Haute École de Bruxelles
École Supérieure d'Informatique
Bachelor en Informatique

Rue Royale, 67. 1000 Bruxelles
02/219.15.46 – esi@heb.be

DEV 2

Algorithmique

2016

Activité d'apprentissage enseignée par :

???

Document produit avec LaTeX.
Version du 18 octobre 2015.

Table des matières

Chapitre 1

Les tableaux à 2 dimensions

1.1 Définition

La **dimension** d'un tableau est le nombre d'indices qu'on utilise pour faire référence à un de ses éléments. Attention de ne pas confondre avec la taille !

En DEV_1, nous avons introduit les tableaux à une dimension. Un seul indice suffisait à localiser un de ses éléments. Pour le dire autrement, chaque case possédait **un** numéro. De nombreuses situations nécessitent cependant l'usage de tableaux à deux dimensions. Ils vous sont déjà familiers par leur présence dans beaucoup de situations courantes : calendrier, grille horaire, grille de mots croisés, sudoku, jeux se déroulant sur un quadrillage (damier, échiquier, scrabble...). Dans ces situations, chaque case est désignée par **deux** numéros.

1.2 Notations

1.2.1 Déclarer

Pour **déclarer** un tableau à 2 dimensions, on écrira :

```
nomTableau : tableau de nbLignes × nbColonnes TypeElément
```

où nbLignes et nbColonnes sont des expressions entières quelconques.

Exemple :

```
tab : tableau de 5×10 entiers
```

déclare un tableau de 5 lignes par 10 colonnes dont chaque case contient un entier.

1.2.2 Utiliser

Pour **accéder** à une case du tableau on donnera les deux indices entre crochets. Comme en DEV_1, on va considérer que la première ligne et la première colonne porte le numéro (l'indice) 0.

Exemple :

```
afficher tab[2,4]                    // affiche le 5e élément de la 3e ligne du tableau nommé tab.
```

1.2.3 Visualiser

Notez que la vue sous forme de tableau avec des lignes et des colonnes est une vision humaine. Il n'y a pas de lignes ni de colonnes en mémoire. Pour être précis, on devrait juste parler de première dimension et de deuxième dimension mais la notion de ligne et de colonne est un abus de langage qui simplifie le discours.

On pourrait aussi visualiser un tableau à deux dimensions comme un tableau à une dimension dont chacun des éléments est lui-même un tableau à une dimension.

Exemple : Soit le tableau déclaré ainsi :

```
tabLettres : tableau de 4×5 caractères
```

On peut le visualiser à l'aide d'une grille à 4 lignes et 5 colonnes.

	0	1	2	3	4
0	d	h	v	q	z
1	j	g	k	o	u
2	i	f	y	r	t
3	n	d	e	a	s

Ainsi, la valeur de tabLettres[2,3] est le caractère 'r'.

La vision « tableau de tableau » (ou décomposition en niveaux) donnerait :

0					1					2					3				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
d	h	v	q	z	j	g	k	o	u	i	f	y	r	t	n	d	e	a	s

Dans cette représentation, le tableau tabLettres est d'abord décomposé à un premier niveau en quatre éléments auxquels on accède par le premier indice. Ensuite, chaque élément de premier niveau est décomposé en cinq éléments de deuxième niveau accessibles par le deuxième indice.

1.2.4 Exemples

Exemple 1 – Remplir les coins. Dans ce petit exemple, on a un tableau de chaines et on donne des valeurs aux coins.

"NO"				"NE"
"SO"				"SE"

```
// Déclare un tableau et donne des valeurs aux coins.
algorithme remplirCoins()
    grille : tableau de 3×5 entiers
    grille[0,0] ← "NO"
    grille[0,4] ← "NE"
    grille[2,0] ← "SO"
    grille[2,4] ← "SE"
fin algorithme
```

Exemple 2 – Gestion des stocks. Reprenons l'exemple du stock de 10 produits qui a servi d'introduction au chapitre sur les tableaux mais, cette fois, pour chaque jour de la semaine.

	article0	article1	article2	...	article7	article8	article9
lundi	cpt[0,0]	cpt[0,1]	cpt[0,2]	...	cpt[0,7]	cpt[0,8]	cpt[0,9]
mardi	cpt[1,0]	cpt[1,1]	cpt[1,2]	...	cpt[1,7]	cpt[1,8]	cpt[1,9]
mercredi	cpt[2,0]	cpt[2,1]	cpt[2,2]	...	cpt[2,7]	cpt[2,8]	cpt[2,9]
jeudi	cpt[3,0]	cpt[3,1]	cpt[3,2]	...	cpt[3,7]	cpt[3,8]	cpt[3,9]
vendredi	cpt[4,0]	cpt[4,1]	cpt[4,2]	...	cpt[4,7]	cpt[4,8]	cpt[4,9]
samedi	cpt[5,0]	cpt[5,1]	cpt[5,2]	...	cpt[5,7]	cpt[5,8]	cpt[5,9]
dimanche	cpt[6,0]	cpt[6,1]	cpt[6,2]	...	cpt[6,7]	cpt[6,8]	cpt[6,9]

```
// Calcule et affiche la quantité vendue de 10 produits
// pour chaque jour de la semaine (de 0 : lundi à 6 : dimanche).
algorithme statistiquesVentesSemaine()

    cpt : tableau de 7×10 entiers

    initialiser(cpt)

    // Pour chaque jour de la semaine
    pour jour de 0 à 6 faire
        traiterStock1Jour(cpt, jour)
        pour produit de 0 à 9 faire
            afficher "quantité vendue de produit ", produit, " ce jour ", jour, " : ", cpt[jour,i]
        fin pour
    fin pour
fin algorithme
```

```
// Initialise le tableau d'entiers à 0
algorithme initialiser(entiers↓↑ : tableau de 7×10 entiers)
    pour i de 0 à 6 faire
        pour j de 0 à 9 faire
            cpt[i,j] ← 0
        fin pour
    fin pour
fin algorithme
```

```
// Effectue le traitement du stock pour une journée.
algorithme traiterStock1Jour(cpt↓↑ : tableau de 7×10 entiers, jour  : entier)
    numéroProduit, quantité : entiers
    afficher "Introduisez le numéro du produit :"
    demander numéroProduit

    tant que numéroProduit ≥ 0 faire

        afficher "Introduisez la quantité vendue :"
        demander quantité

        cpt[jour,numéroProduit] ← cpt[jour,numéroProduit] + quantité

        afficher "Introduisez le numéro du produit :"
        demander numéroProduit

    fin tant que
fin algorithme
```

Pour plus d'exemples, allez faire un tour à la section 1.4 page suivante.

1.3 La troisième dimension (et au-delà)

Certaines situations complexes nécessitent l'usage de tableaux à 3 voire plus de dimensions.

Pour déclarer un tableau statique à k dimensions, on écrira :

```
nomTableau : tableau de tailleDim1 × ... × tailleDimK TypeElément
```

1.4 Parcours d'un tableau à deux dimensions

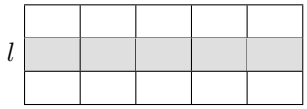
Comme nous l'avons fait pour les tableaux à une dimension, envisageons le parcours des tableaux à deux dimensions (n lignes et m colonnes). Nos algorithmes sont valables quel que soit le type des éléments. Utilisons T pour désigner un type quelconque.

```
tab : tableau de n × m T
```

Commençons par des cas plus simples où on ne parcourt qu'une seule des dimensions puis attaquons le cas général.

1.4.1 Parcours d'une dimension

On peut vouloir ne parcourir qu'une seule ligne du tableau. Si on parcourt la ligne l, on visite les cases $(l, 0)$, $(l, 1)$, ..., $(l, m-1)$. L'indice de ligne est constant et c'est l'indice de colonne qui varie.



Ce qui donne l'algorithme :

```
// Parcours de la ligne l d'un tableau à deux dimensions
pour c de 0 à m-1 faire
    afficher tab[l,c]                          // On peut faire autre chose qu'afficher
fin pour
```

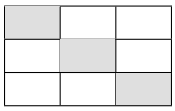
Retenons : pour parcourir une ligne, on utilise une boucle sur les colonnes.

Symétriquement, on pourrait considérer le parcours de la colonne c avec l'algorithme suivant.

```
// Parcours de la colonne c d'un tableau à deux dimensions
pour l de 0 à n-1 faire
    afficher tab[l,c]                          // On peut faire autre chose qu'afficher
fin pour
```

Si le tableau est carré ($n = m$) on peut aussi envisager le parcours des deux diagonales.

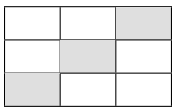
Pour la diagonale descendante, les éléments à visiter sont $(0, 0)$, $(1, 1)$, $(2, 2)$, ..., $(n-1, n-1)$.



Une seule boucle suffit comme le montre l'algorithme suivant.

```
// Parcours de la diagonale descendante d'un tableau carré
pour i de 0 à n-1 faire
    afficher tab[i,i]                          // On peut faire autre chose qu'afficher
fin pour
```

Pour la diagonale montante, on peut envisager deux solutions, avec deux indices ou un seul en se basant sur le fait que $i + j = n - 1 \Rightarrow j = n - 1 - i$.



```
// Parcours de la diagonale montante d'un tableau carré - 2 indices
j ← n-1
pour i de 0 à n-1 faire
    afficher tab[i,j]                                   // On peut faire autre chose qu'afficher
    j ← j - 1
fin pour
```

```
// Parcours de la diagonale montante d'un tableau carré - 1 indice
pour i de 0 à n-1 faire
    afficher tab[i, n - 1 - i]                          // On peut faire autre chose qu'afficher
fin pour
```

1.4.2 Parcours des deux dimensions

Parcours par lignes et par colonnes

Les deux parcours les plus courants sont les parcours ligne par ligne et colonne par colonne. Les tableaux suivants montrent dans quel ordre chaque case est visitée dans ces deux parcours.

Parcours ligne par ligne

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Parcours colonne par colonne

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

Le plus simple est d'utiliser deux boucles imbriquées

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne
pour lg de 0 à n-1 faire
    pour col de 0 à m-1 faire
        afficher tab[lg,col]                            // On peut faire autre chose qu'afficher
    fin pour
fin pour
```

```
// Parcours d'un tableau à 2 dimensions, colonne par colonne
pour col de 0 à m-1 faire
    pour lg de 0 à n-1 faire
        afficher tab[lg,col]                            // On peut faire autre chose qu'afficher
    fin pour
fin pour
```

Mais on peut obtenir le même résultat avec une seule boucle si l'indice sert juste à compter le nombre de passages et que les indices de lignes et de colonnes sont gérés manuellement.

L'algorithme suivant montre ce que ça donne pour un parcours ligne par ligne. La solution pour un parcours colonne par colonne est similaire et laissée en exercice.

```
// Parcours d'un tableau à 2 dimensions via une seule boucle
lg ← 0
col ← 0
pour i de 0 à n*m-1 faire
    afficher tab[lg,col]                                  // On peut faire autre chose qu'afficher
    col ← col + 1                                                    // Passer à la case suivante
    si col = m alors                          // On déborde sur la droite, passer à la ligne suivante
        col ← 0
        lg ← lg + 1
    fin si
fin pour
```

L'avantage de cette solution apparaitra quand on verra des situations plus difficiles.

Interrompre le parcours

Comme avec les tableaux à une dimension, envisageons l'arrêt prématuré lors de la rencontre d'une certaine condition. Et, comme avec les tableaux à une dimension, transformons d'abord nos **pour** en **tant que**.

Par exemple, montrons les deux parcours ligne par ligne, avec une et deux boucle(s).

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne, via un tant que
lg ← 0
tant que lg < n faire
    col ← 0
    tant que col < m faire
        afficher tab[lg, col]                             // On peut faire autre chose qu'afficher
        col ← col + 1
    fin tant que
    lg ← lg + 1
fin tant que
```

```
// Parcours d'un tableau à 2 dimensions via une seule boucle et un tant que
lg ← 0
col ← 0
i ← 0
tant que i < n*m faire                                                    // ou "lg < n"
    afficher tab[lg,col]                                  // On peut faire autre chose qu'afficher
    col ← col + 1                                                    // Passer à la case suivante
    si col = m alors                          // On déborde sur la droite, passer à la ligne suivante
        col ← 0
        lg ← lg + 1
    fin si
    i ← i + 1
fin tant que
```

On peut à présent introduire le test comme on l'a fait dans les algorithmes de parcours des tableaux à une dimension.

Illustrons-le au travers de deux exemples où on cherche un élément particulier. Le premier introduit un test en utilisant un booléen alors que le second introduit un test sans utiliser de booléen.

```
// Parcours avec test d'arrêt - deux boucles et un booléen
trouvé ← faux
lg ← 0
tant que lg < n ET NON trouvé faire
    col ← 0
    tant que col < m ET NON trouvé faire
        si tab[lg, col] est l'élément recherché alors
            trouvé ← vrai
        sinon                                   // Ne pas modifier les indices si arrêt demandé
            col ← col + 1
        fin si
    fin tant que
    si NON trouvé alors                         // Ne pas modifier les indices si arrêt demandé
        lg ← lg + 1
    fin si
fin tant que
// Tester trouvé pour savoir si on a trouvé l'élement recherché
```

```
// Parcours avec test d'arrêt - une boucle et pas de booléen
lg ← 0
col ← 0
i ← 0
tant que i < n*m ET tab[lg, col] n'est pas l'élément recherché faire
    col ← col + 1                                          // Passer à la case suivante
    si col = m alors                  // On déborde sur la droite, passer à la ligne suivante
        col ← 0
        lg ← lg + 1
    fin si
    i ← i + 1
fin tant que
// L'élément recherché a été trouvé si i < n*m.
```

Parcours plus compliqué - le serpent

Envisageons un parcours plus difficile illustré par le tableau suivant.

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

Le plus simple est d'adapter l'algorithme de parcours avec une seule boucle en introduisant un sens de déplacement, ce qui donne l'algorithme :

```
// Parcours du serpent dans un tableau à deux dimensions
lg ← 0
col ← 0
depl ← 1                                        // 1 pour avancer, -1 pour reculer
pour i de 0 à n*m-1 faire
    afficher tab[lg, col]                       // On peut faire autre chose qu'afficher
    si 0 ≤ col + depl ET col + depl < m alors
        col ← col + depl                        // On se déplace dans la ligne
    sinon
        lg ← lg + 1                             // On passe à la ligne suivante
        depl ← -depl                            // et on change de sens
    fin si
fin pour
```

1.5 Exercices

1 Affichage

Écrire un module qui affiche tous les éléments d'un tableau à n lignes et m colonnes
a) ligne par ligne b) colonne par colonne

2 Les nuls

Écrire un module qui reçoit un tableau (n x m) d'entiers et qui affiche la proportion d'éléments nuls dans ce tableau.

3 Tous positifs

Écrire un module qui reçoit un tableau (n x m) d'entiers et qui vérifie si tous les nombres qu'il contient sont strictement positifs. Bien sûr, on veillera à éviter tout travail inutile ; la rencontre d'un nombre négatif doit arrêter le module.

4 Le tableau de cotes

Soit un tableau à n lignes et m colonnes d'entiers où une ligne représente les notes sur 20 d'un étudiant et les colonnes toutes les notes d'un cours.

Écrire un algorithme recevant ce tableau en paramètre et affichant le pourcentage d'étudiants ayant obtenu une moyenne supérieure à 50%.

5 Le carré magique

Un carré magique est un tableau d'entiers carré (c'est-à-dire possédant autant de lignes que de colonnes) ayant la propriété suivante : si on additionne les éléments d'une quelconque de ses lignes, de ses colonnes ou de ses deux diagonales, on obtient à chaque fois le même résultat.

Écrire un module recevant en paramètres le tableau [1 à n, 1 à n] d'entiers représentant le carré et renvoyant une valeur booléenne indiquant si c'est un carré magique ou pas.

6 Le triangle de Pascal

Le triangle de Pascal est construit de la façon suivante :

- ▷ la ligne initiale contient un seul élément de valeur 1 ;
- ▷ chaque ligne possède un élément de plus que la précédente ;
- ▷ chaque ligne commence et se termine par 1 ;
- ▷ pour calculer un nombre d'une autre case du tableau, on additionne le nombre situé dans la case située juste au-dessus avec celui dans la case à la gauche de la précédente.

Écrire un module qui reçoit en paramètre un entier n, et qui renvoie un tableau contenant les $n + 1$ premières lignes du triangle de Pascal (indicées de 0 à n).

N.B. : le « triangle » sera bien entendu renvoyé dans un tableau carré. Quid des cases non occupées ?

Par exemple, pour n qui vaut 5, on aura le tableau suivant :

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

7 Lignes et colonnes

Écrire un module qui reçoit un tableau d'entiers à 2 dimensions en paramètre et qui retourne un booléen indiquant si ce tableau possède 2 lignes ou 2 colonnes identiques.

Dans l'affirmative, ce module renverra également en paramètres les informations suivantes :

- ▷ les indices des lignes ou colonnes identiques
- ▷ un caractère valant 'L' ou 'C' selon qu'il s'agit de lignes ou de colonnes

Dans la négative, les valeurs de ces paramètres seront indéterminées ou quelconques, elles ne seront de toute façon pas utilisées par le module appelant.

8 Le contour du tableau

On donne un tableau d'entiers tabEnt à n lignes et m colonnes. Écrire un module retournant la somme de tous les éléments *impairs* situés sur le bord du tableau.

Exemple : pour le tableau suivant, le module doit renvoyer 32

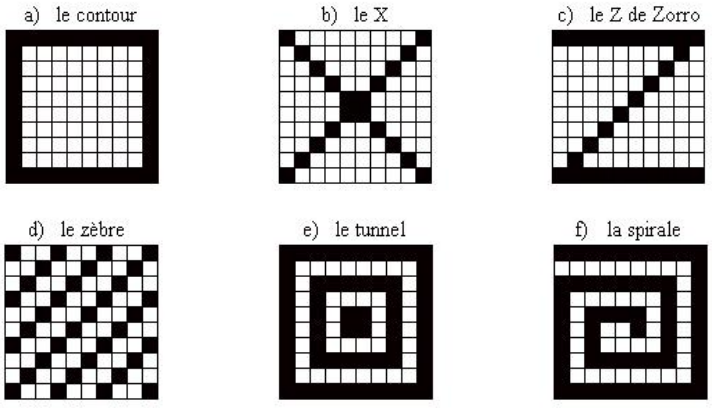
3	4	6	11
2	21	7	9
1	5	12	3

Et pour le suivant, le module doit renvoyer 6

4	1	2	8	5

9 À vos pinceaux !

On possède un tableau à n lignes et n colonnes dont les éléments de type Couleur valent NOIR ou BLANC. On suppose que le tableau est initialisé à BLANC au départ. Écrire un module qui *noircit* les cases de ce tableau comme le suggèrent les dessins suivants (les exemples sont donnés pour un tableau 10 x 10 mais les algorithmes doivent fonctionner quelle que soit la taille du tableau).



Notes

- ▷ Le zèbre doit toujours présenter des lignes obliques et parallèles, quelle que soit la taille.
- ▷ La spirale est un véritable défi et vous est donné comme exercice facultatif. Ne le faites pas si vous êtes en retard.

10 Exercices sur la complexité

Quelle est la complexité

a) d'un algorithme de parcours d'un tableau n x n ?

b) des algorithmes que vous avez écrits pour les exercices : "Les nuls", "Tous positifs", "Le carré magique" et "Le contour d'un tableau" ?

c) des algorithmes que vous avez écrits pour résoudre les exercices du pinceau ?

Chapitre 2

L'orienté objet

Chapitre 3

La liste

Imaginons qu'on désire manipuler par programme une liste de contacts ou encore une liste de rendez-vous. Cette liste va varier ; sa taille n'est donc pas fixée. Utiliser un tableau à cet effet n'est pas l'idéal. En effet, la taille d'un tableau ne peut plus changer une fois le tableau créé. Il faudrait le sur-dimensionner, ce qui n'est pas économe.

Il serait intéressant de disposer d'une structure qui offre toutes les facilités d'un tableau tout en pouvant « grandir » si nécessaire. Construisons une telle structure de données et appelons-la « Liste » pour rester en phase avec son appellation commune en Java.

Par exemple, considérons une liste de courses. On pourrait la représenter ainsi :

1. "fromage"
2. "pain"
3. "salami"

On pourrait ajouter un élément en fin de liste, par exemple de l'eau, pour obtenir la liste :

1. "fromage"
2. "pain"
3. "salami"
4. "eau"

On pourrait aussi supprimer un élément de la liste, par exemple le pain, et obtenir :

1. "fromage"
2. "salami"
3. "eau"

On pourrait aussi insérer un élément dans la liste, par exemple une baguette, ce qui décale, de facto, la position des suivants.

1. "fromage"
2. "salami"
3. "baguette"
4. "eau"

Et encore plein de choses que nous allons détailler.

3.1 La classe Liste

Nous verrons plus loin comment réaliser une classe Liste en pratique mais nous pouvons déjà définir le comportement qu'on en attend (les méthodes qu'elle doit fournir)

Ce comportement sera indentique quel que soit le type des éléments de la liste ; une liste de chaine et une liste d'entiers ne se distinguent que par le type de certains paramètres et valeurs de retour. Ici, nous indiquons T pour indiquer un type quelconque ; vous pouvez le remplacer par ce qui vous convient : entier, chaine, Date. . ..

```
classe Liste de T                                                    // T est un type quelconque
 privé:
    // sera complété plus tard
 public:
    constructeur Liste de T()                                        // construit une liste vide
    méthode get(pos : entier) → T                                    // donne un élément en position pos
    méthode set(pos : entier, valeur : T)                            // modifie un élément en position pos
    méthode taille() → entier                                        // donne le nombre actuels d'éléments
    méthode ajouter(valeur : T)                                      // ajoute un élément en fin de liste
    méthode insérer(pos : entier, valeur : T)                        // insère un élément en position pos
    méthode supprimer()                                              // supprime le dernier élément
    méthode supprimerPos(pos : entier)                               // supprime l'élément en position pos
    méthode supprimer(valeur : T) → booléen                          // supprime l'élément de valeur donnée
    méthode vider()                                                  // vide la liste
    méthode estVide() → booléen                                      // la liste est-elle vide ?
    méthode existe(valeur ↓ : T, pos ↑ : entier) → booléen           // recherche un élément
fin classe
```

Quelques précisions s'imposent :

- ▷ Les méthodes « get » et « set » permettent de connaitre ou modifier un élément de la liste. On considère, au cours d'algorithmique, que le premier élément de la liste est en position 0.
- ▷ « ajouter » ajoute un élément en fin de liste (elle grandit donc d'une unité)
- ▷ « insérer » insère un élément à une position donnée (entre 1 et taille+1). L'élément qui s'y trouvait est décalé d'une position ainsi que tous les éléments suivants.
- ▷ La méthode « supprimerPos » supprime un élément d'une position donnée en décalant les éléments suivants. On pourrait imaginer une technique plus rapide consistant à placer le dernier élément à la place de l'élément supprimé mais ce faisant on changerait l'ordre relatif des éléments ce qui va à l'encontre de l'idée intuitive qu'on se fait d'une liste. Cette amélioration pourrait plutôt s'envisager dans une structure de type **ensemble** pour lequel il n'y a pas d'ordre relatif entre les éléments.
- ▷ La version de « supprimer » avec une valeur en paramètre enlève un élément de valeur donnée. Elle retourne un booléen indiquant si la suppression a pu se faire ou pas (ce qui sera le cas si la valeur n'est pas présente dans la liste). Si la valeur existe en plusieurs exemplaires, on prendra la convention arbitraire que la méthode n'en supprime que la première occurrence.
- ▷ La méthode « existe » permet de savoir si un élément donné existe dans la liste.
  - ▷ si c'est le cas, elle précise aussi sa position dans le paramètre sortant pos
  - ▷ si l'élément n'existe pas, ce paramètre est indéterminé
  - ▷ si l'élément est présent en plusieurs exemplaires, la méthode donne la position de la première occurrence.
- ▷ En pratique, il serait intéressant de chercher un élément à partir d'une partie de l'information qu'elle contient mais c'est difficile à exprimer de façon générique c'est-à-dire lorsque le type n'est pas connu à priori.

Exemple : manipulations de base

Soit l'algorithme suivant :

```
algorithme ex1()
    l : Liste d'entiers
    l ← nouvelle Liste d'entiers()
    l.ajouter(42)
    l.ajouter(54)
    l.set(1,44)
    l.insérer(1,43)
    l.supprimerPos(2)
    l.supprimer(42)
    l.vider
fin algorithme
```

Après sa création, la liste est vide. Ensuite, elle passe par les états suivants :

0. 42	0. 42	0. 42	0. 42	0. 42	0. 43
	1. 54	1. 44	1. 43	1. 43	
			2. 44		

Enfin, le dernier appel la vide complètement

Exemple : recherche du minimum

Dans le chapitre sur les tableaux, vous avez fait un exercice consistant à afficher tous les indices où se trouve le minimum d'un tableau. Reprenons-le et modifions-le afin qu'il retourne la liste des indices où se trouvent les différentes occurrences du minimum. On pourrait l'écrire ainsi :

```
algorithme indicesMinimum(tab : tableau de n entiers) → Liste d'entiers
    min : entier
    indicesMin : Liste d'entiers
    min ← tab[0]
    indicesMin ← nouvelle Liste d'entiers()
    indicesMin.ajouter( 0 )
    pour i de 1 à n-1 faire
        si tab[i] = min alors
            indicesMin.ajouter( i )
        sinon si tab[i] < min alors
            indicesMin.vider()
            indicesMin.ajouter( i )
            min ← tab[i]
        fin si   // rien à faire si tab[i] > min
    fin pour
    retourner indicesMin
fin algorithme
```

3.2 Comment implémenter l'état

Cette liste est bien utile mais comment la réaliser en pratique ? Comment représenter une liste variable d'éléments ? Pour l'instant, la seule structure qui peut accueillir plusieurs éléments de même type est le tableau. Nous allons donc prendre comme attribut principal de la liste, un tableau que nous appellerons éléments. Comment, dès lors, contourner le problème de la limitation de la taille de ce tableau ?

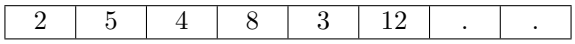
Repartons donc de la notion de tableau et tentons de comprendre sa limitation. Lors de sa création, un tableau se voit attribuer un espace bien précis et contigu en mémoire. Il se peut

très bien que l'espace « juste après » soit occupé par une autre variable ce qui l'empêche de grandir. La parade est claire : si un tableau s'avère trop petit lors de son utilisation, il suffit d'en créer un autre plus grand ailleurs en mémoire et d'y recopier tous les éléments du premier. Évidemment, cette opération est coûteuse en temps et on cherchera à l'effectuer le moins souvent possible.

Quelle taille donner au nouveau tableau ? L'idée qui vient immédiatement est d'augmenter la taille d'une unité afin d'accueillir le nouvel élément mais cette approche implique de fréquents agrandissements. Il est plus efficace d'augmenter la taille proportionnellement, par exemple en la multipliant par un facteur 2.

Taille logique et taille physique. À tout moment, le tableau aura une et une seule taille même si celle-ci pourra changer au cours du temps. Puisqu'on multipliera la taille du tableau par 2 pour des raisons d'efficacité, il y aura toutefois une différence entre la **taille physique** d'un tableau et sa **taille logique**. La taille physique est le nombre de cases réservées pour le tableau alors que la taille logique est le nombre de cases effectivement occupées. Dans ce qui suit, on s'arrangera pour que les cases occupées soient groupées à gauche du tableau (il n'y a pas de trou). Pour l'utilisateur, seule la taille logique a un sens (on lui cache les détails d'implémentation).

Exemple : pour le tableau suivant, la taille logique est de 6 (c'est cette taille qui a du sens pour l'utilisateur de la liste) et la taille physique est de 8.



Quand il faut insérer un élément (en position valide) ou en ajouter un en fin de liste, deux cas se présentent :

- ▷ si la taille logique est plus petite que la taille physique, il suffit d'ajouter l'élément dans le tableau et d'adapter la taille logique.
- ▷ si la taille logique est égale à la taille physique, il faut procéder à un agrandissement du tableau.

Les tableaux dynamiques. En DEV_1, nous n'avons vu que des tableaux qu'on appellera *statiques*, qui sont créés lors de leur déclaration. Ici, nous avons besoin de tableaux qu'on appellera *dynamiques*, créés dans le code (comme le sont les tableaux en Java).

Introduisons une notation. Un tableau dynamique sera déclaré puis créé ainsi :

```
tab : tableau de chaines
tab ← nouveau tableau de n chaines                    // n doit avoir une valeur
```

Implémentation. Présentons les attributs nécessaires et l'algorithme d'agrandissement du tableau.

```
classe Liste de T
 privé:
    éléments : tableau de T
    tailleLogique : entier
    taillePhysique : entier
 privé:
    méthode agrandir()
        nouveauTab : tableau de T
        taillePhysique ← taillePhysique * 2
        nouveauTab ← nouveau tableau de taillePhysique T
        pour i de 0 à tailleLogique-1 faire
            nouveauTab[ i ] ← éléments[ i ]
        fin pour
        éléments ← nouveauTab
    fin méthode
fin classe
```

Réduction du tableau. Tout comme on agrandit le tableau si nécessaire, on pourrait le réduire lorsque des suppressions d'éléments le rendent sous-utilisé (par exemple lorsque la taille logique devient inférieure au tiers de la taille physique). Nous n'aborderons pas cette problématique cette année.

3.3 Implémentation du comportement

Nous avons à présent toutes les cartes en main pour écrire les méthodes publiques de la classe.

```
constructeur Liste de T()
    tailleLogique ← 0                                   // la liste est vide au départ
    taillePhysique ← 32                                 // une bonne valeur pour commencer
    éléments ← nouveau tableau de taillePhysique T
fin constructeur
```

```
méthode get(pos : entier) → T
    si  pos < 0 OU pos ≥ tailleLogique alors
        erreur "position invalide"
    fin si
    retourner éléments[ pos ]
fin méthode
```

```
méthode set(pos : entier, valeur : T)
    si  pos < 0 OU pos ≥ tailleLogique alors
        erreur "position invalide"
    fin si
    éléments[ pos ] ← valeur
fin méthode
```

```
méthode taille() → entier
    retourner tailleLogique                             // et pas la taille physique !
fin méthode
```

```
méthode ajouter(valeur : T)
    si tailleLogique = taillePhysique alors
        agrandir()                                      // méthode privée détaillée supra
    fin si
    éléments[ tailleLogique ] ← valeur
    tailleLogique ← tailleLogique + 1
fin méthode
```

```
méthode insérer(pos : entier, valeur : T)
    si  pos < 0 OU pos ≥ tailleLogique alors
        erreur "position invalide"
    fin si
    si tailleLogique = taillePhysique alors
        agrandir()
    fin si
    décalerDroite( pos )                                    // voir ci-dessous
    tailleLogique ← tailleLogique + 1
    éléments[ pos ] ← valeur
fin méthode
```

```
méthode supprimer()
    // supprime le dernier élément
    si tailleLogique = 0 alors
        erreur "liste vide"
    fin si
    tailleLogique ← tailleLogique - 1
fin méthode
```

```
méthode supprimerPos(pos : entier)
    si  pos < 1 OU pos > tailleLogique alors
        erreur "position invalide"
    fin si
    décalerGauche( pos + 1 )                                // voir méthode ci-dessous
    tailleLogique← tailleLogique - 1
fin méthode
```

```
méthode supprimer(valeur : T) → booléen
    estPrésent : booléen
    pos : entier
    estPrésent ← existe(valeur, pos)
    si estPrésent alors
        supprimer( pos )
    fin si
    retourner estPrésent
fin méthode
```

```
méthode vider()
    tailleLogique ← 0                  // Les éléments ne sont pas effacés mais sont ignorés
fin méthode
```

```
méthode estVide() → booléen
    retourner tailleLogique = 0
fin méthode
```

```
méthode existe(valeur↓ : T, pos↑ : entier) → booléen
    pos ← 0
    // Rq : le ET ci-dessous est une évaluation court-circuitée (cf. le cours d'Algo en DEV1)
    tant que  pos < tailleLogique ET éléments[ pos ] ≠ valeur faire
        pos ← pos + 1
    fin tant que
    retourner pos < tailleLogique
fin méthode
```

```
// Ces méthodes-ci sont privées

méthode décalerDroite(début : entier)
    // Décale tous les éléments d'une position vers la droite à partir de début
    pour i de tailleLogique-1 à début par -1 faire
        éléments[ i + 1 ] ← éléments[ i ]
    fin pour
fin méthode

méthode décalerGauche(début : entier)
    // Décale toutes les éléments d'une position vers la gauche à partir de début ;
    // ce paramètre vaut toujours au moins 2.
    pour i de début à tailleLogique-1 faire
        éléments[ i - 1 ] ← éléments[ i ]
    fin pour
fin méthode
```

La recherche se fait sur un élément complet.

Prenons comme exemple une liste de contacts. Lors d'une recherche, on doit fournir **tout** le contact à rechercher. Il s'agit juste de savoir s'il est présent et où. Une autre méthode intéressante serait de retrouver un contact à partir d'une partie de l'information, par exemple son nom. Cette méthode est fort proche de notre méthode de recherche mais il serait très difficile de l'écrire génériquement. On vous demandera d'écrire explicitement une telle méthode de recherche en cas de besoin.

3.4 Et sans tableau dynamique ?

Certains langages (c'est le cas de Cobol) ne permettent pas de créer dynamiquement un nouveau tableau. Il vous faudra travailler avec un tableau classique en le créant suffisamment grand.

Les algorithmes d'ajout/suppression/recherche vus pour la liste peuvent être appliqués tels quels à un tableau statique à une modification près : lors d'un ajout dans un tableau plein, on ne peut pas l'agrandir ; il faut générer une erreur.

3.5 Exercices

1 Manipulation d'une liste

Écrire un module qui crée la liste suivante :

0. 494
1. 209
2. 425

affiche sa taille, demande si la valeur 425 est présente, supprime la valeur 209 puis insére la valeur 101 en tête de liste.

2 Liste des premiers entiers

Écrire un module qui reçoit un entier n en paramètre et retourne la liste contenant les entiers de 1 à n dans l'ordre décroissant. On peut supposer que n est positif.

3 Somme d'une liste

Écrire un module qui calcule la somme des éléments d'une liste d'entiers.

4 Les extrêmes

Écrire un module qui supprime le minimum et le maximum des éléments d'une liste d'entiers. On peut supposer que le maximum et le minimum sont uniques.

5 Anniversaires

Écrire un module qui reçoit une liste de structure Personne (nom + prénom + date de naissance) et retourne la liste de ceux qui sont nés durant un mois passé en paramètre (donné sous la forme d'un entier entre 1 et 12).

6 Concaténation de deux listes

Écrire un module qui reçoit 2 listes et ajoute à la suite de la première les éléments de la seconde ; la seconde liste n'est pas modifiée par cette opération.

7 Fusion de deux listes

Soit deux listes **ordonnées** d'entiers (redondances possibles). Écrire un module qui les fusionne. Le résultat est une liste encore ordonnée contenant tous les entiers des deux listes de départ (qu'on laisse inchangées).

Exemple : Si les 2 listes sont (1, 3, 7, 7) et (3, 9), le résultat est (1, 3, 3, 7, 7, 9).

8 Le nettoyage

Écrire un module qui reçoit une liste de chaines en paramètre et supprime de cette liste tous les éléments de valeur donnée en paramètre. L'algorithme retournera le nombre de suppressions effectuées.

9 Éliminer les doublons d'une liste

Soit une liste **ordonnée** d'entiers avec de possibles redondances. Écrire un module qui enlève les redondances de la liste.

Exemple : Si la liste est (1, 3, 3, 7, 8, 8, 8), le résultat est (1, 3, 7, 8).

a) Faites l'exercice en créant une **nouvelle liste** (la liste de départ reste inchangée)

b) Refaites l'exercice en **modifiant** la liste de départ (pas de nouvelle liste)

10 Rendez-vous

Soit la structure « RendezVous » composée d'une date et d'un motif de rencontre. Écrire un module qui reçoit une liste de rendez-vous et la met à jour en supprimant tous ceux qui sont désormais passés.

Chapitre 4

Représentation des données