

VII Structures récursives

Collections de données de taille indéterminée

VII.1 Types Pile et File



VII.1.1 La Pile

Une *Pile* est un type permettant de représenter un ensemble de données, du même `type T`, à accès séquentiel et écriture non destructive.

L'accès aux données se fait dans l'ordre inverse duquel elles ont été insérées.

Définition : *Pile*

On appelle Pile, un ensemble formé d'un nombre variable de données sur lequel on effectue les opérations suivantes :

- ❑ ajouter une nouvelle donnée ;
- ❑ consulter la dernière donnée ajoutée et non supprimée depuis ;
- ❑ supprimer la dernière donnée ajoutée et non supprimée depuis ;
- ❑ savoir si l'ensemble est vide ou non ;

Fonctionnalités

`init: Int → StackT // crée une pile (vide)`
`top: StackT → T | Vide // retourne le sommet de la pile`
`pop: StackT → StackT x T // retire le sommet de la pile`
`push: StackT x T → StackT // ajoute un nouvel élément`
`isEmpty: StackT → Bool // vérifie si la pile est vide`
`isFull: StackT → Bool // vérifie si la pile est pleine`
`capacity: StackT → Int // nombre maximum d'éléments`
`count: StackT → Int // nombre d'éléments dans la pile`

Caractéristiques (features)

S1: $\text{isEmpty}(\text{init}(n)) == \text{true}$

S2: $\text{count}(\text{init}(n)) == 0$

S3: $\text{top}(\text{push}(p, t)) == t$

S4: $\text{top}(p) == \text{Vide} \iff \text{isEmpty}(p)$

S5: $(p_2, t_2) = \text{pop}(\text{push}(p_1, t_1)) \iff (p_2 == p_1) \ \&\& \ (t_2 == t_1)$

S6: $(p_2, t) = \text{pop}(p_1) \Rightarrow \text{push}(p_2, t) == p_1$

S7: $\text{count}(\text{push}(p, t)) == \text{count}(p) + 1$

S8: $(p_2, t) = \text{pop}(p_1) \Rightarrow \text{count}(p_2) == \text{count}(p_1) - 1$

S9: $\text{pop}(p) \Rightarrow \neg \text{isEmpty}(p)$

S10: $\text{isFull}(p) \iff \text{count}(p) == \text{capacity}(p)$

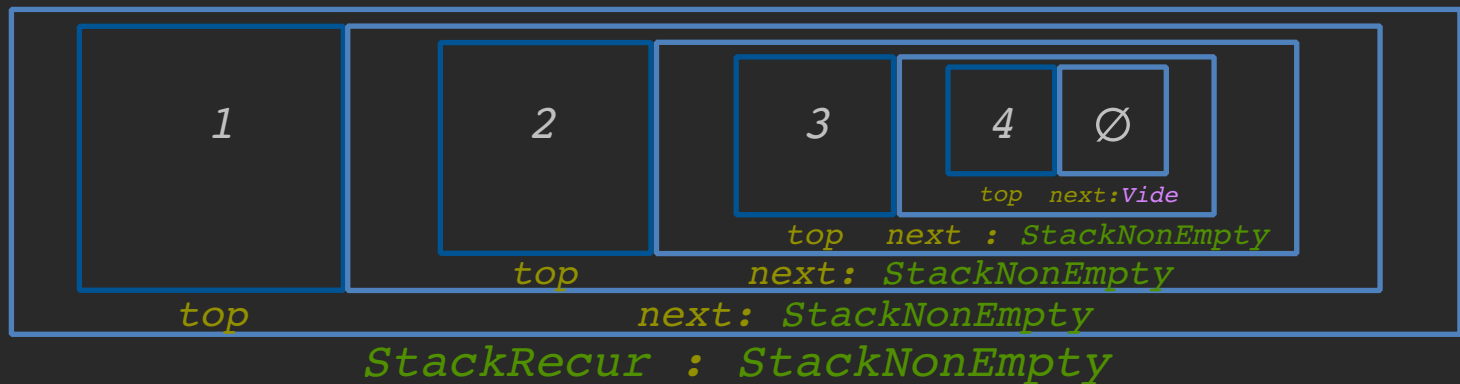
S11: $\text{isEmpty}(p) \iff \text{count}(p) == 0$

S12: $\text{capacity}(\text{init}(n)) == n$

S13: $\text{push}(p, t) \Rightarrow \neg \text{isFull}(p)$

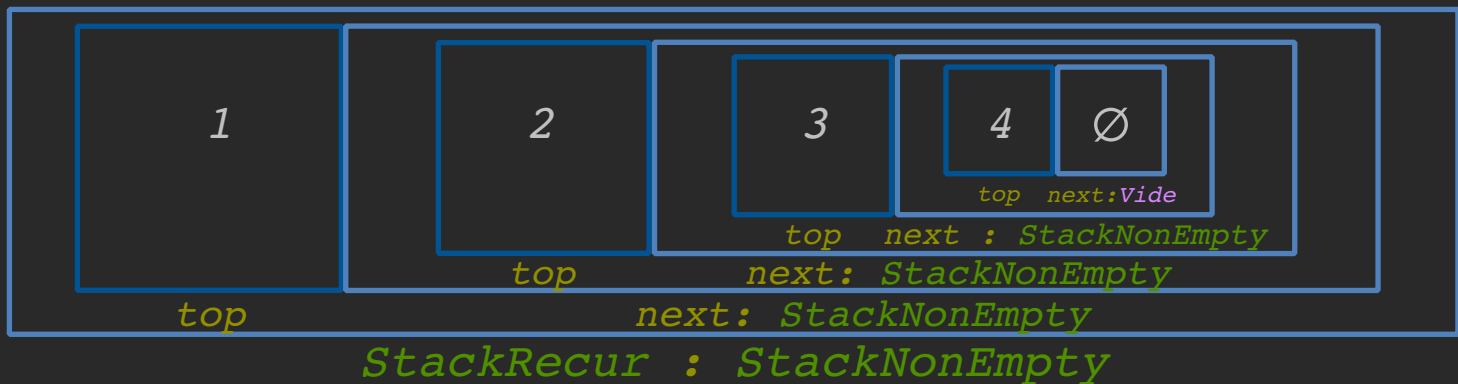
Description logique

Une pile peut être définie récursivement : en effet une pile n'est rien d'autre qu'une pile sur laquelle on rajoute un nouvelle élément par dessus le sommet de la pile ; ce nouvel élément devient le nouveau sommet de pile



Description logique

Une pile peut être définie récursivement : en effet une pile n'est rien d'autre qu'une pile sur laquelle on rajoute un nouvelle élément par dessus le sommet de la pile ; ce nouvel élément devient le nouveau sommet de pile



`StackRecur: (Vide | StackNonEmpty)`

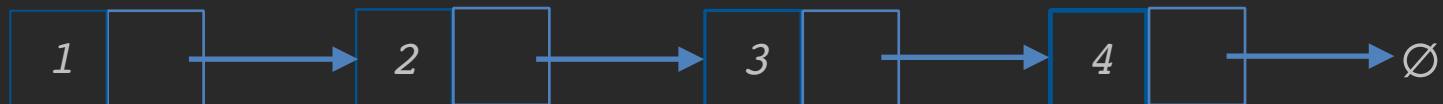
`StackNonEmpty: (top: T, next: StackRecur)`

Description logique

Une pile peut être définie récursivement : en effet une pile n'est rien d'autre qu'une pile sur laquelle on rajoute un nouvelle élément par dessus le sommet de la pile ; ce nouvel élément devient le nouveau sommet de pile

`StackRecur: (Vide | StackNonEmpty)`

`StackNonEmpty: (top: T, next: StackRecur)`




```
func init() → StackRecur  
  return new StackRecur(Vide)  
endfunc
```

```
func top(stack: StackRecur) → T  
  if isEmpty(stack) then ERREUR  
  else return stack.val  
endfunc
```

```
func isEmpty(stack: StackRecur) → Bool  
  return stack == Vide  
endfunc
```

```
func push(stack: StackRecur, t: T) → StackRecur  
  return new StackNonEmpty(val: t, next: stack)  
endfunc
```

```
func pop(stack: StackRecur) → (StackRecur, T)  
  if isEmpty(stack) then ERREUR  
  else return (stack.next, stack.val)  
endfunc
```

Structure purement récursive ?

Le problème principal d'une structure récursive est qu'elle est dévoilée à son utilisateur : celui-ci utilise donc une structure de données plutôt qu'un type abstrait.

En outre, il est impossible d'avoir des propriétés globales (comme `capacity`) sans la répéter dans la structure `StackNonEmpty` et encore moins de manipuler la collection dans sa globalité

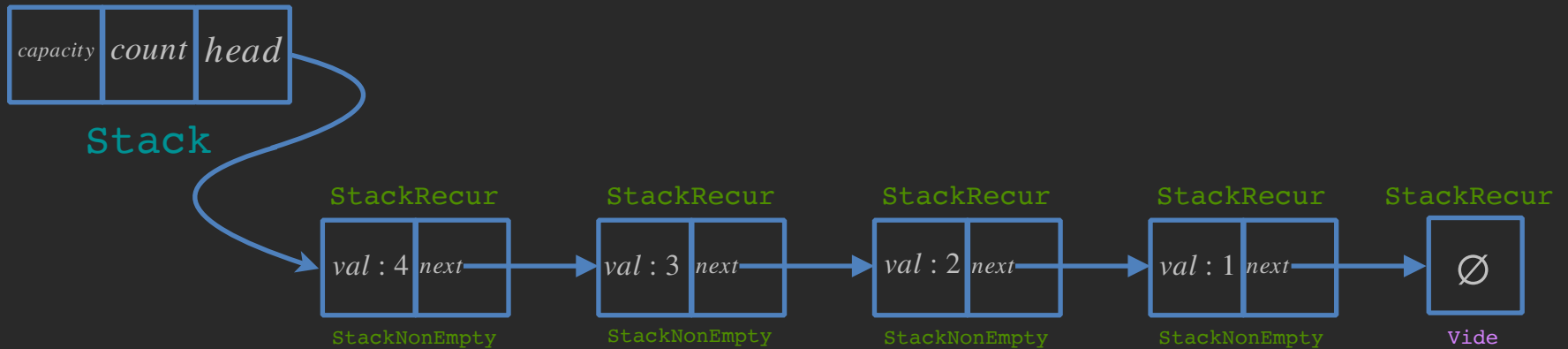
Avec une telle structure, *accéder à un élément revient à accéder à tout* ou partie de la collection donnant ainsi la possibilité de « casser celle-ci » ; ex: la fonction `top` permet d'accéder au premier élément de la sous-liste représentée par l'élément passé en paramètre.

⇒ il faut « cacher » la structure récursive à l'aide d'une structure principale : on parle de tête (*head*) de

```

Stack: (capacity: Int, count: Int, head: StackRecur)
StackRecur: (Vide | StackNonEmpty)
StackNonEmpty: (val: T, next: StackRecur)

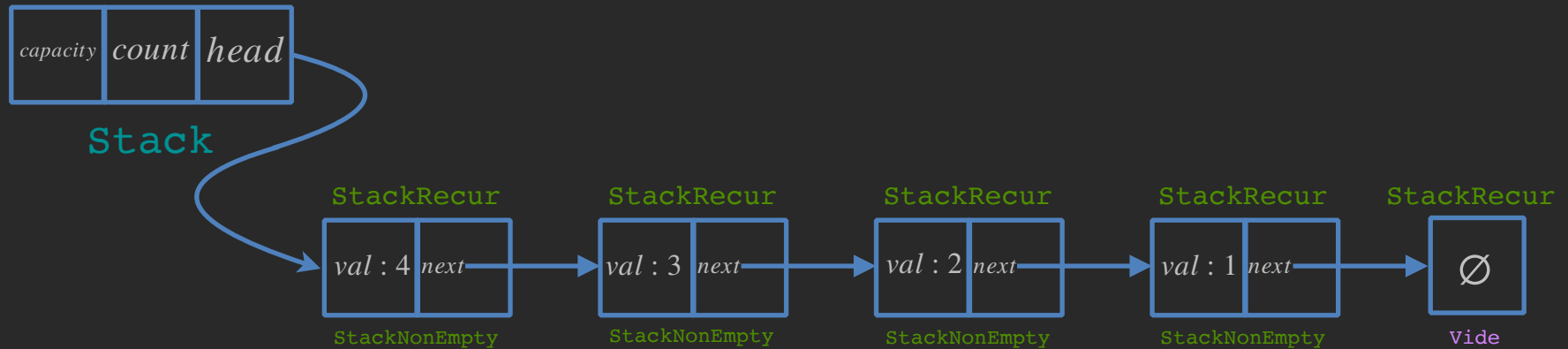
```



```

Stack: (capacity: Int, count: Int, head: StackRecur)
StackRecur: (Vide | StackNonEmpty)
StackNonEmpty: (val: T, next: StackRecur)

```



On garde les algorithmes récursifs pour la partie récursive des types **StackRecur**, **StackNonEmpty** et le type **Stack** implémente les fonctions de la spécification fonctionnelle en utilisant ceux de **StackRecur** et **StackNonEmpty**

```
func init(capacity: Int) -> Stack
  capacity = capacity
  stack = StackRecur(Vide)
  count = 0
endfunc
```

```
func push(stack: Stack, t: T) -> Stack
  if isFull(stack) then ERREUR
  stack.head = StackNonEmpty(t, stack.head)
  stack.count += 1
  return stack
endfunc
```

```
func pop(stack: Stack) -> (Stack, T)
  if isEmpty(stack) then ERREUR
  let ret : T = stack.head.val
  stack.head = stack.head.next // ⚠ penser à libérer la mémoire si nécessaire
  stack.count -= 1
  return (stack, ret)
endfunc
```

```
func top(stack: Stack) → T
  if isEmpty(stack) then ERREUR
  else return top(stack.head)
endfunc
```

```
func isEmpty(stack: Stack) → Bool
  return isEmpty(stack.head)
endfunc
```

```
func isFull(stack: Stack) → Bool
  return stack.count == stack.capacity
endfunc
```

```
func capacity(stack: Stack) → Int
  return stack.capacity
endfunc
```

```
func count(stack: Stack) → Int
  return stack.count
endfunc
```

Représentation physique

La représentation physique dépend du langage choisi. Si l'on choisit la description logique récursive, on doit résoudre le problème de la représentation de Vide.

En Swift, on utilisera donc un type interne défini par une class et le mécanisme d'optionnel pour signifier que la valeur peut-être Vide.

Il suffit alors d'écrire les algorithmes en les traduisant directement en Swift ...

Représentation physique

La représentation physique dépend du langage choisi. Si l'on choisit la description logique récursive, on utilisera le mécanisme de type optionnel pour signifier que la valeur peut être *vide*.

Représentation physique

La représentation physique dépend du langage choisi. Si l'on choisit la description logique récursive, on utilisera le mécanisme de type optionnel pour signifier que la valeur peut être `vide`.

En revanche, il n'est pas possible de faire des types valeur récursif, par exemple il est impossible de faire :

```
struct ListeNV {  
    var val : Int  
    var suivant : ListeNV?  
}  
typealias Liste = ListeNV?
```


TypeAlias permet de définir le type Liste comme étant en réalité un optionnel du type Nœud

Représentation physique

La représentation physique dépend du langage choisi. Si l'on choisit la description logique récursive, on utilisera le mécanisme de type optionnel pour signifier que la valeur peut être *vide*.

En revanche, il n'est pas possible de faire des types valeur récursif, par exemple il est impossible de faire :

```
struct ListeNV {  
    var v : Int  
    var suivant : ListeNV?  
}  
typealias Liste = ListeNV?
```



TypeAlias permet de définir le type Liste comme étant en réalité un optionnel du type Nœud

En Swift, les structs sont des *types valeurs*. Or, pour pouvoir implémenter à l'aide d'optionnels l'alternative pour des types récurifs, il est nécessaire de disposer de *type référence* :

- ❑ un *type référence*, est un type pour lequel chaque donnée (variable) de ce type est considérée comme une référence sur ces données : une variable de ce type indique en fait l'adresse de la donnée ; c'est un *pointeur*
- ❑ l'affectation ne copie donc pas les données mais la référence, les 2 variables référencent alors les mêmes données

En swift, un type référence est introduit par le mot clef *Class*

Il définit un objet, avec ses mécanismes sous-jacents, mais nous l'utiliseront ici uniquement en tant que type référence.

```

/// Pile de données
protocol TStack {
    /// ckeck if stack is empty
    /// - returns: `true` if stack is empty
    var isEmpty : Bool { get }
    /// get the value on the top of the stack
    /// - returns: `nil` if the stack is empty, else return value on
    /// top of the stack
    var capacity: Int { get }
    var count: Int { get }
    var top : Int? { get }
    /// remove top of the stack
    /// - precondition: stack must not be empty
    /// - returns: new stack with top removes
    /// - throws: `fatalerror` if stack was empty
    mutating func pop() -> Self
    /// push a new value on top of the stack
    /// - Parameter val: value to be push on top of stack
    /// - returns: the new stack with value on top
    mutating func push(_ val: Int) -> Self
}

```

```

struct Stack : TStack{
    fileprivate class StackNE{ // utilisation d'un type référence
        var val : Int
        var next : StackNE?
        init(val: Int, next: StackNE? = nil){
            self.val = val
            self.next = next
        }
        var count : Int {
            guard let next = self.next else { return 1 }
            return 1 + next.count
        }
    }
    fileprivate var head : StackNE?
    private(set) var capacity : Int
    public init(capacity: Int){
        self.head = nil
        self.capacity = capacity
    }
    public var isEmpty : Bool{ return self.head == nil }
}

```

```

public var count {
    guard let head = self.head else { return 0 }
    return head.count
}
public var top : Int?{
    guard let head = self.head else { return nil }
    return head.val
}
mutating public func push(_ val: Int){
    guard self.count < self.capacity else { fatalError("Push on
full stack") }
    self.head = StackNE(val: val, next: self.head)
    return self
}
mutating public func pop(){
    guard let head = self.head else { fatalError("ERROR pop on
empty stack!") }
    self.head = head.next
    return self
}
}

```

VII.1.2 La File

Une **File** (Queue) est un type permettant de représenter un ensemble de données, du même type T , à accès séquentiel et écriture non destructive.

L'accès aux données se fait dans l'ordre dans lequel elles ont été insérées.

Définition : **File**

On appelle **File**, un ensemble formé d'un nombre variable de données sur lequel on effectue les opérations suivantes :

- ❑ ajouter une nouvelle donnée ;
- ❑ consulter la première donnée ajoutée et non supprimée depuis ;
- ❑ supprimer la première donnée ajoutée et non supprimée depuis ;
- ❑ savoir si l'ensemble est vide ou non ;

Spécification fonctionnelle

Fonctionnalités

`init`: $\rightarrow \text{QueueT}$ //crée une file (vide)
`first`: $\text{QueueT} \rightarrow T \mid \text{Vide}$ //retourne le premier de la file
`deQueue`: $\text{QueueT} \rightarrow \text{QueueT} \times T$ //retire le premier de la file
`enQueue`: $\text{QueueT} \times T \rightarrow \text{QueueT}$ //ajoute un nouvel élément à la fin de la file
`isEmpty`: $\text{QueueT} \rightarrow \text{Bool}$ //vérifie si la File est vide
`isFull`: $\text{QueueT} \rightarrow \text{Bool}$ // vérifie si la File est pleine
`capacity`: $\text{QueueT} \rightarrow \text{Int}$ // nombre maximum d'éléments
`count`: $\text{QueueT} \rightarrow \text{Int}$ // nombre d'éléments dans la File

Spécification fonctionnelle

Fonctionnalités

Q1: $\text{isEmpty}(\text{init}()) == \text{True}$

Q2: $\text{count}(\text{init}()) == 0$

Q3: $\text{first}(q) == \text{Vide} \Rightarrow \text{isEmpty}(q)$

Q4: $\text{deQueue}^n(\text{deQueue}^n(q, t_{k_n})) == \text{Vide}$

Q5: $\text{first}(\text{enqueue}(\dots(\text{enqueue}(q, t_0), t_1), \dots, t_n)) == t_0$

Q6: $\text{deQueue}(q) \Rightarrow \neg \text{isEmpty}(q)$

Q7: $\text{deQueue}(\text{enqueue}^n(\text{init}(), t_i)) == \text{enqueue}^{n-1}(\text{init}(), t_i)$

Q8: $\text{count}(\text{enqueue}(q, t)) == \text{count}(q) + 1$

Q9: $(q_2, t) = \text{deQueue}(q_1) \Rightarrow \text{count}(q_2) == \text{count}(q_1) - 1$

Q10: $\text{isFull}(q) \iff \text{count}(q) == \text{capacity}(q)$

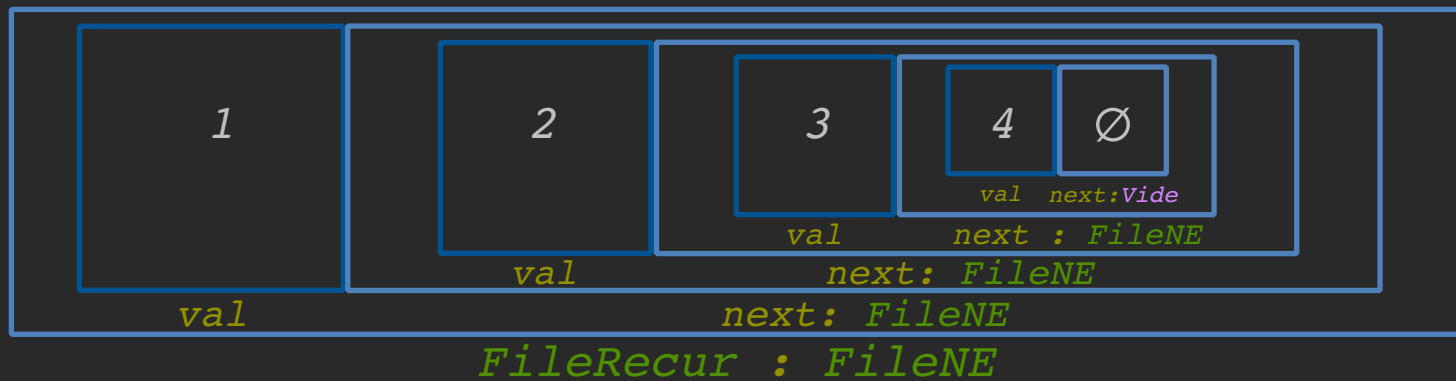
Q11: $\text{isEmpty}(q) \iff \text{count}(q) == 0$

Q12: $\text{capacity}(\text{init}(n)) == n$

Q13: $\text{enqueue}(q, t) \Rightarrow \neg \text{isFull}(q)$

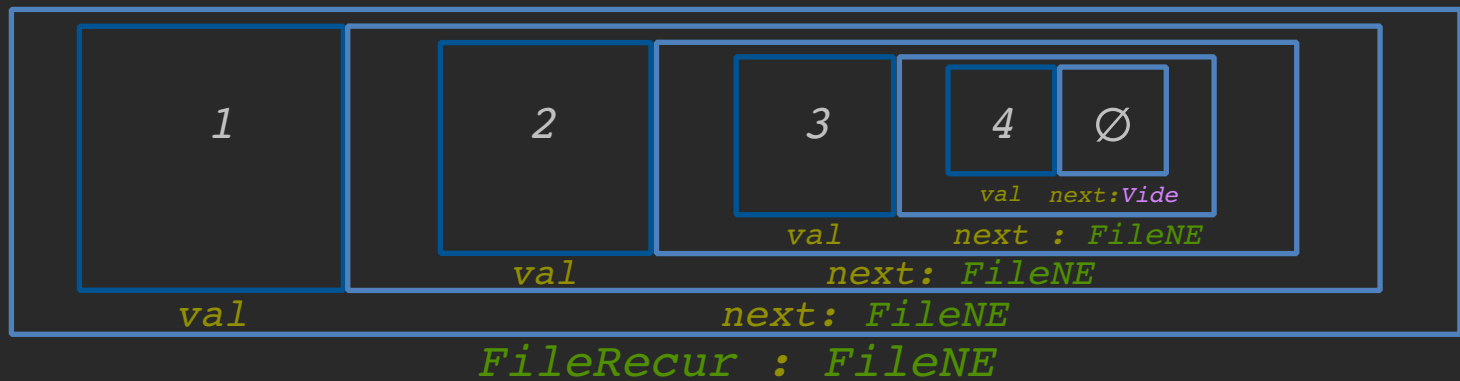
Description logique

Une file est une structure récursive : en effet une file n'est rien d'autre qu'une file avec un élément et contenant une autre file :



Description logique

Une file est une structure récursive : en effet une file n'est rien d'autre qu'une file avec un élément et contenant une autre file :



```
FileRecur: (Vide | FileNE)  
FileNE: (val: T, next: FileRecur)
```

Description logique

Une file est une structure récursive : en effet une file n'est rien d'autre qu'une file avec un élément et contenant une autre file :

```
FileRecur: (Vide | FileNE)  
FileNE: (val: T, next: FileRecur)
```



```
func init() → FileRecur  
  return new FileRecur(Vide)  
endfunc
```

```
func first(f: FileRecur) → T  
  if isEmpty(f) then return Vide  
  else               return f.val  
endfunc
```

```
func isEmpty(f: FileRecur) → Bool  
  return f == Vide  
endfunc
```

```

func enqueue(f: FileRecur, t: T) → FileRecur
  if isEmpty(f) then return new FileNE(t, f)
  else {
    f.next = enqueue(f.next, t) // ⚠ chaînage arrière
    return self
  }
endfunc

```

```

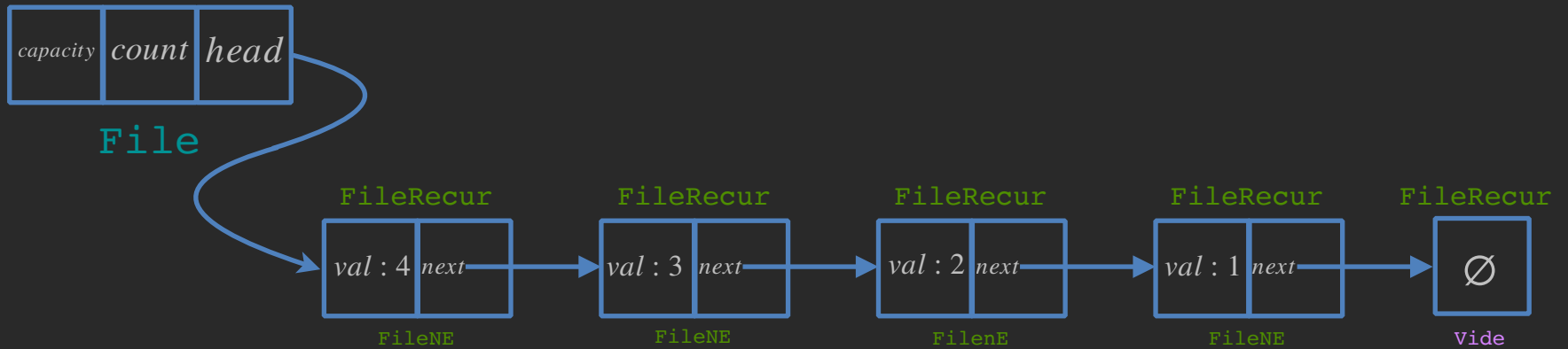
func dequeue(f: FileRecur) → (FileRecur, T)
  if isEmpty(f) then ERREUR
  else
    let ret = f.val
    return (f.next, f.val) // ⚠ fuite mémoire
  endif
endfunc

```

```

File: (capacity: Int, count: Int, head: FileRecur)
FileRecur: (Vide | FileNE)
FileNE: (val: T, next: FileRecur)

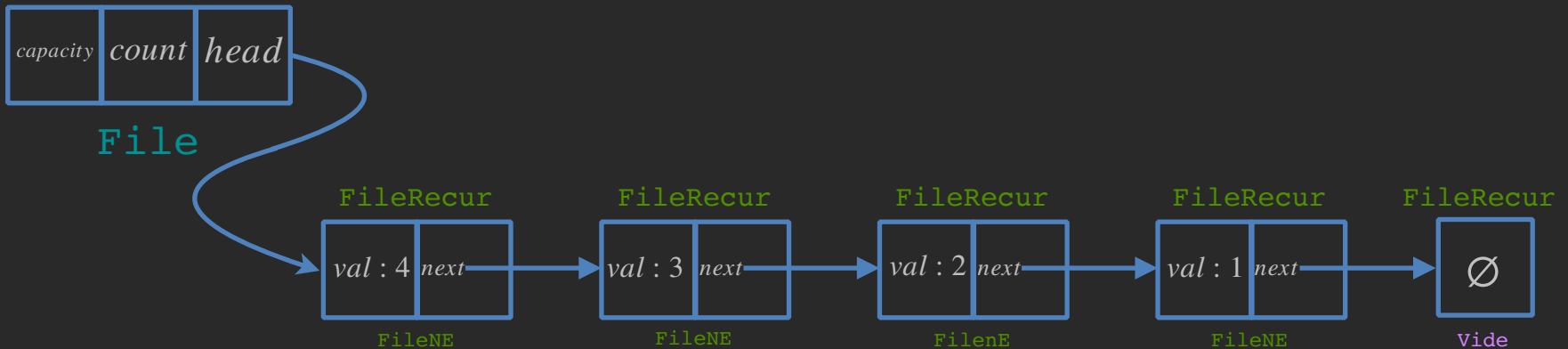
```



```

File: (capacity: Int, count: Int, head: FileRecur)
FileRecur: (Vide | FileNE)
FileNE: (val: T, next: FileRecur)

```



On garde les algorithmes récursifs pour la partie récursive des types **FileRecur**, **FileNE** et le type **File** implémente les fonctions de la spécification fonctionnelle en utilisant ceux de **FileRecur** et **FileNE**

Représentation physique

La représentation physique de `FileT` en Swift reprend les mêmes principes que pour `PileT`:

```
protocol TQueue {
    /// check if queue is empty
    /// - returns: `true` if stack is empty
    var isEmpty : Bool { get }
    /// get the first value in the queue
    /// - returns: `nil` if the queue is empty, else return first value
    var first : Int? { get }
    var capacity: Int { get }
    var count: Int { get }
    /// remove first value of the queue
    /// - precondition: queue must not be empty
    /// - returns: new queue with first value removed
    /// - throws: `fatalerror` if queue was empty
    mutating func dequeue() -> Self
    /// put a new value at the end of the queue
    /// - Parameter val: value to be enqueue
    /// - returns: the new queue with value at the end of the queue
    mutating func enqueue(_ val: Int) -> Self
}
```

```

struct Queue : TQueue{
    private class QueueNE{
        var val : Int
        var next : QueueNE?
        init(val: Int){
            self.val = val
            self.next = nil
        }
        var count : Int {
            guard let next = self.next else { return 1 }
            return 1 + next.count
        }
    }
    private var head : QueueNE?
    private(set) var capacity : Int

    public init(capacity: Int){
        self.capacity = capacity
        self.queue = nil
    }
    var isEmpty : Bool{ return self.head == nil }

```

```

public var count {
    guard let head = self.head else { return 0 }
    return head.count
}
var first : Int?{
    guard let head = self.head else { return nil }
    return head.val
}
private func enqueueRecur(q: QueueNE?, val: Int) -> QueueNe{
    guard let q = q else { return QueueNE(val: t) }
    q.next = enqueueRecur(q: q.next, val: t)
    return q
}
func enqueue(_ val: Int){
    self.head = enqueueRecur(q: self.head, val: val)
}
public func dequeue(){
    guard let head = self.head else {
        fatalError("ERROR dequeue on empty queue!")
    }
    self.head = head.next
    return self
}
}

```

VII.2 La Liste

VII.2.1 Listes

Une *Liste* est un type permettant de représenter un ensemble de données, du même type T , à accès séquentiel et écriture non destructive.

L'accès aux données se fait séquentiellement et on peut insérer/supprimer un élément au début, en fin ou même au milieu de la liste

Définition : *Liste*

On appelle *Liste*, un ensemble formé d'un nombre variable de données sur lequel on effectue les opérations suivantes :

- ❑ ajouter une nouvelle donnée ;
- ❑ consulter une donnée, en particulier la première et la dernière
- ❑ supprimer une donnée, en particulier la première et la dernière
- ❑ savoir si l'ensemble est vide ou non ;

Spécification fonctionnelle

```
init: -> ListT // crée une liste (vide)
isEmpty: ListT -> Booleen
first: ListT -> T // retourne le premier de la liste
last: ListT -> T // retourne le dernier de la liste
// retourne le nombre d'éléments t dans la liste l
nb_occur: ListT x T -> int
// insère un élément en début de liste
insert_first: ListT x T -> ListT
// insère un élément en fin de liste
insert_last: ListT x T -> ListT
// supprime le premier (resp dernier) élément de la liste
remove_first: ListT -> ListT
remove_last: ListT -> ListT
// supprime le premier élément t de la liste
remove_elt: ListT x T -> ListT
```

Spécification fonctionnelle

Caractéristiques

L1: isEmpty(init())==True
L2: first(l) => !isEmpty(l)
L3: last(l) => !isEmpty(l)
L4: first(insert_first(l,t))==t
L5: last(insert_last(l,t))==t
L6: remove_first(init())==init()
L7: remove_first(insert_first(t,l))==l
L8: remove_last(init())==init()
L9: remove_last(insert_last(t,l))==l
L10: nb_occur(init(),t)=0
L11: remove_elt(l,t)=>nb_occur(l,t)>1
L12: nb_occur(remove_elt(l,t))==nb_occur(l)-1
L13: l2=insert_first(l,t1) =>
remove_elt(insert_first(t2,l2),l2)==insert_first(t2,l)

Description logique

Une liste peut être définie récursivement : en effet une liste n'est rien d'autre qu'une liste ayant un premier élément et contenant une autre liste :



Description logique

Une liste peut être définie récursivement : en effet une liste n'est rien d'autre qu'une liste ayant un premier élément et contenant une autre liste :

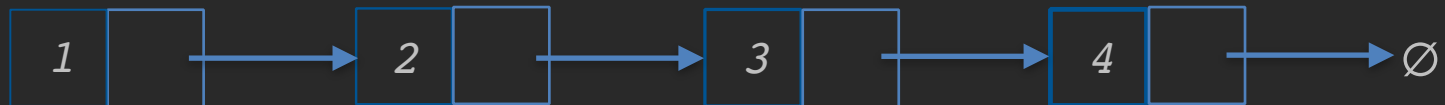


```
ListRecur: (Vide | ListNE)  
ListNE: (elt: T, next: ListRecur)
```

Description logique

Une liste peut être définie récursivement : en effet une liste n'est rien d'autre qu'une liste ayant un premier élément et contenant une autre liste :

```
ListRecur: (Vide | ListNE)  
ListNE: (elt: T, next: ListRecur)
```



```
func creer_liste() -> ListeRecur
  return new ListeRecur(Vide)
endfunc
```

```
func isEmpty(list: ListeRecur) -> Bool
  return list==Vide
endfunc
```

```
func first(list: ListeRecur) -> T
  if isEmpty(list) then
    ERREUR
  else
    return liste.elt
  endif
endfunc
```

```

func last(list: ListeRecur) -> T
  if isEmpty(list) then
    ERREUR
  else
    if isEmpty(liste.next) then
      return liste.elc
    else
      return last(liste.next)
    endif
  endif
endfunc

```

```

func insert_first(list: ListeRecur, t: T) -> ListeRecur
  return new ListeNE(t, list)
endfunc

```

```

func insert_last(list: ListeRecur, t: T) -> ListeRecur
  if isEmpty(list) then
    return new ListeNE(t,Vide)
  else
    liste.next = new ListeNE(t,liste.next)
    return list
  endif
endfunc

```

```

func nb_occur(list: ListeRecur, t: T) -> Int
  if isEmpty(list) then
    return 0
  else
    if first(list)==t then
      return nb_occur(liste.next,t)+1
    else
      return nb_occur(list.next,t)
    endif
  endif
endfunc

```

```
func remove_first(list: ListeRecur) -> ListeRecur
  if isEmpty(list) then
    return list
  else
    return liste.next
  endif
endfunc
```

```
func remove_last(list: ListeRecur) -> ListeRecur
  if isEmpty(list) then ERREUR
  else
    if isEmpty(liste.next) then
      return Vide
    else
      list.next = remove_last(list.next)
      return list
    endif
  endif
endfunc
```

Représentation physique

La représentation physique de `Liste` en Swift reprend les mêmes principes que pour `Pile` et `File` :

```
protocol TList {  
    /// check if list is empty  
    /// - returns: `true` if stack is empty  
    var isEmpty : Bool { get }  
    /// get the first value of the list  
    /// - returns: `nil` if the list is empty, else return first value  
    var first : Int? { get }  
    /// get last value of the list  
    /// - returns: `nil` if list is empty, else last value of the list  
    var last : Int? { get }  
    /// size of the list  
    /// - returns: number of elements in list  
    /// - postcondition: count : Int  $\geq$  0  
    var count : Int { get }
```

```

/// put a new value as first value of the list
/// - Parameter val: value to be added
/// - returns: the new list with value at the beginning of the list
@discardableResult mutating func add_first(_ val: Int) -> Self
/// put a new value as last value of the list
/// - Parameter val: value to be added
/// - returns: the new list with value at the end of the list
@discardableResult mutating func add_last(_ val: Int) -> Self
/// remove first value of the list
/// - precondition: list must not be empty
/// - returns: new list with first value removed
/// - throws: `fatalerror` if list was empty
@discardableResult mutating func remove_first() -> Self
/// remove last value of the list
/// - precondition: list must not be empty
/// - returns: new list with last value removed
/// - throws: `fatalerror` if list was empty
@discardableResult mutating func remove_last() -> Self
}

```



```

struct List : TList{
  fileprivate class ListNE{
    var val : Int
    var next : ListNE?
    init(val: Int, next: ListNE? = nil){
      self.val = val
      self.next = next
    }
  }
  fileprivate var head : ListNE?

  public init(){
    self.head = nil
  }
  public var isEmpty : Bool{
    return self.head == nil
  }
  public var first : Int?{
    guard let head = self.head else { return nil }
    return head.val
  }
}

```

```

// --- fonction privée récursive
private func last(ofListe l: ListNE) -> Int{
    guard let next = l.next else { return l.val }
    return last(ofListe: next)
}
// ---
// définition de last pour le type List
public var last : Int?{
    guard let llast = self.head else { return nil }
    return last(ofListe: llast)
}
@discardableResult
public func add_first(_ val: Int) -> Self{
    let node = ListNE(val: val, next: self.head)
    self.head = node
    return self
}

```

```

// --- fonction privée récursive
private func add_last(ofListe l: ListNE?, val: Int) -> ListNE{
    guard let l = l else { return ListNE(val: val) }
    l.next = add_last(ofListe: l.next, val: val)
    return l
}
// ---
// fonction add_last du type List
@discardableResult public func add_last(_ val: Int) -> Self{
    self.head = add_last(ofListe: self.head, val: val)
    return self
}

@discardableResult public func remove_first() -> Self{
    guard let head = self.head else {
        fatalError("ERROR remove_first on empty list!")
    }
    self.head = list.next
    return self
}

```

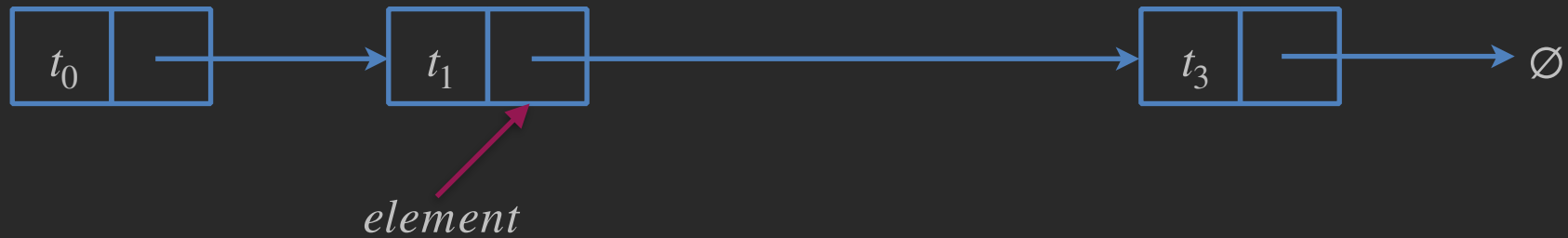
```

// --- fonction privée récursive
func remove_last(ofListe l: ListeNE) -> ListNE? {
    guard let next = self.next else { return nil }
    l.next = remove_last(ofListe: next)
    return l
}
// --- fin du bloc à placer dans la classe ListNE
// fonction remove_last du type List
@discardableResult public func remove_last() -> Self{
    guard let list = self.list else {
        fatalError("ERROR remove_last on empty list!")
    }
    self.list = remove_last(ofListe: list)
    return self
}

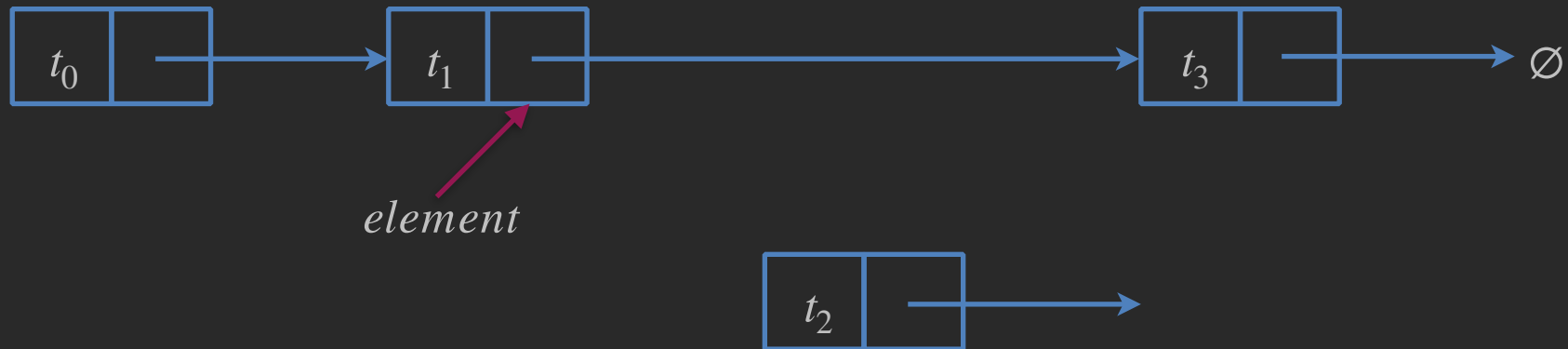
```

- ❑ Exercice : implémentez
 - la fonction nb_occur

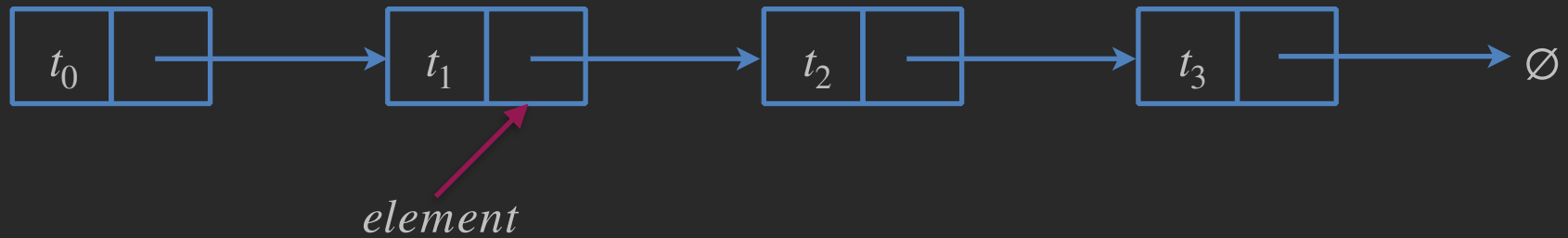
Exemple d'opération : insertion après



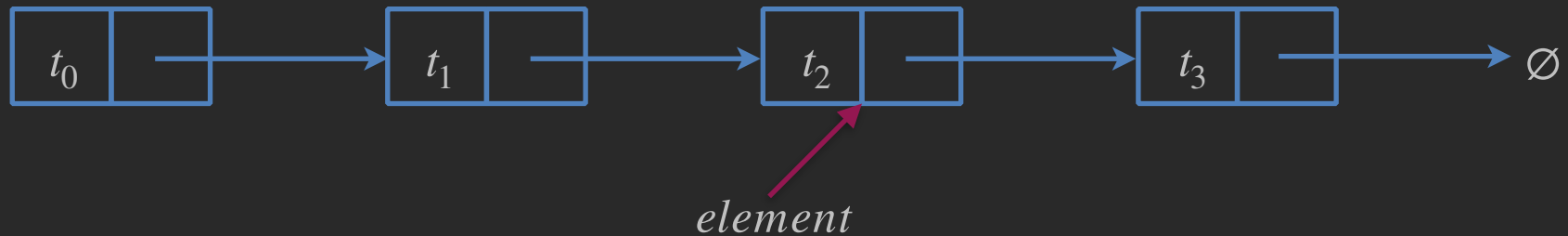
Exemple d'opération : insertion après



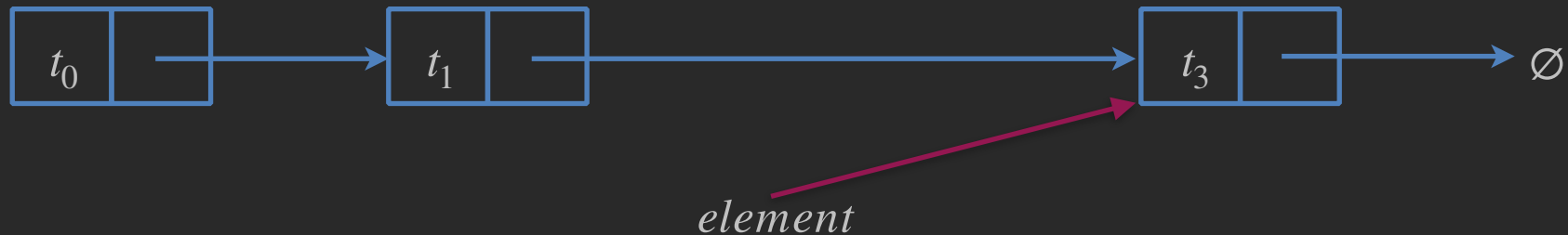
Exemple d'opération : insertion après



Exemple d'opération : suppression



Exemple d'opération : suppression



- Exercice : implémentez
 - la fonction `insert_after(node: ListNE)`
 - la fonction `remove(node: ListNE)`

VII.2.2 Les itérateurs

Une *Liste* permet donc de représenter un ensemble de données que l'on veut pouvoir accéder séquentiellement à partir de n'importe laquelle.

On voudrait alors, avant ou après un élément donné, ajouter un autre nouvel élément. On veut également pouvoir supprimer un élément.

Les opérations à rajouter au type liste sont les suivantes :

- ❑ parcourir la liste
- ❑ repérer un élément particulier
- ❑ ajouter avant ou après un élément particulier
- ❑ supprimer un élément particulier (ou après/avant cet élément)

Si on veut pouvoir désigner un élément particulier, par exemple en cas de multiple occurrences, deux solutions :

- ❑ pouvoir désigner spécifiquement un `ListNE` et donc connaître la structure !
- ❑ parcourir la liste et pouvoir désigner un élément courant

=> nécessité d'abstraire le parcours (à la « `for in` ») et de disposer d'un élément courant : concept d'*itérateur*

Pour pouvoir parcourir les éléments d'une collection, il faut disposer d'au moins 4 fonctions :

- ❑ initialiser un parcours
- ❑ récupérer la valeur courante
- ❑ avancer à la prochaine valeur
- ❑ vérifier si le parcours est terminé

Une solution serait d'intégrer ces fonctionnalités au type de la collection

Mais cela limiterait à un seul parcours simultané

Itérateur : spécification fonctionnelle

La solution concevoir un type reprenant ces fonctions afin de pouvoir disposer de plusieurs instances si besoin

`make_it : List -> ItList` // crée un itérateur initialisé sur la première valeur de la liste, ou finit le parcours si la liste est vide -> `make_it` appartiendra au type `List` pour s'assurer qu'un intégrateur ne soit pas créé indépendamment d'une `List`

`current: ItList -> (Int | Vide)` // retourne la valeur courante ou Vide si le parcours est fini

`next: ItList -> (ItList, (Int|Vide))` // retourne la valeur courante et avance sur la valeur suivante

`reinit: ItList -> ItList` // réinitialise l'itérateur, la valeur courante devient la première valeur de la liste

Problématique : pour implémenter les fonctions, on aura besoin d'avoir accès à la structure `List` !

Itérateur : notion de type ami

Pour pallier ce problème sans rendre public l'implémentation de List, il faut pouvoir autoriser le type `ItList` à y accéder tout en l'interdisant aux utilisateurs de List.

Solutions :

- ❑ Notion de type ami dans certains langages
- ❑ Niveau de protection interne à une librairie

En swift : utilisation de `fileprivate` pour limiter la portée du type `ListNE` au fichier et implémentation du type `ItList` dans le même fichier

Itérateur : description logique

```
ItList(list: List, lcurrent: ListNE?)  
// crée un itérateur initialisé sur la 1ère valeur, ou finit le parcours si  
la liste est vide  
func init(l: Liste) -> ItList{  
    return ItList(list: l, lcurrent: l.head)  
}  
// retourne la valeur courante ou Vide si le parcours est fini  
func current(it: ItList) -> (Int | Vide){  
    if it.lcurrent==Vide { return Vide }  
    else{ return it.lcurrent.val }  
}  
// retourne la valeur courante et avance sur la valeur suivante  
func next(it: ItList) -> (ItList, (Int | Vide)){  
    if it.lcurrent==Vide then { return (it, Vide) }  
    else{  
        let v = it.lcurrent.val ; it.lcurrent = it.lcurrent.next  
        return v  
    }  
}  
// réinitialise l'itérateur, comme lors de sa création  
func reinit(it: ItList) -> ItList{  
    it.lcurrent = it.head ; return it  
}
```

Itérateur : représentation physique

```
public struct ItList : IteratorProtocol{
    private(set) var list : List
    private var lcurrent : List.ListNE?
    public var current : Int? { return self.lcurrent?.val }
    fileprivate init(_ l: List){
        self.list = l
        self.lcurrent = l.head
    }
    public mutating func next() -> Int?{
        guard let list = self.lcurrent else { return nil }
        let val = list.val
        self.lcurrent = list.next
        return val
    }
    public mutating func reinit(){
        self.lcurrent = l.head
    }
}
```



```

struct List : TList, Sequence{
  fileprivate class ListNE{
    var val : Int
    var next : ListNE?
    init(val: Int, next: ListNE? = nil){
      self.val = val
      self.next = next
    }
  }
  [...]

  ///-----
  /// Iterator
  ///-----

  public func makeIterator() -> ItList {
    return ItList(self)
  }
}

```