Types créés par l'utilisateur

Éric Jacoboni

8 février 2021

Université Jean Jaurès, Toulouse

Sommaire

Sommaire

Introduction

Prologue

Dans ce module, nous utiliserons Go pour implanter les concepts du cours.

Remarques

- *Le cours Go du premier semestre est supposé connu ! (si ce n'est pas le cas, vous savez ce qui vous reste à faire... le cours du premier semestre est encore en ligne)*
- *On ne peut pas apprendre un langage si on n'écrit pas de programmes avec : si vous vous contentez des 2 heures de cours par semaine, vous n'y arriverez pas.*

Les structures Go

- Au premier semestre, nous avons vu :
  - **Les types simples** : nombres et booléens.
  - **Les types composés** : chaînes de caractères et tranches (tableaux dynamiques).
  - Ces deux types composés sont *homogènes* : les chaînes ne sont composées que de caractères, les tranches regroupent des éléments *de même type*.
- Go permet également de créer des types composés *hétérogènes* : les *structures*. Ce sont elles que nous utiliserons dans ce cours.

Les structures Go

- La construction *struct* permet de regrouper zéro ou plusieurs valeurs – éventuellement de types différents – sous un même nom de type.
- Dans une *struct*, chaque valeur est appelé *champ* : cela ressemble beaucoup au concept de *type d'entité* et de *table SQL* vus en cours de BD.

Exemple de création d'un type Employe

```
type Employe struct {   // Le mot clé 'type' indique qu'on définit un nouveau type
    Id int
    Nom string
    Prenom string
    Taille float32
}
```

Exemple de création d'un Employe

```
var lui Employe    // Tous ses champs sont à "zéro"
moi := Employe{42, "Jacoboni", "Eric", 1.88}
```

Exemple d'utilisation d'un Employe

```
lui.Nom = "Dupont"      // On accède aux champs avec la notation pointée
if moi.Taille > 1.75 {...}
```

Configuration pour les TP

- Go est installé.
- Le répertoire *go* et ses sous-répertoires (*bin*, *pkg* et *src*) a été créé sous votre répertoire personnel et le commande *go env GOPATH* contient son chemin d'accès.
- Facultativement, vous avez installé Visual Code, et son extension Go
- Pour *chaque* exercice ou TP, vous créerez un répertoire portant le nom de l'exercice ou du TP sous le répertoire *go/src* (par exemple, *go/src/personne* ou *go/src/listechainee*). Ces noms seront tout en minuscules, sans accents ni espaces.
- Chacun de ces répertoires devra comporter un sous-répertoire *app* où vous placerez le fichier programme de test (que vous appelerez systématiquement *main.go*). Les autres fichiers seront placés directement dans le répertoire.
- Exemple :

```
% tree go/src/personne
go/src/personne
|- personne.go
|- app
    |- main.go
```

Application : création d'un type Personne

- Définir un type *Personne*. Une personne est décrite par son nom, son prénom, sa date de naissance (une chaîne de caractères au format *jj/mm/aaaa*) et son adresse (qui est elle-même formée de la rue, de la ville et du code postal).
- Faire en sorte que l'affichage d'une personne affiche correctement les informations.
- Écrire un sous-programme permettant de saisir une personne.
- Écrire un programme principal permettant de saisir plusieurs personnes, de les stocker dans une tranche, puis d'afficher ces personnes.

Implémentation d'un type Personne

- Pour définir l'adresse d'une personne, on va d'abord créer un type *Adresse* pour encapsuler les 3 composantes de celle-ci.
- Le type *Personne* aura donc 4 composantes : le nom, le prénom, la date de naissance et l'adresse. Les 3 premières sont des chaînes de caractères, la 4^{e} est du type *Adresse*

Fichier personne.go

```
package personne

import "fmt"

// Adresse permet de représenter une adresse postale
type Adresse struct {
    Rue, Ville, Cp string
}

// Personne permet de représenter une personne
type Personne struct {
    Prenom, Nom, DateNaissance string
    Adresse                    Adresse
}
```

Implémentation d'un type Personne

- Pour afficher correctement une *Adresse* et une *Personne*, la technique (identique à celle de Java et de nombreux autres langages orientés objets) consiste à *redéfinir* la méthode *String()* pour chacun de ces types (voir le cours de POO).
- On ajoute donc une méthode *String()* à chacun de ces types :

Fichier personne.go

```
// Redéfinition de la représentation textuelle d'une Adresse
func (a Adresse) String() string {
    return fmt.Sprintf("%s, %s %s", a.Rue, a.Cp, a.Ville)
}

// Redéfinition de la représentation textuelle d'une personne
func (p Personne) String() string {
    return fmt.Sprintf("%s %s, né le %s. Adresse : %s.", p.Prenom, p.Nom, p.DateNaissance, p.Adresse)
}
```

Fonctions vs. méthodes

- Une *fonction* Go est identique aux fonctions Python : elle prend (éventuellement) des paramètres et renvoie (éventuellement) un résultat. C'est toujours ce que nous avons utilisé pour coder nos sous-programmes.
- Une *méthode* Go est identique aux méthodes que l'on rencontre dans les langages orientés objets comme Java et Python : elle prend (éventuellement) des paramètres et renvoie (éventuellement) un résultat *mais elle s'applique à un objet* en utilisant une *notation pointée*.
- Les deux sous-programmes *String()* que nous venons d'ajouter sont des méthodes qui s'appliquent donc à des objets *Adresse* et *Personne*. Vous noterez que leurs définitions ont une partie supplémentaire placée avant leur nom, qui est une sorte de paramètre supplémentaire pour désigner l'objet sur lequel elles portent (leur « récepteur »).

Saisie d'une Adresse

- Pour saisir une personne, il suffit de saisir chacun de ses 4 champs.
- Si on sait déjà saisir une chaîne de caractères, il reste à définir la saisie d'une adresse : on écrira donc une fonction *SaisieAdresse()* qui effectuera cette tâche.

Fichier personne.go

```
func saisieAdresse() Adresse {
    var (
        res     Adresse
        scanner = bufio.NewScanner(os.Stdin)  // Ajouter "bufio" et "os" à la liste des import
    )
    fmt.Print("Rue : ")
    scanner.Scan()
    res.Rue = scanner.Text()
    fmt.Print("Ville : ")
    scanner.Scan()
    res.Ville = scanner.Text()
    fmt.Print("Code postal : ")
    scanner.Scan()
    res.Cp = scanner.Text()
    return res
}
```

Saisie d'une Personne

Fichier personne.go

```
// SaisiePersonne permet de saisir une personne au clavier
func SaisiePersonne() Personne {
    var (
        res     Personne
        scanner = bufio.NewScanner(os.Stdin)
    )
    fmt.Print("Prénom : ")
    scanner.Scan()
    res.Prenom = scanner.Text()
    fmt.Print("Nom : ")
    scanner.Scan()
    res.Nom = scanner.Text()
    fmt.Print("Date de naissance (JJ/MM/AAAA) : ")
    scanner.Scan()
    res.DateNaissance = scanner.Text()
    res.Adresse = saisieAdresse()
    return res
}
```

Remarque

On remarque que le motif fmt.Print(...) / scanner.Scan() / champ = scanner.Text() revient systématiquement...Il serait donc judicieux d'encapsuler ces trois opérations dans une fonction input() qui renverrait la valeur saisie...

Le programme principal

- Il reste à écrire un programme de test (dans le sous-répertoire *app*) qui saisit un ensemble de personnes puis les affiche.

Fichier app/main.go

```
package main

import (
    "fmt"
    "personne"
    "strings"
)

func main() {
    var personnes []personne.Personne // Une tranche de personnes

    // Saisie des personnes
    encore := true
    for encore {
        rep := ""
        personnes = append(personnes, personne.SaisiePersonne())
        fmt.Print("Encore (o/n) ? ")
        fmt.Scanln(&rep)
        encore = strings.HasPrefix(strings.ToUpper(rep), "O")
    }

    // Affichage de toutes les personnes stockées dans la tranche
    for _, pers := range personnes {
        fmt.Println(pers)                  // Appel implicite de pers.String()
    }
}
```

Application : création d'un type Fraction

- Pour implémenter une fraction, on a besoin d'encapsuler deux valeurs : le numérateur et le dénominateur – tous deux de type entier.
- Pour simplifier les opérations sur les fractions, il faut stocker les fractions *sous forme simplifiée* : 4/8 sera stockée comme 2/4.
- Il faut également tenir compte du signe du numérateur et du dénominateur et faire en sorte que ce soit le numérateur qui porte le signe.
- Enfin, pour afficher correctement des fractions, nous redéfinirons la méthode *String()* afin de représenter la fraction 3/4 par la chaîne "3/4"

Implémentation d'un type Fraction

- Pour simplifier une fraction, il suffit de diviser son numérateur et son dénominateur par leur PGCD : il faut donc écrire une fonction qui calcule le PGCD.
- Pour gérer le signe, le plus simple est de rendre positifs le numérateur et le dénominateur (en prenant leur valeur absolue) puis de multiplier le numérateur par -1 si la fraction est négative.
- Pour savoir si la fraction est négative, il suffit de multiplier le numérateur par le dénominateur et d'étudier le signe du résultat...

Implémentation d'un type Fraction en Go

Fichier fraction.go

```go
package fraction
import (
    "fmt"
    "math"
)

// Une fraction est formée d'un numérateur et d'un dénominateur entiers
type Fraction struct {
    num, den int
}

// Méthode de conversion d'une Fraction en string
func (f Fraction) String() string {
    return fmt.Sprintf("%d/%d", f.num, f.den)  // Voir cette page pour les différents formats possibles.
}

// Fonction utilitaire pour simplifier la fraction (voir cette page)
func pgcd(a, b int) int {
    if b == 0 {
        return a
    }
    return pgcd(b, a % b)
}
```

Implémentation d'un type Fraction en Go

Fichier fraction.go

```
// Fonction de création d'une fraction
func NewFraction(num, den int) Fraction {
    if den == 0 {
        panic("Dénominateur nul")     // Stoppe l'exécution du programme
    }
    signe := num * den
    num, den = int(math.Abs(float64(num))), int(math.Abs(float64(den)))
    pgcd := pgcd(num, den)
    res := Fraction{num/pgcd, den/pgcd}
    if signe < 0 {
        res.num = -res.num
    }
    return res
}

// Addition de deux Fractions
func Plus(f1, f2 Fraction) Fraction {
    return NewFraction(f1.num*f2.den+f1.den*f2.num, f1.den*f2.den)
}

// Multiplication de deux Fractions
func Mult(f1, f2 Fraction) Fraction {
    return NewFraction(f1.num*f2.num, f1.den*f2.den)
}
```

Utilisation du nouveau type Fraction

Fichier app/main.go

```
package main

import (
    "fmt"
    "fraction"
)

func main() {
    f1 := fraction.NewFraction(3, 4)
    f2 := fraction.NewFraction(-6, -8)
    f3 := fraction.NewFraction(-2, 1)
    f4 := fraction.Plus(f1, f2)
    f5 := fraction.Mult(f1, f2)
    fmt.Println(f1, f2, f3, f4, f5)  // Appel implicite de String() : 3/4 3/4 -2/1 3/2 9/16
}
```

Remarques

- *Ici, la « difficulté » est qu'on veut effectuer des opérations complexes au moment de la création de la Fraction, d'où la fonction NewFraction...*
- *Sinon, on aurait pu simplement écrire f1 := Fraction{3, 4}...Mais cela ne créait pas de Fraction normalisée...*
- *Si l'on n'avait pas redéfini String(), les fractions se seraient affichées sous la forme {3 4} {3 4} {-2 1} {3 2} {9 16}*

Rappels sur les pointeurs en Go

- Un pointeur est une valeur représentant l'*adresse* d'une variable.
- Pour tout type T, il existe un type $*T$ signifiant que la valeur est un pointeur vers une variable de type T.
- Pour toute variable v de type T, la notation $\&v$ renvoie l'adresse de v (qui est donc de type $*T$).
- Inversement, si p est un pointeur de type $*T$, la valeur pointée est notée $*p$ (et est donc de type T).
- Un pointeur peut également valoir *nil* : une valeur signifiant que le pointeur ne pointe sur rien.

Rappels sur les pointeurs en Go

Exemples de manipulation des pointeurs

```
val := 42               // val est un int et contient la valeur 42
pval := &val            // pval est un *int (un pointeur vers un int).
                        // pval "pointe" sur val puisqu'il contient son adresse
*pval = *pval + 2       // *pval est une autre façon d'accéder à val…
                        // val et *pval sont synonymes… tout comme &val et  pval
fmt.Println(val)        // 44

frac1 := fraction.NewFraction(3, 2)
frac2 := fraction.NewFraction(3, 4)
pfrac := &frac1
fmt.Println(fraction.Plus(*pfrac, frac2))   // 9/4
```

Utilisation des pointeurs pour les passages par référence

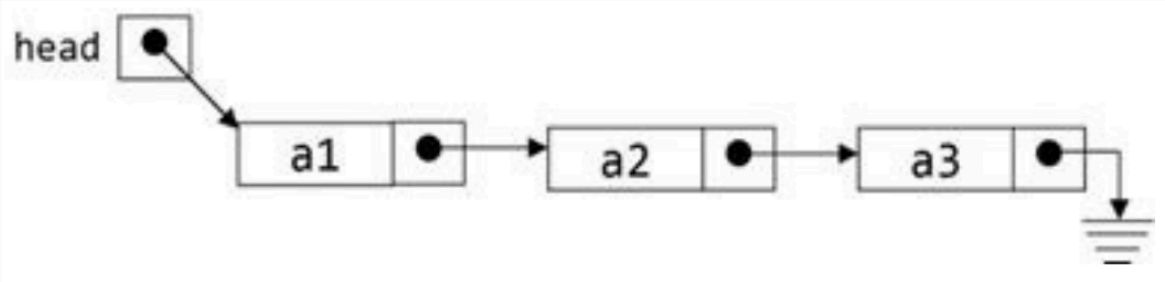
```
func swap(a, b *int) {      // a et b sont des pointeurs vers des int => passage par référence
    temp := *a              // temp est un int qui reçoit la valeur pointée par a
    *a = *b                 // la valeur pointée par a reçoit la valeur pointée par b
    *b = temp               // la valeur pointée par b reçoit la valeur de temp
}

val1, val2 := 10, 100
swap(&val1, &val2)         // désormais, val1 contient 100 et val2 contient 10
```

Types récursifs

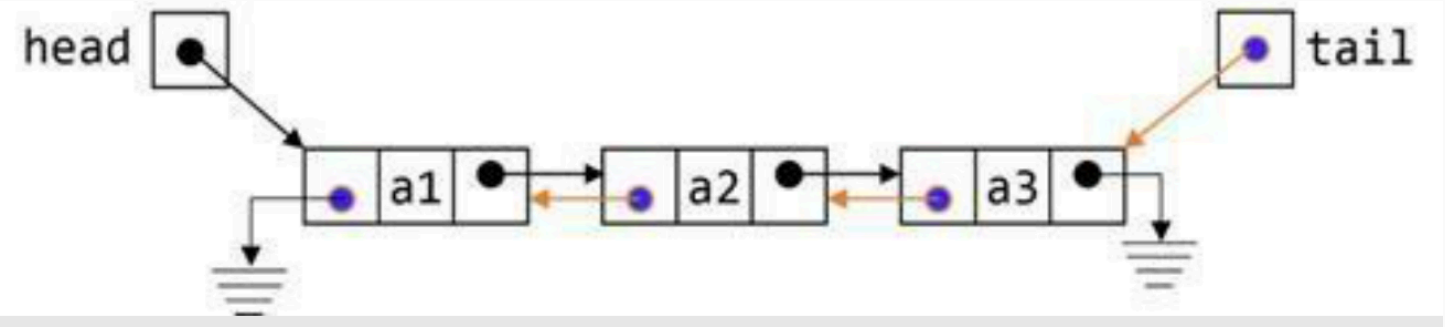
Structures récursives

- Une structure récursive est une structure dont l'un au moins des composants permet de désigner une autre instance de cette structure.
- Par exemple, une *liste simplement chaînée* peut être vue comme une suite de *cellules*, chaque cellule désignant la suivante au moyen d'un de ses composants. Ce composant est donc un pointeur vers une cellule.
- Une cellule de liste simplement chaînée est donc formée de deux composants : la valeur qu'elle contient, et un pointeur vers la cellule suivante. Elle permet un parcours unidirectionnel.



Structures récursives

- Une cellule de *liste doublement chaînée* est formée de trois composants : la valeur qu'elle contient, un pointeur vers la cellule précédente et un pointeur vers la cellule suivante. Elle permet donc un parcours bidirectionnel.



- L'avantage des listes chaînées est qu'elles ne sont pas contigües en mémoire et que l'on peut donc facilement ajouter ou supprimer des éléments « au milieu de la liste » : il suffit de créer/supprimer une cellule (la même chose avec une tranche nécessite de décaler les éléments...).

Cellule récursive en Go

En Go, on crée un type cellule à l'aide du constructeur *struct* :

Type cellule simplement chaînée en Go (fichier liste.go)

```
package liste

import "fmt"

type cellule struct {
    valeur  int                // On suppose que les valeurs seront des entiers...
    suivant *cellule           // Le champ suivant "pointe" vers une autre cellule
}

// Méthode de conversion d'une cellule en chaîne : redéfinition de la méthode String()
func (cell cellule) String() string {
    return fmt.Sprintf("%d", cell.valeur)
}
```

Remarque

Ici, le type cellule commence par une minuscule : il ne sera donc pas visible à l'extérieur du fichier `liste.go` *: l'utilisateur d'une liste chaînée n'a pas besoin de connaître les détails de son implémentation.*

Liste simplement chaînée

Une liste simplement chaînée est simplement définie par sa première cellule (sa « tête »). Le champ suivant de la dernière cellule de la liste contiendra la valeur spéciale *nil* pour indiquer qu'il n'y a rien après.

Type Liste en Go (fichier liste.go (suite))

```
type Liste struct {
    tete *cellule                          // nil par défaut
}

// Méthode de conversion d'une liste en chaîne : redéfinition de la méthode String()
func (ls Liste) String() string {
    res := ""
    for curr := ls.tete; curr != nil; curr = curr.suivant {
        res +=  curr.String() + " "
    }
    return res[:len(res) - 1] // Pour ôter l'espace final...
}
```

Remarque

Ici, le type Liste commence par une majuscule : il sera donc visible à l'extérieur du fichier `liste.go`.

Opérations sur une liste chaînée

Les opérations que l'on veut pouvoir réaliser sur une liste consistent à :

- Tester si la liste est vide (méthode *IsEmpty*).
- Renvoyer la longueur de la liste (méthode *Length*).
- Ajouter un élément au début de la liste (méthode *Cons*).
- Ajouter un élément à la fin de la liste (méthode *Append*).
- Renvoyer l'élément situé en tête de liste (méthode *Head*).
- Renvoyer la liste privée de son premier élément (méthode *Tail*).
- Rechercher un élément dans la liste (méthode *Search*).
- Supprimer un élément dans la liste (méthode *Remove*).
- Renvoyer l'élément situé à un « indice » indiqué (méthode *At*).
- Modifier l'élément situé à un « indice » indiqué (méthode *SetAt*).
- Convertir une liste chaînée en tranche Go (méthode *ToSlice*).

Opérations sur une liste chaînée

On veut donc pouvoir utiliser notre liste chaînée avec un programme comme celui-ci :

Exemple d'utilisation de la liste chaînée

```
package main

import (
    "fmt"
    "liste"
)

func main() {
    maListe := liste.Liste{}     // nom du paquetage.Nom du type
    maListe.Cons(42)
    maListe.Cons(2)
    maListe.Cons(10)
    fmt.Printf("Il y a %d éléments dans %s\n", maListe.Length(), maListe)

    if !maListe.IsEmpty() {
        fmt.Println("La liste n'est pas vide")
    }

    val, err := maListe.Head()
    if err == nil {
        fmt.Println(val)
    } else {
        fmt.Println(err)
    }
```

Exemple d'utilisation de la liste chaînée

```
    tail, err := maListe.Tail()
    if err == nil {
        fmt.Println(tail)
    } else {
        fmt.Println(err)
    }

    maListe.Append(20)
    maListe.Append(100)
    fmt.Printf("Il y a %d éléments dans %s\n", maListe.Length(), maListe)

    fmt.Println(maListe.Search(1000))
    fmt.Println(maListe.Search(100))

    fmt.Printf("L'élément à l'indice 2 est %d\n", maListe.At(2))

    maListe.SetAt(2, 142)
    fmt.Printf("L'élément à l'indice 2 est %d\n", maListe.At(2))

    slice := maListe.ToSlice()
    fmt.Println(slice)

    maListe.Remove(142)
    fmt.Println(maListe)
}
```

Opérations sur une liste chaînée

Opérations sur le type Liste (fichier liste.go (suite))

```go
// IsEmpty teste si la liste est vide
func (ls Liste) IsEmpty() bool {
    return ls.tete == nil
}

// Length renvoie la longueur de la liste
func (ls Liste) Length() int {
    res := 0
    for curr := ls.tete ; curr != nil; curr = curr.suivant {
        res++
    }
    return res
}

// Cons ajoute elt en tête de liste
func (ls *Liste) Cons(elt int) {        // On modifie le récepteur => ça doit être un pointeur
    ls.tete = &cellule{elt, ls.tete}   // Notez le & devant cellule : ls.tete est un pointeur !
}

// Head renvoie le premier élément d'une liste non vide (erreur sinon)
func  (ls Liste) Head() (int, error) {
    if ls.IsEmpty() {
        return 0, fmt.Errorf("La liste est vide")
    }
    return ls.tete.valeur, nil
}
```

Opérations sur une liste chaînée (suite)

Opérations sur le type Liste (fichier liste.go (suite))

```
// Tail renvoie la liste privée de son premier élément
func (ls Liste) Tail() (Liste, error) {
    if ls.IsEmpty() {
        return Liste{}, fmt.Errorf("La liste est vide")
    }
    return Liste{ls.tete.suivant}, nil
}

// Append ajoute elt à la fin de la liste
func (ls *Liste) Append(elt int) {    // On modifie le récepteur => ça doit être un pointeur
    nouv := &cellule{elt, nil}
    if ls.IsEmpty() {        // La liste était vide => premier ajout (en tête)...
        ls.tete = nouv
    } else {                // On se positionne sur la dernière cellule
        curr := ls.tete
        for curr.suivant != nil {
            curr = curr.suivant
        }
        curr.suivant = nouv  // et on fait le lien avec la nouvelle cellule
    }
}
```

Remarques

Notez la technique Go pour indiquer qu'une fonction/méthode renvoie une erreur : on renvoie deux valeurs et la deuxième indique s'il y a eu ou non une erreur (s'il n'y a pas eu d'erreur, cette valeur vaut nil).

Opérations sur une liste chaînée (suite)

Travaux pratiques

À vous de jouer... (Search et Remove sont un peu plus complexes que les autres)

Évolution de la liste chaînée

- Le codage des fonctionnalités de la liste met en évidence le fait qu'on doit parcourir toute la liste pour faire certaines opérations (calcul de la longueur, ajout à la fin).
- D'ailleurs, plutôt que calculer le nombre d'éléments, pourquoi ne pas le stocker ?

Travaux pratiques

Proposer une nouvelle implémentation, plus efficace, tenant compte de ces remarques

La pile

Présentation

- Une pile est une structure de données avec un seul point d'accès : son *sommet*.
- On ajoute les données sur le sommet (on les « *empile* »).
- On ôte les données du sommet (on les « *dépile* »).
- C'est donc une structure *LIFO* (*Last In First Out*) : le dernier élément ajouté sera le premier traité...
- Cette structure est très utilisée en informatique : les variables locales et les résultats de fonctions, par exemple, sont créés sur une pile.

Remarque

Pensez à une pile d'assiettes, ou à une pile de documents : seul l'élément du dessus est visible...

Opérations sur une pile

Outre les opérations *Empiler* et *Depiler* déjà mentionnées, une pile dispose également des opérations suivantes :

- *CreerPile* crée une pile vide.
- *EstVide* teste si une pile est vide.
- *Sommet* renvoie la valeur située au sommet.
- On peut également lui ajouter des fonctions de conversion pour, par exemple, créer une pile à partir d'une tranche (*FromSlice*) et obtenir une tranche à partir d'une pile (*ToSlice*). En ce cas, pas besoin de *CreerPile*...

Remarque

- *Les opérations Depiler et Sommet doivent gérer le fait que la pile peut être vide...*
- *Il est souvent pratique d'écrire Depiler de sorte qu'elle renvoie l'élément supprimé de la Pile.*

Implémentation à partir d'une tranche

En Go, les tranches peuvent être gérées comme des piles :

- Il suffit de considérer que le dernier élément est le sommet (puisqu'on dispose déjà de la fonction *append* sur les tranches et qu'elle ajoute à la fin)...
- Pour la suppression, il faudra donc ôter le dernier élément et réaffectant la tranche sans celui-ci.
- Pour masquer tous ces détails, on crée un nouveau type *Pile* qui *encapsule* la tranche contenant les éléments.

Nouveau type Pile défini à partir d'une tranche

```
package pileslice

type Pile struct {
        corps []int
    }
```

Opérations sur la pile

TP...

Utilisation du package pileslice

Exemple d'utilisation des opérations sur la pile

```
package main

import (
    "fmt"
    "piles/pileslice"
)

func main() {
    pile := pileslice.FromSlice(1, 10, 100)
    pile.Empiler(1000)
    fmt.Println(pile.ToSlice())        // [1 10 100 1000]

    for !pile.EstVide() {
        top, _ := pile.Depiler()
        fmt.Println(top)             // Affiche de 1000 à 1
    }

    top, err := pile.Depiler();
    if err != nil {
        fmt.Println(err)
    } else {                         // Affiche "Pile vide"
        fmt.Println(top)
    }
}
```

Implémentation à partir d'une liste chaînée

Ici, le sommet pointera vers la première cellule de la liste chaînée :

Nouveau type Pile défini à partir d'une liste chaînée

```
package pilechainee

import "liste"

type Pile struct {
    corps liste.Liste
}
```

Opérations sur la pile

TP...

Utilisation du package pilechainee

Exemple d'utilisation des opérations sur la pile

```
package main

import (
    "fmt"
    "piles/pilechainee"
)

func main() {
    pile := pilechainee.FromSlice(1, 10, 100)
    pile.Empiler(1000)
    fmt.Println(pile.ToSlice())       // [1000 100 10 1]

    for !pile.EstVide() {
        top, _ := pile.Depiler()
        fmt.Println(top)              // Affiche de 1000 à 1
    }

    if top, err := pile.Depiler(); err != nil {
        fmt.Println(err)
    } else {                          // Affiche "Pile vide"
        fmt.Println(top)
    }
}
```

Opérations sur la pile

Pourquoi l'ordre de FromSlice est-il inversé par rapport à l'implémentation précédente ?

La file d'attente

Présentation

- Une file d'attente est une structure de données avec deux points d'accès : une *tête* et une *queue*.
- On ajoute les données dans la queue de la file (d'où l'expression « faire la queue »).
- On ôte les données depuis la tête.
- C'est donc une structure *FIFO* (*First In First Out*) : le premier élément ajouté sera le premier traité...
- Cette structure est très utilisée en informatique : file d'attente de processus, de travaux d'impression, etc.

Remarque

Pensez à une file d'attente à une caisse de supermarché...

Opérations sur une file

Outre les opérations *Oter* et *Ajouter* déjà mentionnées, une file dispose également des opérations :

- *CreerFile* crée une file vide.
- *EstVide* teste si une file est vide.
- *NbElts* renvoie le nombre d'éléments dans la file d'attente.
- *Tete* renvoie la valeur en tête de file.
- On peut également lui ajouter des fonctions de conversion pour, par exemple, créer une file à partir d'une tranche (*FromSlice*) et obtenir une liste à partir d'une file (*ToSlice*).

Remarque

- *Les opérations Tete et Oter doivent gérer le cas où la file est vide...*
- *Comme pour la pile, la présence de FromSlice remplace celle de CreerFile.*

Implémentation à partir d'une tranche

En Go, les tranches peuvent être gérées comme des files :

- La tête de la file sera l'élément à l'indice 0.
- On mémorisera le nombre d'éléments de la file pour ne pas avoir à le recalculer à chaque fois.

Nouveau type File défini à partir d'une tranche

```
package fileslice

type File struct {
    corps []int
    nbElts int
}
```

Opérations sur la file

TP...

Utilisation du package fileslice

Exemple d'utilisation des opérations sur la file

```
package main

import (
    "files/fileslice"
    "fmt"
)

func main() {
    file := fileslice.FromSlice(1, 10, 100)
    file.Ajouter(1000)

    fmt.Println(file.ToSlice())     // [1 10 100 1000]

    for !file.EstVide() {
        elt, _ := file.Oter()
        fmt.Println(elt)            // Affiche de 1 à 1000
    }

    if elt, err := file.Oter(); err != nil {
        fmt.Println(err)
    } else {                        // Affiche "La file est vide"
        fmt.Println(elt)
    }
}
```

Implémentation à partir d'une liste chaînée

- La tête pointera vers la première cellule de la liste chaînée (comme pour la pile).
- Pour optimiser l'ajout en queue de file, on mémorisera aussi l'adresse de la dernière cellule de la liste (donc on utilisera la version « améliorée » de *ListeChainee*...)

Nouveau type File défini à partir d'une liste chaînée

```
package filechainee
import "liste/liste2"

type File struct {
    corps liste2.ListeChainee
    nbElts int
}
```

Opérations sur la file

TP...

Utilisation du package filechainee

Exemple d'utilisation des opérations sur la file

```
package main

import (
    "files/filechainee"
    "fmt"
)

func main() {
    file := filechainee.FromSlice(1, 10, 100)
    file.Ajouter(1000)

    fmt.Println(file.ToSlice())     // [1 10 100 1000]

    for !file.EstVide() {
        elt, _ := file.Oter()
        fmt.Println(elt)            // Affiche de 1 à 1000
    }

    if elt, err := file.Oter(); err != nil {
        fmt.Println(err)
    } else {                        // Affiche "La file est vide"
        fmt.Println(elt)
    }
}
```

Arbres binaires de recherche

Présentation

- Un *arbre binaire* est constitué de *nœuds*. Ce sont les nœuds qui portent les valeurs.
- Chaque nœud peut avoir au maximum deux *fils*. Si un nœud n'a aucun fils, il est appelé « nœud terminal » ou « *feuille* ».
- Chaque nœud a un nœud père, sauf la *racine* de l'arbre.
- Un arbre binaire est donc une structure récursive : il est soit vide, soit formé d'un nœud et de deux sous-arbres binaires, appelés traditionnellement SAG (« sous-arbre gauche ») et SAD (« sous-arbre droit »).

Présentation

- Un *arbre binaire de recherche* (ABR) est un arbre binaire dans lequel les valeurs sont placées afin d'optimiser les recherches.
- Si la valeur insérée est inférieure ou égale à la valeur de la racine, on insère (récursivement) cette valeur dans le SAG.
- Si la valeur est strictement supérieure à la valeur de la racine, on insère (récursivement) cette valeur dans le SAD.
- Voir https ://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche

Remarque

Si l'arbre est « équilibré », une recherche s'effectue en $\log n$ *(ou en* n *dans le pire des cas).*

Implémentation du type Noeud

Un ABR est donc entièrement défini par son *Noeud* racine :

Type Noeud en Go

```
package abr

import "fmt"

type noeud struct          // Non exporté => minuscule
    data     int
    sag, sad *noeud
}

func (node noeud) String() string {
    return fmt.Sprintf("%d", node.data)
}
```

Insertion d'une valeur dans un ABR

L'insertion d'une valeur reprend l'algorithme décrit dans le transparent précédent :

Insertion d'une valeur dans un ABR

```
// insert insère la valeur data dans l'arbre dont la racine est node
func (node *noeud) insert(data int) {
    if data <= node.data {
        if node.sag == nil {
            node.sag = &noeud{data, nil, nil}
        } else {
            node.sag.insert(data)
        }
    } else {
        if node.sad == nil {
            node.sad = &noeud{data, nil, nil}
        } else {
            node.sad.insert(data)
        }
    }
}
```

Affichage d'un arbre

L'affichage consiste à parcourir l'arbre en profondeur d'abord, de gauche à droite, afin d'obtenir les valeurs triées :

Affichage d'un ABR

```
func (node *noeud) PrintAbr() {
    if node == nil {
        return
    }
    node.sag.PrintAbr()
    fmt.Printf("%s ", node)
    node.sad.PrintAbr()
}
```

Recherche dans un ABR

La recherche suit la même logique que l'insertion :

Recherche dans un ABR

```
func (node *noeud) contains(data int) bool {
    if node == nil {
        return false
    } else if data == node.data {
        return true
    } else if data < node.data {
        return node.sag.contains(data)
    } else {
        return node.sad.contains(data)
    }
}
```

Remarque

Le problème de cette méthode est qu'elle est peu utile puisqu'elle ne fait que renseigner si, oui ou non, la donnée est dans l'arbre...

Recherche dans un ABR (bis)

Au lieu de tester la présence d'une valeur, on peut vouloir renvoyer le nœud qui la contient, mais il faut alors gérer le cas où la valeur n'est pas présente :

Recherche dans un ABR

```
func (node *noeud) lookup(data int, parent *noeud) (*noeud, *noeud) {
    if node == nil {
        return nil, nil
    } else if data < node.data {
        if node.sag == nil {
            return nil, nil
        } else {
            return node.sag.lookup(data, node)
        }
    } else if data > node.data {
        if node.sad == nil {
            return nil, nil
        } else {
            return node.sad.lookup(data, node)
        }
    } else {
        return node, parent
    }
}
```

Recherche dans un ABR : remarques

- On renvoie aussi le nœud père du nœud recherché car, si l'on fait une recherche, on voudra surement ensuite traiter le nœud trouvé et il faut en ce cas connaître son père...
- Comme on veut pouvoir renvoyer *nil, nil* si l'élément n'est pas dans l'arbre, la valeur de retour doit être de type *(*noeud, *noeud)*, et non *(noeud, noeud)*...

Autres fonctions utiles

Taille et profondeur d'un arbre

```
func (node *noeud) size() int {
    if node == nil {
        return 0
    } else {
        return node.sag.size() + 1 + node.sad.size()
    }
}

func (node *noeud) maxDepth() int {
    if node == nil {
        return 0
    } else {
        lDepth := node.sad.size()
        rDepth := node.sad.size()
        if lDepth > rDepth {
            return lDepth + 1
        } else {
            return rDepth + 1
        }
    }
}
```