Chap. 4. Quelques structures de données : PILES, FILES, MATRICES et LISTES

I. Structures de données : généralités

- Cette notion est capitale en programmation. Les structures de données permettent de mettre en mémoire (RAM) les données dans des structures logiques qui rendent leur manipulation aisée. Les données ne peuvent rester sur les périphériques d'archivage (disque dur, clé USB, DVD etc.) car les temps d'accès sont trop longs (lecture/écriture). De plus ils ne sont pas conçu pour cela (bien que cela puisse se faire).

- Les structures intéressantes sont dynamiques : elles grandissent et rétrécissent à volonté.

- A chaque type de structure on adjoint des méthodes (= fonctions ou procédures selon les langages) qui on pour rôle de manipuler les données.

On peut discerner :
Les structures finies : constantes, variables, enregistrements...
Les structures indexées : tableaux à une (=vecteur) ou plusieurs dimensions (équiv. matrices). Le nombre de dimensions peut être très grand, au besoin.
Structures récursives : listes, graphes, arbres.

Les opérations à effectuer sont en général équivalentes à :

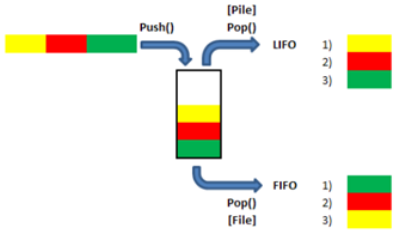
```
search(S, k)          # cherche k dans S et retourne un pointeur sur k
Insert(S,k)           # insère k dans S
Delete (S, k)         # supprime k de S
Successeur (S, k)
Predecesseur (S,k)
Maximum(S) et Minimum (S)
```

Comme le temps requis pour ces opérations dépend de S et de la taille des données, on choisira la structure S en fonction des opérations les plus fréquentes et de la taille des données.

I. Les piles et les files

Piles et Files

- Dans une pile, l'élément supprimé est le dernier inséré
  - LIFO – Last In First Out – Dernier arrivé premier parti
- Dans une file, l'élément supprimé est le plus ancien
  - FIFO – First In First Out – Premier arrivé premier parti



Pour les piles et les files on utilisera la structure de base "tableau".

La pile sera vue comme un empilement vertical d'objets dont seul le premier élément est accessible.

La file sera vue comme un tunnel dans lequel les wagons du train se suivent et dont seul le premier est accessible.

Algorithmes pile et files

```r
### Piles et files#########

push <- function (ajout, pile){
  l <- length(pile)

  for (i in l:1)
     pile[i+1] <- pile[i]
  pile[1] <- ajout
  return(pile)
}

pop <- function ( pile){
  l <- length(pile)
  x <- pile[1]
  for (i in 1:l)
  pile[i] <- pile[i+1]
  pile[l] = 0.0
  return(pile)
}

taille <- function(pile){
  t<-0
  for (i in 1:length(pile)){
    if (pile[i] >= 0)
       t = t+1
  }
  return(t)
}

cree_pile <- function(taille){
 pile <- vector(taille, mode ="integer")
 return(pile)
}


pile <-cree_pile(1)
pile
for (i in 1:5){
n <- round(runif(1, min=5, max=20), digits=2)
pile <-push(n,pile)
}

p <-(taille(pile))
pop(pile)
push(1.23,pile)

#♥-----------------------------------
# exercice : transformer la pile en file

```

```r
################ file #########

push <- function (ajout, myfile){
  l <- length(myfile)

  for (i in l:1)
    myfile[i+1] <- myfile[i]
  myfile[1] <- ajout
  # suppression des na
  myfile <- myfile[! is.na(myfile)]
  return(myfile)
}

pop <- function ( myfile){
  l <- (length(myfile))
  is.na (myfile) <- c(l)
  # suppression des na
  myfile <- myfile[! is.na(myfile)]
  return(myfile)
}

taille <- function(myfile){
  t<-0
  for (i in 1:length(myfile)){
    if (! is.na(myfile[i]))
       t = t+1
  }
  return(t)
}

cree_myfile <- function(taille){
  myfile <- vector(taille, mode ="numeric")
  return(myfile)
}

myfile <-cree_myfile(0) ; myfile
for (i in 1:5){
  n <- round(runif(1, min=5, max=20), digits=2)
  myfile <-push(n,myfile)
}

myfile
cat("longueur de la file : ", taille(myfile))
myfile <-pop(myfile) ; myfile
cat("longueur de la file : ", taille(myfile))
myfile<-push(1.23,myfile) ; myfile
cat("longueur de la file : ", taille(myfile))
#----------------------------------------
```

II. Le cas des matrices (exemples empruntés en partie à Ricco Rakotomalala, http://eric.univ-lyon2.fr/~ricco/cours/slides/tableaux%20et%20matrices%20avec%20r.pdf)

R étant un langage de programmation orienté statistiques, il manipule aisément vecteurs et matrices (algèbre linéaire...). Il présente donc les structures particulières vecteurs et matrices.
Une matrice est un tableau à deux dimensions avec un accès ligne et colonne. On dispose avec R d'opérateurs spécifiques (inversion, déterminant, produit matriciel, etc.). Une matrice à une dimension est un vecteur. Pour créer une matrice on peut opérer soit à partir d'un vecteur, soit en convertissant un dataframe, soit en la créant directement (remarquez bien que le dataframe est une structure de données propre à R !).

A partir d'un vecteur c() :

```r
v <- c(1.2,2.3,4.1,2.5,1.4,2.7)
m <- matrix(v,nrow=2,ncol=3)
attributes(m)
print(m)
```

```
> attributes(m)
$dim
[1] 2 3

> print(m)
     [,1] [,2] [,3]
[1,]  1.2  4.1  1.4
[2,]  2.3  2.5  2.7
```

• Connaître les nombres de lignes et de colonnes avec les commandes nrow() et ncol()

```
> #nombre de lignes
> print(nrow(m))
[1] 2
> #nombre de colonnes
> print(ncol(m))
[1] 3
```

Création de matrice à partir d'un dataframe.

```
d <-read.table("http://sites.unice.fr/coquillard/UE7/tomates.txt", header = TRUE, sep = "\t")
dm <- as.matrix(d)
dm
dim(dm)
```

```
> dim(dm)
[1] 30  6
```

```
> dm
      X    T X0.1mg X0.2mg X0.6mg X0.8mg
 [1,]  1 27.6   12.7   19.2    5.2   13.4
 [2,]  2 26.6   22.6   13.0    7.6    8.6
 [3,]  3 19.9   13.2   10.9   11.9    5.4
 [4,]  4 22.9   20.2   11.0   10.1    4.7
 [5,]  5 16.2   19.0    8.3    5.5   10.9
 [6,]  6 24.6   22.3   13.0    7.3    5.8
 [7,]  7 26.4   20.8   16.5    1.1   10.1
 [8,]  8 20.1   19.1   20.2   12.9    4.7
 [9,]  9 23.2   12.1   17.6    8.7    4.4
[10,] 10 31.8   14.9   15.7    6.9    4.6
[11,] 11 26.1   13.8   11.7    4.6   10.6
[12,] 12 23.1   16.0   12.7    8.8    2.8
[13,] 13 23.9   23.0    5.0    4.0    6.1
[14,] 14 22.1   22.4    7.3    8.6   11.9
[15,] 15 28.4   16.9    9.2   11.9   15.5
[16,] 16 22.3   13.0   16.8   12.9    3.3
[17,] 17 22.4   18.6   14.3    6.1   15.2
[18,] 18 28.3   11.5   11.4    8.8    6.5
[19,] 19 30.4   17.7   12.5    1.8    7.1
[20,] 20 17.6   17.5   16.1   12.6    5.4
[21,] 21 26.2   19.9   13.4    6.5    4.7
[22,] 22 30.1   20.6    6.7    5.7    2.7
[23,] 23 21.4   25.9    9.1    5.7   11.1
[24,] 24 30.4   15.8    8.3   10.9    8.3
[25,] 25 28.7   14.8   10.8   10.0    8.4
[26,] 26 19.6   21.7   10.1    3.5    9.1
[27,] 27 27.5   19.3   15.5    9.4   14.4
[28,] 28 23.5   23.3    8.0    9.0    7.7
[29,] 29 28.8   20.5   15.8   13.9    9.0
[30,] 30 22.4   22.6    8.8    5.9    9.0
```

Manipulations élémentaires

Un élément.....	m[1,2] # renvoie 4.1
Somme de la colonne 3.....	m[3] # renvoie 4.1 (on peut linéariser l'accès, il décompte par colonne) !!!
Ligne 1.....	m[1,] # renvoie le *vecteur* (1.2, 4.1, 1.4) !!!
Colonne 1.....	m[,1] # renvoie le *vecteur* (1.2, 2.3) !!!
Colonnes 2 et 3.....	m[,2:3] # renvoie une matrice composée des colonnes 2 et 3 de m

Création "ex-nihilo"

```
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
               dimnames = list(c("row1", "row2"),
                               c("Col1", "Col2", "Col3")))
mdat
```

```
> mdat
     Col1 Col2 Col3
row1    1    2    3
row2   11   12   13
>
```

Attribuer et manipuler par des noms

```
> print(m)
     [,1] [,2] [,3]
[1,]  1.2  4.1  1.4
[2,]  2.3  2.5  2.7
> #nommage
> rownames(m) <- c("pierre","paul")
> colnames(m) <- c("x1","x2","x3")
> print(m)
       x1  x2  x3
pierre 1.2 4.1 1.4
paul   2.3 2.5 2.7
>
> #intérêt ? accès par nom
> print(m["pierre",])
 x1  x2  x3
1.2 4.1 1.4
> print(m[,c("x1","x3")])
       x1  x3
pierre 1.2 1.4
paul   2.3 2.7
```

On peut attribuer des noms aux lignes et aux colonnes avec **rownames()** et **colnames()**

On peut s'appuyer sur les noms pour accéder au contenu de la matrice

Modification dynamique des dimensions de la matrice.

On utilise les fonctions cbind() pour les colonnes et rbind() pour les lignes. Un warning est affiché si les dimensions ne correspondent pas (la ligne additionnelle est trop grande ou trop petite).

```
> v <- c(1.2,2.3,4.1,2.5,1.4,2.7)
> m <- matrix(v,nrow=2,ncol=3)
> print(m)
     [,1] [,2] [,3]
[1,]  1.2  4.1  1.4
[2,]  2.3  2.5  2.7
> #extraction de la 3e colonne
> z <- m[,3]
> print(z)
[1] 1.4 2.7
> #adjonction de cette colonne à m
> w <- cbind(m,z)
> print(w)
     [,1] [,2] [,3]  z
[1,]  1.2  4.1  1.4 1.4
[2,]  2.3  2.5  2.7 2.7
> #un autre vecteur de 4 valeurs
> s <- c(1.1,2.0,1.4,0.5)
> u <- rbind(w,s)
> print(u)
     [,1] [,2] [,3]  z
     1.2  4.1  1.4 1.4
     2.3  2.5  2.7 2.7
s    1.1  2.0  1.4 0.5
```

Opérateurs

On dispose d'opérateurs propres aux matrices (à gauche) et communs (à droite) :

```
> print(m)
     [,1] [,2] [,3]
[1,]  1.2  4.1  1.4
[2,]  2.3  2.5  2.7
> #transposée
> tm <- t(m)
> print(tm)
     [,1] [,2]
[1,]  1.2  2.3
[2,]  4.1  2.5
[3,]  1.4  2.7
>
> #produit matriciel
> p <- tm %*% m
> print(p)
      [,1]  [,2]  [,3]
[1,]  6.73 10.67  7.89
[2,] 10.67 23.06 12.49
[3,]  7.89 12.49  9.25
>
> #valeurs et vect. propres
> print(eigen(p))
$values
[1]  3.632417e+01  2.715828e+00 -2.083170e-15

$vectors
           [,1]       [,2]         [,3]
[1,] -0.4078975  0.5027224  0.762161303
[2,] -0.7780304 -0.6282234 -0.002013636
[3,] -0.4777953  0.5938060 -0.647384039
```

```
> print(m)
     [,1] [,2] [,3]
[1,]  1.2  4.1  1.4
[2,]  2.3  2.5  2.7
> #création de z
> #multiplication par un scalaire
> z <- 2 * m
> print(z)
     [,1] [,2] [,3]
[1,]  2.4  8.2  2.8
[2,]  4.6  5.0  5.4
>
> #addition élément par élément
> w <- z + m
> print(w)
     [,1] [,2] [,3]
[1,]  3.6 12.3  4.2
[2,]  6.9  7.5  8.1
```

2.4 = 2 * 1.2
8.2 = 2 * 4.1
…

3.6 = 2.4 + 1.2
12.3 = 8.2 + 4.1
…

III. les listes

Une liste est un ensemble d'éléments ordonnés les uns à la suite des autres dans une structure indexée.

Attention, détail important : avec R, ces éléments peuvent être de types différents. L'utilité d'une telle structure est ainsi de regrouper dans un même objet (la liste) une série d'autres objets relevant de caractéristiques communes. Il est même possible (et recommandé parfois) de construire des listes de listes. On peut alors traiter tous les éléments d'une même liste de la même façon, ou bien même en une seule fois (un ligne de code) en appelant des fonctions dédiées à ces opérations.

D'un point de vue technique, la liste est composée d'éléments dont chacun possède un attribut et deux pointeurs appelés *prédécesseur* et *successeur*, **chacun pointant respectivement sur l'adresse** de l'élément précédent et du suivant.
Ce qui fait que la liste n'est pas forcément stockée de façon contigüe en mémoire. Autre avantage, il est possible de supprimer un élément quelconque de la liste en raboutant les 2 parties dissociées en faisant correctement pointer les pointeurs sur les bonnes adresses.
Lorsque la liste est composée d'éléments complexes (plusieurs attributs, i.e. tableaux, listes, fonctions etc.), le fait d'incrémenter de 1 ($p = p + 1$) un pointeur sur un élément de la liste ne le déplace pas d'une unité de mémoire *mais de la taille de l'élément pointé*. On navigue ainsi par sauts, directement en mémoire.
Enfin, le fait de passer un pointeur à une fonction est beaucoup plus efficace que de lui passer l'ensemble des valeurs (attributs) de l'élément pointé. Ainsi ces valeurs ne sont-elles pas recopiées dans la pile mémoire de la fonction. D'où un gain énorme d'espace mémoire et de temps d'exécution.

NB : Tous les langages utilisent les pointeurs, mais seuls certain autorisent leur utilisation explicite (C, C++, Pascal, Assembleur, Ada, Fortran...). Les autres (Java, Python,...) effectuent des passages par valeurs aux fonctions, n'autorisant pas la modification de l'original.

Soit à créer une liste de poids moléculaires de la famille des immunoglobulines. Or, on distingue : les IgG, IgA, IgM, IgD et IgE. De plus, à l'intérieur de ces sous-familles, il n'y a pas le même nombre de molécules et toutes n'ont pas le même poids. Admettons que l'on cherche à évaluer pour chacune des sous-familles le poids moyen, la variance etc...Un tableau semble tout indiqué. Mais il vient tout de suite que certaines colonnes n'auront pas le même nombre de ligne (donc un tableau "creux") et nous aurons perdu aussi les regroupements en sous-familles. Il faudrait alors considérer non pas un mais plusieurs tableaux. La solution idéale est la liste. Soient les données suivantes (fictives mais réalistes, en kd) :
IgG : 160, 155, 163, 158
IgA : 160, 220, 395, 257, 180,
IgM : 900, 920, 950, 970, 900, 965
IgD : 180, 182, 185, 181, 184, 181, 178
IgE : 188, 185,189, 190, 183

Pour créer un liste on dispose de la fonction list(). Pour créer la liste Iglob on peut :

- Créer une liste vide puis la remplir par la suite
- Préciser chaque nouvel objet à insérer, en donnant un nom à chaque objet de la liste

Il n'est pas nécessaire de spécifier les noms des éléments lors de la création de la liste. On peut alors indiquer les seuls noms des objets à stocker : liste <- list(objet1, objet2, ..., objetn).

```
Iglob <- list( IgG = c( 160, 155, 163, 158),
               IgA = c(160, 220, 395, 257, 180),
               IgM =c(900, 920, 950, 970, 900, 965),
               IgD =c(180, 182, 185, 181, 184, 181, 178),
               IgE =c(188, 185,189, 190, 183) )

Iglob

# quelques manipulation élémentaires
names(Iglob)
length(Iglob)
Iglob[["IgA"]]
Iglob[2]
Iglob[2:4]

# on peut aussi accéder aux éléments de la liste ainsi,
# où v est un vecteur et pourra ensuite être traité en tant que tel.
# Si on avait construit une liste de matrices, v serait une matrice.
v <- Iglob$IgA


# du calcul
mean(Iglob[["IgA"]])
var(Iglob[["IgA"]])
var(Iglob[[2]])
```

```
$IgG
[1] 160 155 163 158

$IgA
[1] 160 220 395 257 180

$IgM
[1] 900 920 950 970 900 965

$IgD
[1] 180 182 185 181 184 181 178

$IgE
[1] 188 185 189 190 183
```

```
> # quelques manipulation élémentaires
> names(Iglob)
[1] "IgG" "IgA" "IgM" "IgD" "IgE"
> length(Iglob)
[1] 5
> Iglob[["IgA"]]
[1] 160 220 395 257 180
> Iglob[2]
$IgA
[1] 160 220 395 257 180

> Iglob[2:4]
$IgA
[1] 160 220 395 257 180

$IgM
[1] 900 920 950 970 900 965

$IgD
[1] 180 182 185 181 184 181 178
```

```
> # du calcul
> mean(Iglob[["IgA"]])
[1] 242.4
> var(Iglob[["IgA"]])
[1] 8671.3
> var(Iglob[[2]])
[1] 8671.3
```

lapply() Si on souhaite extraire quelques valeurs concernant chacune des sous-familles, nous pourrions répéter les appels ci-dessus pour chacune d'entre elle, mais cela pourrait s'avérer fastidieux dans le cas de listes très longues. On utilisera la fonction lapply(x, FUN,...) où x est la liste à traiter et FUN la fonction que l'on souhaite appliquer à tous les éléments de la liste. les trois points sont des arguments optionnels que prendra la fonction que l'on souhaite appliquer.

```
lapply(Iglob, min, na.rm=T)
lapply(Iglob, length)
lapply(Iglob,mean)
```

```
> lapply(Iglob, min, na.rm=T)
$IgG
[1] 155

$IgA
[1] 160

$IgM
[1] 900

$IgD
[1] 178

$IgE
[1] 183
```

```
> lapply(Iglob, length)
$IgG
[1] 4

$IgA
[1] 5

$IgM
[1] 6

$IgD
[1] 7

$IgE
[1] 5
```

```
> lapply(Iglob,mean)
$IgG
[1] 159

$IgA
[1] 242.4

$IgM
[1] 934.1667

$IgD
[1] 181.5714

$IgE
[1] 187
```

sapply() Dans ce qui précède, les objets renvoyés sont des listes. Or, cela n'est pas forcément souhaitable car certaines fonctions n'acceptent pas de liste en argument (la fonction sort() par exemple). sapply renverra, dans la mesure du possible les résultats sus forme d'objets plus usuels (vecteurs, matrices, tableaux)

```
31 sapply(Iglob,mean)
32 sort(sapply(Iglob,mean))
33 sapply(Iglob,summary)
```

```
> sapply(Iglob,mean)
     IgG      IgA      IgM      IgD      IgE
159.0000 242.4000 934.1667 181.5714 187.0000
>
> sort(sapply(Iglob,mean))
     IgG      IgD      IgE      IgA      IgM
159.0000 181.5714 187.0000 242.4000 934.1667
>
> sapply(Iglob,summary)
            IgG   IgA      IgM      IgD IgE
Min.    155.00 160.0 900.0000 178.0000 183
1st Qu. 157.25 180.0 905.0000 180.5000 185
Median  159.00 220.0 935.0000 181.0000 188
Mean    159.00 242.4 934.1667 181.5714 187
3rd Qu. 160.75 257.0 961.2500 183.0000 189
Max.    163.00 395.0 970.0000 185.0000 190
>
```

Ou encore, un test de corrélation entre deux sous-liste de mêmes longueurs:

```
> cor.test(Iglob$IgE, Iglob$IgA)

        Pearson's product-moment correlation

data:  Iglob$IgE and Iglob$IgA
t = 1.045, df = 3, p-value = 0.3728
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.6719105  0.9609055
sample estimates:
      cor
0.5165963

> test <- cor.test(Iglob$IgE, Iglob$IgA)
> is.list(test)
[1] TRUE
```

Ce qui prouve bien que l'on obtient une liste, dont peut voir la structure avec :

```
> names(test)
[1] "statistic" "parameter" "p.value" "estimate" "null.value" "alternative" "method" "data.name"
"conf.int"
```

Mais aussi avec :

```
> str(test)
List of 9
 $ statistic  : Named num 1.05
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named int 3
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.373
 $ estimate   : Named num 0.517
  ..- attr(*, "names")= chr "cor"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "correlation"
 $ alternative: chr "two.sided"
 $ method     : chr "Pearson's product-moment correlation"
 $ data.name  : chr "Iglob$IgE and Iglob$IgA"
 $ conf.int   : atomic [1:2] -0.672 0.961
  ..- attr(*, "conf.level")= num 0.95
 - attr(*, "class")= chr "htest"
```

Ce qui permet alors d'afficher, ou de sauver dans une variable, par exemple :

```
> test$p.value
[1] 0.3728049
```

Avec le très riche package rlist

```
#-----Avec le package rlist
install.packages(rlist)
library(rlist)

x <- list(a=1,b=2,c=3)
list.append(x,d=4,e=5)
list.append(x,d=4,f=c(2,3))

x <- list(a=1,b=2,c=3)
list.prepend(x, d=4, e=5)
list.prepend(x, d=4, f=c(2,3))

x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
          p2 = list(type='B',score=list(c1=9,c2=9)),
          p3 = list(type='B',score=list(c1=9,c2=7)))

# on insère une liste dans la liste en position 2
list.insert(x, 2, p2.1 = list(type='B',score=list(c1=8,c2=9)))

# on supprime des éléments de la liste
x <- list(p1 = list(type='A',score=list(c1=10,c2=8)),
          p2 = list(type='B',score=list(c1=9,c2=9)),
          p3 = list(type='B',score=list(c1=9,c2=7)))
list.remove(x, 'p1')
list.remove(x, c(1,2))
```

```
> list.append(x,d=4,e=5)
$a
[1] 1

$b
[1] 2

$c
[1] 3

$d
[1] 4

$e
[1] 5
```

```
> list.prepend(x, d=4, e=5)
$d
[1] 4

$e
[1] 5

$a
[1] 1

$b
[1] 2

$c
[1] 3
```

```
> list.remove(x, 'p1')
$p2
$p2$type
[1] "B"

$p2$score
$p2$score$c1
[1] 9

$p2$score$c2
[1] 9


$p3
$p3$type
[1] "B"

$p3$score
$p3$score$c1
[1] 9

$p3$score$c2
[1] 7
```

```
> list.remove(x, c(1,2))
$p3
$p3$type
[1] "B"

$p3$score
$p3$score$c1
[1] 9

$p3$score$c2
[1] 7
```

```
> list.insert(x, 2, p2.1 = list(type='B',score=list(c1=8,c2=9)))
$p1
$p1$type
[1] "A"

$p1$score
$p1$score$c1
[1] 10

$p1$score$c2
[1] 8


$p2.1
$p2.1$type
[1] "B"

$p2.1$score
$p2.1$score$c1
[1] 8

$p2.1$score$c2
[1] 9


$p2
$p2$type
[1] "B"

$p2$score
$p2$score$c1
[1] 9

$p2$score$c2
[1] 9


$p3
$p3$type
[1] "B"

$p3$score
$p3$score$c1
[1] 9

$p3$score$c2
[1] 7
```