permasalahan tersebut digunakan beberapa struktur data, yakni *graf*, *adjacency list*, *heap* dan *map*. Dalam sistem navigasi kota, *graf* digunakan untuk merepresentasikan jaringan jalan, di mana simpul (*node*) mewakili setiap sisi (*edge*) menghubungkan jalan antar lokasi dengan bobot yang menunjukkan jarak waktu tempuh. Kemudian *Priority queue* yang diimplementasikan dengan *heap*, membantu memilih simpul dengan bobot terkecil secara efisien, mempercepat pencarian jalur yang optimal menggunakan algoritma dikstra. Selain itu, *map* digunakan untuk menyimpan data lokasi dan rute dengan akses cepat berdasarkan *key*, memungkinkan manajemen data yang efisien.

Contoh Program:

```
1. package main
2.
3. import (
4.     "container/heap"
5.     "fmt"
6. )
7. type Graph struct {
8.     nodes map[string]map[string]float64
9.     closed map[string]map[string]bool
10. }
11. func NewGraph() *Graph {
12.     return &Graph{
13.         nodes: make(map[string]map[string]float64),
14.         closed: make(map[string]map[string]bool),
15.     }
16. }
17. func (g *Graph) AddEdge(from, to string, weight float64, isClosed bool) {
18.     if _, exists := g.nodes[from]; !exists {
19.         g.nodes[from] = make(map[string]float64)
20.     }
21.     g.nodes[from][to] = weight
22.
23.     if _, exists := g.closed[from]; !exists {
24.         g.closed[from] = make(map[string]bool)
25.     }
26.     g.closed[from][to] = isClosed
27. }
28. type Item struct {
29.     node    string
30.     priority float64
```

```

31.     index    int
32. }
33. type PriorityQueue []*Item
34.
35. func (pq PriorityQueue) Len() int {
36.     return len(pq)
37. }
38.
39. func (pq PriorityQueue) Less(i, j int) bool {
40.     return pq[i].priority < pq[j].priority
41. }
42.
43. func (pq PriorityQueue) Swap(i, j int) {
44.     pq[i], pq[j] = pq[j], pq[i]
45. }
46.
47. func (pq *PriorityQueue) Push(x interface{}) {
48.     item := x.(*Item)
49.     *pq = append(*pq, item)
50. }
51.
52. func (pq *PriorityQueue) Pop() interface{} {
53.     old := *pq
54.     n := len(old)
55.     item := old[n-1]
56.     *pq = old[0 : n-1]
57.     return item
58. }
59.
60. func (g *Graph) Dijkstra(start, end string) (float64, []string) {
61.     pq := &PriorityQueue{}
62.     heap.Init(pq)
63.
64.     dist := make(map[string]float64)
65.     prev := make(map[string]string)
66.     for node := range g.nodes {
67.         dist[node] = 1<<63 - 1
68.     }
69.     dist[start] = 0
70.     heap.Push(pq, &Item{node: start, priority: 0})
71.     for pq.Len() > 0 {
72.         current := heap.Pop(pq).(*Item)
73.
74.         if current.node == end {
75.             break
76.         }
77.
78.         for neighbor, weight := range g.nodes[current.node] {
79.             if g.closed[current.node][neighbor] {
80.                 continue
81.             }
82.
83.             alt := dist[current.node] + weight
84.             if alt < dist[neighbor] {
85.                 dist[neighbor] = alt
86.                 prev[neighbor] = current.node
87.                 heap.Push(pq, &Item{node: neighbor, priority: alt})
88.             }
89.         }

```

```

90.     }
91.
92.     var path []string
93.     node := end
94.     for node != "" {
95.         path = append([]string{node}, path...)
96.         node = prev[node]
97.     }
98.
99.     return dist[end], path
100. }
101.
102. func main() {
103.     g := NewGraph()
104.     g.AddEdge("A", "B", 5, false)
105.     g.AddEdge("A", "C", 10, false)
106.     g.AddEdge("B", "C", 3, false)
107.     g.AddEdge("B", "D", 2, true)
108.     g.AddEdge("C", "D", 1, false)
109.     g.AddEdge("D", "A", 7, false)
110.
111.     distance, path := g.Dijkstra("A", "D")
112.     fmt.Printf("Jarak terpendek: %.2f\n", distance)
113.     fmt.Println("Rute terpendek:", path)
114. }
```

Penjelasan Kode Tiap Bagian

- Import Package

```

1. package main
2.
3. import (
4.     "container/heap"
5.     "fmt"
6. )
```

Pada kode program diatas `fmt`: Untuk menampilkan *output* ke konsol. `container/heap`: Menyediakan implementasi *heap (priority queue)*, yang penting untuk efisiensi algoritma Dijkstra agar simpul dengan jarak terkecil diproses lebih dulu.

- Struktur Graph

```

7. type Graph struct {
8.     nodes map[string]map[string]float64
9.     closed map[string]map[string]bool
10. }
```

Pada kode program diatas adalah struktur *graph*, `nodes` Peta dari simpul asal ke simpul tujuan, dengan bobot sebagai nilai (misalnya: A → B = 5). `closed` Peta *boolean* untuk menentukan apakah jalan dari satu simpul ke simpul lainnya tertutup (`true`) atau terbuka (`false`).

- Fungsi Inisialisasi Graph

```

11. func NewGraph() *Graph {
12.     return &Graph{
13.         nodes: make(map[string]map[string]float64),
14.         closed: make(map[string]map[string]bool),
15.     }
16. }
```

Pada kode program diatas adalah untuk membuat fungsi *graph* baru, menginisialisasi *graf* kosong dengan peta *nodes* dan *closed*. Menggunakan *make ()* karena *map* harus diinisialisasi sebelum digunakan.

- Menambahkan Edge ke Graph

```

17. func (g *Graph) AddEdge(from, to string, weight float64, isClosed bool) {
18.     if _, exists := g.nodes[from]; !exists {
19.         g.nodes[from] = make(map[string]float64)
20.     }
21.     g.nodes[from][to] = weight
22.
23.     if _, exists := g.closed[from]; !exists {
24.         g.closed[from] = make(map[string]bool)
25.     }
26.     g.closed[from][to] = isClosed
27. }
```

Pada program diatas adalah menambah *edge*, memastikan simpul asal (*from*) sudah ada di *nodes* dan *Closed*. Menambahkan simpul tujuan (*to*) dengan bobot (*weight*) dan status jalan (*isClosed*). Contoh: A → B = 5, terbuka (*isClosed = false*).

- Struktur Item dalam Priority Queue

```

1. type Item struct {
2.     node    string
3.     priority float64
4.     index   int
5. }
```

Pada kode program diatas merupakan *type PriorityQueue* *Item* . *node*: Nama simpul *priority*: Jarak dari simpul awal ke simpul ini (untuk Dijkstra, makin kecil makin prioritas). *index*: Posisi dalam *heap* (untuk kebutuhan internal *heap*).

```
6. type PriorityQueue []*Item
```

Kode program ini adalah definisi *Priority Queue*, dimana Slice dari pointer *Item*, digunakan sebagai *heap*.

- Definisi dan Implementasi Priority Queue

```

35. func (pq PriorityQueue) Len() int{
36.     return len(pq)
37. }
38. func (pq PriorityQueue) Less(i, j int) bool {
```

```

39.     return pq[i].priority < pq[j].priority
40. }
41.
42. func (pq PriorityQueue) Swap(i, j int) {
43.     pq[i], pq[j] = pq[j], pq[i]
44. }
45.
46. func (pq *PriorityQueue) Push(x interface{}) {
47.     item := x.(*Item)
48.     *pq = append(*pq, item)
49. }

```

Implementasi Interface Heap (untuk *PriorityQueue*) *Len*, *Less*, *Swap* Diperlukan agar slice bisa berfungsi sebagai *heap (min-heap)*. *Push* Menambahkan item ke *priority queue*. *Pop* mengambil item dengan prioritas tertinggi (jarak terkecil). Berikut, dibawah ini merupakan Algoritma Dijkstra:

```

51. func (g *Graph) Dijkstra(start, end string) (float64, []string) {

```

- Inisialisasi priority queue.

```

52. pq := &PriorityQueue{}
53. heap.Init(pq)
54.
55. dist := make(map[string]float64)
56. prev := make(map[string]string)

```

dist: Menyimpan jarak terpendek dari *start* ke setiap simpul.

prev: Menyimpan simpul sebelumnya dalam *rute* untuk rekonstruksi jalur.

```

57. for node := range g.nodes {
58.     dist[node] = 1<<63 - 1
59. }
60. dist[start] = 0

```

Semua jarak di-set ke "tak hingga" ($1<<63 - 1$ = nilai *float64* maksimum). Jarak simpul awal (start) di-set ke 0. Masukkan simpul awal ke *priority queue*.

```

61. heap.Push(pq, &Item{node: start, priority: 0})

```

- Proses Iterasi Dijkstra

```

62. for pq.Len() > 0 {
63.     current := heap.Pop(pq).(*Item)
64.
65.     if current.node == end {
66.         break
67.     }
68.
69.     for neighbor, weight := range g.nodes[current.node] {
70.         if g.closed[current.node][neighbor] {
71.             continue
72.         }
73.     }
74. }

```

Selama *priority queue* tidak kosong: Ambil simpul dengan prioritas tertinggi (jarak terpendek). Jika simpul tujuan sudah ditemukan (`current.node == end`), berhenti. Untuk setiap tetangga (`neighbor`) dari simpul saat ini. Jika jalan tertutup, lewati (`continue`).

- Iterasi Algoritma Dijkstra

```
74.     alt := dist[current.node] + weight
75.     if alt < dist[neighbor] {
76.         dist[neighbor] = alt
77.         prev[neighbor] = current.node
78.         heap.Push(pq, &Item{node: neighbor, priority: alt})
79.     }
80. }
81. }
```

Hitung jarak baru `alt`. Jika jarak baru lebih kecil, perbarui `dist` dan `prev`, lalu masukkan ke *priority queue*.

- Menyusun Jalur Terpendek

```
83. var path []string
84. node := end
85. for node != "" {
86.     path = append([]string{node}, path...)
87.     node = prev[node]
88. }
89.
90. return dist[end], path
91. }
```

Setelah perhitungan selesai, bangun jalur dari `end` ke start dengan menelusuri `prev`. Hasil akhir adalah slice `path` dari awal ke akhir.

- Fungsi main

```
93. func main() {
94.     g := NewGraph()
95.     g.AddEdge("A", "B", 5, false)
96.     g.AddEdge("A", "C", 10, false)
97.     g.AddEdge("B", "C", 3, false)
98.     g.AddEdge("B", "D", 2, true)
99.     g.AddEdge("C", "D", 1, false)
100.    g.AddEdge("D", "A", 7, false)
```

Membuat graf dan menambahkan jalan-jalan antar simpul.

Contoh:

A → B bobot 5 (terbuka) B → D bobot 2 (tertutup)

```
102. distance, path := g.Dijkstra("A", "D")
103. fmt.Printf("Jarak terpendek: %.2f\n", distance)
104. fmt.Println("Rute terpendek:", path)
105. }
```

Memanggil algoritma Dijkstra untuk mencari jalur dari A ke D. Menampilkan jarak dan *rute* ke layar.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run navigasiKota.go
Jarak terpendek: 9.00
Rute terpendek: [A B C D]
```

Gambar 8.2 Output Sistem Navigasi Kota

C. SISTEM UNDO REDO DALAM PENGOLAHAN DATA DINAMIS

Dalam dunia pemrograman, pengelolaan data yang fleksibel dan efisien adalah aspek penting yang sering ditemui dalam berbagai aplikasi. Sistem yang dapat menangani perubahan data secara dinamis, termasuk kemampuan untuk menghapus dan memulihkan data, sangat diperlukan dalam berbagai skenario, seperti manajemen dokumen, pengeditan gambar, serta pengelolaan basis data. Salah satu tantangan utama adalah memastikan bahwa data yang dihapus tidak hilang sepenuhnya, sehingga dapat dipulihkan (undo) jika diperlukan, dan bahkan memungkinkan pembatalan pemulihannya (redo) untuk fleksibilitas yang lebih tinggi.

Konsep *Linked List* dan *Stack* adalah dua struktur data yang sangat berguna dalam mengatasi permasalahan ini. *Linked List* memungkinkan penyimpanan dan manipulasi data secara dinamis tanpa keterbatasan ukuran seperti *array statis*, sementara *Stack* memberikan mekanisme pencatatan perubahan data dengan prinsip *Last In, First Out (LIFO)*, memungkinkan penyimpanan elemen yang dihapus agar bisa dipulihkan kembali. Dengan kombinasi kedua struktur ini, sistem pengelolaan data dapat memiliki fleksibilitas tinggi serta kemampuan untuk melakukan undo dan redo secara efisien, memberikan pengguna kontrol lebih besar terhadap data yang mereka kelola. Implementasi ini tidak hanya meningkatkan pengalaman pengguna, tetapi juga menjadi dasar bagi pengembangan sistem yang lebih kompleks dengan fitur manajemen data yang lebih canggih.

Contoh Program:

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type Node struct {
8.     data int
9.     next *Node
10. }
11.
12. type LinkedList struct {
13.     head *Node
14. }
15.
16. type Stack struct {
17.     items []int }
```

```

18. }
19.
20. func (ll *LinkedList) Add(data int) {
21.     newNode := &Node{data: data}
22.     if ll.head == nil {
23.         ll.head = newNode
24.     } else {
25.         current := ll.head
26.         for current.next != nil {
27.             current = current.next
28.         }
29.         current.next = newNode
30.     }
31.     fmt.Println("Menambahkan:", data)
32. }
33.
34. func (ll *LinkedList) Remove(value int, undoStack *Stack) {
35.     if ll.head == nil {
36.         fmt.Println("Tidak ada data untuk dihapus")
37.         return
38.     }
39.     var prev *Node
40.     current := ll.head
41.
42.     for current != nil && current.data != value {
43.         prev = current
44.         current = current.next
45.     }
46.     if current != nil {
47.         undoStack.Push(current.data)
48.         if prev != nil {
49.             prev.next = current.next
50.         } else {
51.             ll.head = current.next
52.         }
53.         fmt.Println("Menghapus:", value)
54.     } else {
55.         fmt.Println("Data tidak ditemukan")
56.     }
57. }
58.
59. func (ll *LinkedList) Undo(undoStack *Stack, redoStack *Stack) {
60.     data, success := undoStack.Pop()
61.     if success {
62.         ll.Add(data)
63.         redoStack.Push(data)
64.         fmt.Println("Undo:", data)
65.     }
66. }
67.
68. func (ll *LinkedList) Redo(redoStack *Stack) {
69.     data, success := redoStack.Pop()
70.     if success {
71.         ll.Remove(data, &Stack{})
72.         fmt.Println("Redo:", data)
73.     }
74. }
75.
76. func (ll *LinkedList) Print() {

```

```

77.     current := ll.head
78.     fmt.Println("Linked List: ")
79.     for current != nil {
80.         fmt.Printf("%d -> ", current.data)
81.         current = current.next
82.     }
83.     fmt.Println("nil")
84. }
85.
86. func (s *Stack) Push(data int) {
87.     s.items = append(s.items, data)
88. }
89.
90. func (s *Stack) Pop() (int, bool) {
91.     if len(s.items) == 0 {
92.         fmt.Println("Tidak ada data untuk diproses")
93.         return 0, false
94.     }
95.     index := len(s.items) - 1
96.     data := s.items[index]
97.     s.items = s.items[:index]
98.     return data, true
99. }
100.
101. func main() {
102.     ll := &LinkedList{}
103.     undoStack := &Stack{}
104.     redoStack := &Stack{}
105.     ll.Add(10)
106.     ll.Add(20)
107.     ll.Add(30)
108.     ll.Print()
109.
110.    ll.Remove(20, undoStack)
111.    ll.Print()
112.
113.    ll.Undo(undoStack, redoStack)
114.    ll.Print()
115.
116.    ll.Redo(redoStack)
117.    ll.Print()
118. }

```

Penjelasan Kode Tiap Bagian

- Membuat Node untuk Linked List

```

7.     type Node struct {
8.         data int
9.         next *Node
10.    }

```

Struktur `Node` mendefinisikan elemen dalam *linked list*. `data` menyimpan nilai utama, sedangkan `next` berfungsi sebagai *pointer* yang mengarah ke *node* berikutnya dalam daftar. Dengan pendekatan berbasis *pointer*, *linked list* bisa bertambah secara dinamis tanpa batasan ukuran awal. Jika `next == nil`, berarti *node* tersebut adalah *node* terakhir dalam daftar.

- Membuat Linked List dengan Inisialisasi Head

```
12. type LinkedList struct {
13.     head *Node
14. }
```

`LinkedList` berfungsi sebagai wadah utama untuk menyimpan dan mengelola data yang berantai. `head` menunjuk ke *node* pertama dalam daftar, memungkinkan traversal serta operasi seperti *insert*, *delete*, dan *search*. *Linked list* tidak memiliki indeks tetap seperti *array*, sehingga semua operasi dilakukan dengan cara traversal dari `head`.

- Membuat Stack untuk Undo dan Redo

```
16. type Stack struct {
17.     items []int
18. }
```

`Stack` digunakan untuk menyimpan riwayat data yang dihapus untuk mendukung fitur undo dan redo. Stack diimplementasikan menggunakan `slice` (`[]int`), yang memungkinkan penyimpanan dan pengambilan elemen dengan prinsip *LIFO* (*Last In, First Out*). Dalam skenario ini, *stack* berfungsi sebagai *buffer* untuk data yang dihapus.

- Membuat Fungsi Penambahan Data ke Linked List

```
20. func (ll *LinkedList) Add(data int) {
21.     newNode := &Node{data: data}
22.     if ll.head == nil {
23.         ll.head = newNode
24.     } else {
25.         current := ll.head
26.         for current.next != nil {
27.             current = current.next
28.         }
29.         current.next = newNode
30.     }
31.     fmt.Println("Menambahkan:", data)
32. }
```

Fungsi `Add()` memasukkan *node* baru ke dalam *linked list*, dan dilakukan dengan cara traversal hingga mencapai *node* terakhir. Jika `head == nil`, elemen pertama (`newNode`) langsung menjadi `head`. Jika tidak, iterasi dilakukan dari `head` hingga menemukan *node* terakhir (`current.next == nil`), lalu `current.next` diarahkan ke `newNode`. Operasi ini bisa lebih efisien dengan menambahkan referensi `tail`, sehingga penambahan bisa dilakukan tanpa perlu traversal penuh.

- Membuat Fungsi Penghapusan Data dari Linked List

```
34. func (ll *LinkedList) Remove(value int, undoStack *Stack) {
35.     if ll.head == nil {
36.         fmt.Println("Tidak ada data untuk dihapus")
37.         return
38.     }
39.     var prev *Node
```

```

40.     current := ll.head
41.
42.     for current != nil && current.data != value {
43.         prev = current
44.         current = current.next
45.     }

```

Fungsi `Remove()` mencari *node* berdasarkan nilai (`value`) lalu menghapusnya dari daftar. Jika *node* ditemukan, *pointer next* dari *node* sebelumnya (`prev`) diarahkan ke *node* setelahnya (`current.next`). Jika *node* yang dihapus adalah `head`, maka `head` langsung diperbarui ke `current.next`. Data yang dihapus disimpan dalam `undoStack`, memastikan bahwa data bisa dipulihkan nanti. Perlu dicatat bahwa operasi ini memerlukan traversal hingga *node* yang cocok ditemukan, sehingga bisa dioptimalkan dengan referensi langsung ke *node* tertentu jika ada mekanisme pencarian yang lebih efisien.

- Fungsi Undo (Pemulihan Data yang Dihapus)

```

60. func (ll *LinkedList) Undo(undoStack *Stack, redoStack *Stack) {
61.     data, success := undoStack.Pop()
62.     if success {
63.         ll.Add(data)
64.         redoStack.Push(data)
65.         fmt.Println("Undo:", data)
66.     }
67. }

```

Fungsi `Undo()` mengembalikan elemen yang baru saja dihapus ke dalam *linked list* dengan mengambilnya dari `undoStack`. Setelah pemulihan, data juga dimasukkan ke `redoStack` untuk memungkinkan pembatalan undo. `Pop()` memastikan bahwa elemen yang terakhir dihapus adalah elemen pertama yang dipulihkan, sesuai dengan prinsip *stack*. Dengan mekanisme ini, pengguna dapat melakukan undo tanpa harus menyimpan seluruh riwayat perubahan secara eksplisit dalam *linked list*.

- Fungsi Redo (Pembatalan Undo)

```

69. func (ll *LinkedList) Redo(redoStack *Stack) {
70.     data, success := redoStack.Pop()
71.     if success {
72.         ll.Remove(data, &Stack{})
73.         fmt.Println("Redo:", data)
74.     }
75. }

```

Fungsi `Redo()` menghapus kembali elemen yang baru saja dipulihkan oleh `Undo()`. Elemen diambil dari `redoStack`, lalu langsung dihapus dari *linked list* menggunakan `Remove()`. Berbeda dengan penghapusan biasa, elemen yang dihapus dalam redo tidak dimasukkan ke `undoStack` kembali, karena redo harus memastikan bahwa perubahan sebelumnya tidak bisa di-undo kembali. Hal ini menghindari siklus berulang antara undo dan redo yang tak terbatas.

- Fungsi Menampilkan Isi Linked List

```
77. func (ll *LinkedList) Print() {  
78.     current := ll.head  
79.     fmt.Print("Linked List: ")  
80.     for current != nil {  
81.         fmt.Printf("%d -> ", current.data)  
82.         current = current.next  
83.     }  
84.     fmt.Println("nil")  
85. }
```

Fungsi `Print()` melakukan traversal dari `head`, mencetak setiap elemen hingga mencapai akhir daftar (`nil`). Format "`d -> d -> ... nil`" digunakan untuk menggambarkan hubungan antar *node*. Karena sifatnya sebagai *linked list*, traversal dilakukan secara *sequential* tanpa indeks seperti *array*.

- Fungsi Push dan Pop untuk Stack

```
87. func (s *Stack) Push(data int) {  
88.     s.items = append(s.items, data)  
89. }  
90.  
91. func (s *Stack) Pop() (int, bool) {  
92.     if len(s.items) == 0 {  
93.         fmt.Println("Tidak ada data untuk diproses")  
94.         return 0, false  
95.     }  
96.     index := len(s.items) - 1  
97.     data := s.items[index]  
98.     s.items = s.items[:index]  
99.     return data, true  
100. }
```

`Push()` menambahkan elemen ke *slice stack*, sementara `Pop()` mengambil elemen terakhir dan memperbarui *slice* agar elemen tersebut tidak lagi tersedia. Jika stack kosong, `Pop()` mengembalikan nilai *default* `0` dan `false` sebagai indikator kegagalan. Karena berbasis *slice*, kedua operasi ini dilakukan dengan efisiensi tinggi tanpa perlu *shifting* elemen secara *eksplisit*.

- Fungsi main

```
102. func main() {  
103.     ll := &LinkedList{}  
104.     undoStack := &Stack{}  
105.     redoStack := &Stack{}  
106.     ll.Add(10)  
107.     ll.Add(20)  
108.     ll.Add(30)  
109.     ll.Print()  
110.  
111.     ll.Remove(20, undoStack)  
112.     ll.Print()  
113.  
114.     ll.Undo(undoStack, redoStack)
```

```
115.    ll.Print()
116.
117.    ll.Redo(redoStack)
118.    ll.Print()
119. }
```

Fungsi `main()` menjalankan skenario pengujian terhadap seluruh sistem. `LinkedList` dan dua stack (`undoStack`, `redoStack`) diinisialisasi untuk mengelola riwayat perubahan. Elemen `10`, `20`, dan `30` dimasukkan ke dalam *linked list*. Setelah elemen `20` dihapus, perubahan tercermin dalam `undoStack`, memungkinkan `Undo()` untuk memulihkannya kembali. `Redo()` kemudian digunakan untuk menghapusnya lagi, menunjukkan bahwa pengguna dapat mengontrol data dengan fleksibilitas penuh.

Output:

```
PS C:\A\UM\VS CODE\6_Go\src> go run undo_redo.go
Menambahkan: 10
Menambahkan: 20
Menambahkan: 30
Linked List: 10 -> 20 -> 30 -> nil
Menghapus: 20
Linked List: 10 -> 30 -> nil
Menambahkan: 20
Undo: 20
Linked List: 10 -> 30 -> 20 -> nil
Menghapus: 20
Redo: 20
Linked List: 10 -> 30 -> nil
```

Gambar 8.3 Output Sistem Undo dan Redo dalam Pengolahan Data Dinamis

DAFTAR PUSTAKA

- Docker Documentation. (2024). *Introduction to Docker and Go*. Diakses dari <https://docs.docker.com>
- GeeksforGeeks. (n.d.). *Trie I (Insert and Search)*. Diakses dari <https://www.geeksforgeeks.org/trie-insert-and-search/>
- Go.dev. (2024). *Case studies and use cases*. Diakses dari <https://go.dev>
- Go.dev. (2024). *Downloads - Go programming language*. Diakses dari <https://go.dev/dl>
- Go.dev. (2024). *Interface - A tour of Go*. Diakses dari <https://go.dev/tour/methods/1>
- Go.dev. (2024). *Struct - A tour of Go*. Diakses dari <https://go.dev/tour/moretypes/2>
- Golang.org. (n.d.). *The Go programming language specification - Array types*. Diakses dari https://golang.org/ref/spec#Array_types
- Khairlambang, G., Taqwa, N., & Awangga, R. M. (2023). *Pengenalan bahasa Golang dan membuat package dengan Google API*.
- Prayogo, N. A. (2024). *Dasar pemrograman Golang (v4.0.20241115)*. Diakses dari <https://github.com/novalagung/dasarpemrogramangolang>
- Saragih, R. R. (2016). *Pemrograman dan bahasa pemrograman*. STMIK-STIE Mikroskil.
- Suhadi, M. S., Samperura, B., & Awangga, R. M. (2023). *Memulai pemrograman Go: Panduan mudah mengenal Golang*. PT. Penerbit Buku Pedia.
- The Go Authors. (n.d.). *Effective Go: Names*. Diakses dari https://go.dev/doc/effective_go#names