V. Premiers types construits

Les collections de données de taille fixe

V.1 Le type tableau

On a vu les tableaux comme structures de données. Mais dans la plupart des langages modernes, les tableaux sont de véritables types.

Voici ce que pourrait être un type abstrait tableau :

Définition : Type Tableau

Un *tableau* est une collection d'éléments, chacun étant associé à un *indice* qui permet d'accéder *directement* à l'élément : le tableau est donc une collection ordonnées (au sens des indices) d'éléments à accès direct (via l'indice).

V.1.1 Le type abstrait TInt

Spécification fonctionnelle

```
init: Int x Int → TInt // init(a,n) crée un tableau de n éléments initialisés avec la valeur a

count : TInt → Int // donne le nombre d'éléments du tableau ; count(init(a,n)) == n

[] : TInt x Int → Int // retourne l'élément à l'indice donné en paramètre ;
[] : TInt x Int x Int → TInt // change la valeur de l'entier à l'indice par l'entier donné en paramètre
// T = init(a,n) ⇒ ∀ 0 ⩽ j < n, T[j]==a ; T[j]=k ⇒ T[j]==k

contains: TInt x Int → Bool // True si l'entier donnée en paramètre appartient au tableau
// contains(T,a) ⇒ ∃i, T[i]==a
```

`firstIndex : TInt x Int → Int | Vide // indice de la première valeur (dans l'ordre des indices) donnée en paramètre`
`// firstIndex(T,a)==i ⇒ T[i]==a et` $\forall\, 0 \leqslant j < i$, `T[j]≠a`
`// firstIndex(T,a)==Vide ⇒` $\forall\, 0 \leqslant i < n$, `T[i]≠a`

`lastIndex : TInt x Int → Int | Vide // indice de la dernière valeur (dans l'ordre des indices) donnée en paramètre`
`// lastIndex(T,a)==i ⇒ T[i]==a et` $\forall\, i < j < n$, `T[j]≠a`
`// lastIndex(T,a)==Vide ⇒` $\forall\, 0 \leqslant i < n$, `T[i]≠a`

`nbOccur : TInt x Int → Int // nombre d'occurrences de la valeur passée en paramètre`
`// nbOccur(T,a)==p ⇒` $\exists i_0, \ldots, i_p$ `tel que` $\forall i_{k,0 \leqslant k \leqslant p}, i_k \in [0, n-1]$ `et` $T[i_k]$`==a`

`Min : TInt → Int // plus petite valeur de T`
`// min(T)==m ⇒` $\forall\, 0 \leqslant i < n, T[i] \geqslant m$
`Max : TInt → Int // plus grande valeur de T`
`// max(T)==m ⇒` $\forall\, 0 \leqslant i < n, T[i] \leqslant m$

Le type abstrait TInt : tableau d'entiers

V.1.2 Le type concret TInt

1. Proposez une structure de données pour le type TInt
2. Implémentez les fonctions du type, en prouvant à chaque fois vos algorithmes et en évaluant leur complexité.
3. On souhaite également implémenter d'autres fonctions, à chaque fois vous prouverez votre algorithme et donnerez sa complexité :
    - 3.1. la fonction compare
    - 3.2. la fonction isSub

```
Compare : TInt x TInt → Int // cette fonction compare deux tableaux T1 et T2 et
- retourne -1 si tous les éléments de T1 sont plus petits ou égaux aux éléments de T2 aux mêmes indices.
  exemple : T1=[1,2,1] et T2=[1,2,3]
- retourne 0 si ils sont tous égaux
- 1 sinon, par exemple T1=[1,4,1] et T2=[1,2,3]

isSub : TInt x TInt → Bool // isSub(T1,T2) retourne True si T1 est sous-tableau de T2 et False sinon.
// T1 est sous-tableau de T2 si
// ∃ 0 ⩽ k < count(T2), tel que ∀ 0 ⩽ i < count(T1), T1[i]==T2[k+i]
// et tel que k + count(T1) ⩽ count(T2)
exemples :
- isSub([3,4,5],[1,2,3,4,5,6])==True
- isSub([3,4,5],[1,3,2,4,5,6])==False
- isSub([3,4,5],[1,3,4])==False
```

V.1.3 L'opérator [ ] en swift

Swift permet de coder l'opérateur crochet dans un type en rajoutant la fonction subscript (ex pour des valeurs Int):

```
subscript(index: Int) -> Int {
  get { } // retourne la valeur à l'index.
  set { } // change la valeur à l'index.
}
```

Exemple :

...

```swift
struct Dummy{
  var p : Int
  var s : Int
  init(_ a: Int, _ b: Int){ self.p = a ; self.s = b }
  subscript(i : Int) -> Int{
    get {
      assert( (i >= 0) && (i < 2) , "Index out of range")
      if (i == 0) { return self.p }
      else { return self.s }
    }
    set{
      assert( (i >= 0) && (i < 2) , "Index out of range")
      if (i == 0) { self.p = newValue }
      else { self.s = newValue }
    }
  }
}
var d = Dummy(1, 2)
d[0] = 0 ; print("d[0]=\(d[0])")
```


L'opérator [ ] en swift