Les collections

7

Comme l'indique le nom français "ordinateur", un ordinateur sait ordonner, trier, gérer des collections de données. Comme tout langage de programmation, le langage Eiffel dispose de plusieurs types de collections en fonction de ce qu'on souhaite en faire.

- STRING est une une collection ne permettant de gérer que des caractères.
- TUPLE correspond à de petites collections de taille fixe.
- ARRAY, LINKED_LIST, DICTIONARY sont les principales collections. Elles permettent de stocker n'importe quel type d'objets et sont de taille variable (extensible).
- STREAM sont des classes qui gèrent des flux de données.

1. Chaîne de caractères: STRING

STRING est la classe permettant de gérer et de manipuler des "phrases" appelées communément en informatique des chaînes de caractères.

1.1. Les principales fonctionnalités

Cette liste n'est pas exhaustive mais présente les principales caractéristiques de la classe STRING

1.1.1.Constructeurs

```
make (needed_capacity: INTEGER)
```

crée une chaîne de caractères qui a au moins *needed_capacity* caractères.

```
copy (other: like Current)
```

copie la STRING *other* dans l'objet courant.

```
make_empty
```

créé une chaîne de caractères vide.

```
make_filled (c: CHARACTER; n: INTEGER)
```

crée une chaîne de caractères avec n copies du caractère 'c'.

Exemples:

```
str1, str2, str3 : STRING
......
create str1.make_empty
str1.append("du texte")

create str2.make(10)
str2.put('T',1);str2.put('e ',2); str2.put('x',3); str2.put('t',4)

str3 := "Je suis un objet de classe STRING"
```

La dernière écriture utilisant l'opérateur d'affectation « := » est une facilité qui est équivalente à la

création de la chaîne str1.

1.1.2. Comptabilisation:

```
count : INTEGER
```

pour compter le nombre de caractères composant la chaîne de caractères

```
lower : INTEGER
upper : INTEGER
```

sont les indices de début et de fin de la chaîne de caractères. Cela est utile si on souhaite lire les caractères un par un dans une boucle du début de la chaîne à la fin.

```
from  i := my_string.lower
until i > my_string.upper
loop
    -- do something
    i := i + 1
end
```

```
capacity
```

est le nombre de caractères réservé en mémoire.

1.1.3. Test et comparaison:

Egalité entre deux STRING

```
is_equal
```

Ainsi qu'il l'a été mentionné dans le chapitre 'Classe et Objet', l'égalité entre deux STRING se fait par la méthode *is_equal*.

```
if "bioinformatics".is_equal("bioinformatics") then
    io.put_string("True%N")
end
```

Les inégalités sont aussi disponibles pour les STRING

```
 >, > , >=, <=
```

Ces inégalités font en fait la somme des codes ASCII des caractères présents dans chacune des STRING et comparent ces nombres:

```
if "bio" > "Bio" then
    io.put_string("True%N")
end
```

On peut le vérifier en faisant afficher la somme de "bio" et "Bio"

```
if "bio" > "Bio" then
    io.put_string("True " +
                ('b'.code + 'i'.code + 'o'.code).out + " > " +
                ('B'.code + 'i'.code + 'o'.code).out +
                "%N")
end
```

Le résultat est:

```
True 314 > 282
```

On peut aussi tester le contenu d'une STRING pour savoir si c'est un nombre décimal, hexadécimal ou. Par exemple:

```
is_integer: BOOLEAN
```

teste si **l'objet courant** est un INTEGER. De la même façon, on a:

```
is_bit: BOOLEAN
is_number: BOOLEAN
```

```
is_real: BOOLEAN
```

1.1.4. Lecture:

Pour accéder aux caractères d'une STRING, plusieurs méthodes sont disponibles.

```
item(index : INTEGER) : CHARACTER
```

permet de lire le index^{ème} caractère de **Current** (la STRING courante).

Pour des accès en début ou en fin de STRING, les méthodes suivantes sont très pratiques.

```
first : CHARACTER
last : CHARACTER
```

1.1.5. Ecriture:

```
append
```

concatène une string à la fin de **Current**

```
add_last(ch : CHARACTER)
```

ajoute un caractère à la fin de **Current**

1.1.6. Recherche/ remplacement:

```
index_of (c: CHARACTER; start_index: INTEGER): INTEGER
```

retourne l'indice de la première occurrence du caractère 'c' trouvée à partir de l'indice start_index. S'il n'y en a pas, la fonction retourne 0.

```
reverse_index_of (c: CHARACTER; start_index: INTEGER): INTEGER
```

Même chose que *index_of* mais cette fois la recherche se fera en partant de start_index en rebroussant chemin vers le début de la chaîne de caractères.

```
has (c: CHARACTER): BOOLEAN
```

Retourne True (vrai) si le caractère 'c' est présent dans la chaîne de caractères.

```
has_substring (other: STRING): BOOLEAN
```

Retourne True (vrai) si la sous-chaîne de caractères other est présente dans la chaîne de caractères.

```
occurrences (c: CHARACTER): INTEGER
```

Retourne le nombre de fois que le caractère 'c' apparaît dans la chaîne de caractères.

substring(i1, i2 : INTEGER) extrait une sous-chaîne de caractères comprise entre i1 et i2

1.1.7. Suppression:

```
right_adjust
left_adjust
```

supprime tous les espaces inutiles à droite et à gauche, respectivement.

```
remove_first (n: INTEGER)
```

supprime les n premiers caractères. Si n >= count, tout est supprimé.

```
remove_last (n: INTEGER)
```

supprime les n derniers caractères. Si n >= count, tout est supprimé.

```
remove_substring (start_index, end_index: INTEGER)
```

supprime tous les caractères compris entre strt_index to et_index inclus.

1.2. Exemple : SEQUENCE

En bioinformatique, l'objet qui se rapproche le plus d'une chaîne de caractères est la séquence primaire habituellement donnée sous forme d'une suite de codes à une lettre. Dans cet exemple, on se propose d'implanter une classe SEQUENCE qui comprend (i) un attribut *data* pour contenir la séquence et (ii) un nombre qui nous indiquera quelle est la nature de la séquence (protéique, nucléique: ADN, ARNm, ADNc,etc.).

Remarque: On pourrait déterminer automatiquement la nature de la séquence d'après sa composition en lettres mais cela dépasse le cadre de cet exemple. Vous pouvez parfaitement essayé de le faire mais faire attention aux nucléotides qui peuvent être définis différemment selon les tables utilisées (pour plus de détail, voir http://www.infobiogen.fr/doc/documents.php?cours=tabaanuc)

```
class SEQUENCE
creation
    with

feature {ANY} -- Constructor

    with_protein(sequ : STRING) is
          do
                flag := Protein
                data := sequ
          end

feature {ANY}

    Unknown : INTEGER is 0
    Protein : INTEGER is 1
    Nucleic : INTEGER is 2
    Dna : INTEGER is 4
    Mrna : INTEGER is 8
    Cdna : INTEGER is 16

feature {ANY} -- Access

    to_staden : STRING is
          -- Display sequence in Staden usual format
          do
                Result := data
          end

feature {NONE}
    flag : INTEGER
    data : STRING
end -- end of class SEQUENCE
```

Une classe TEST pour utiliser notre nouvelle classe SEQUENCE

```
class TEST
creation
    make
feature {ANY}
    make is
          do
```

```
                create a_sequ.with(
                "[
                SESLRIIFAGTPDFAARHLDALLSSGHNVV
                GVFTQPDRPAGRGKKLMPSPVKVLAEEKGL
                PVFQPVSLRPQENQQLVAELQADVMVVVAY
                ]")
                io.put_string(a_sequ.to_staden)
                io.put_new_line
        end
end -- end of class TEST
```

Après compilation et exécution de test, la séquence s'affiche:

```
SESLRIIFAGTPDFAARHLDALLSSGHNVV
GVFTQPDRPAGRGKKLMPSPVKVLAEEKGL
PVFQPVSLRPQENQQLVAELQADVMVVVAY
```

Exercice: Selon le même principe, ajoutez un constructeur *with_nucleic_acid(a_sequ : STRING)* pour prendre en compte les séquences nucléiques

1.3. Routines utiles pour SEQUENCE

On peut compléter par exemple la classe SEQUENCE par une fonction:

- calculant la longueur:

```
length : INTEGER is
        do
                Result := data.count
        end
```

- calculant la fréquence d'un résidu ou d'un nucléotide.

```
frequency_of(a_char : CHARACTER) : DOUBLE is
        do
                Result := data.occurences(a_char) /data.count
        end
```

- changeant la casse des caractères pour les séquences nucléiques (minuscules) et protéiques (majuscules).

```
to_staden : STRING is
        do
                if flag = Protein then
                        Result := data.to_upper
                elseif flag >= Nucleic then
                        Result := data.to_lower
                else
                        -- flag = Unknown
                        Result := "Unknown sequence"
                end
        end
```

1.4. Constructeur plus complet

On peut perfectionner le constructeur de manière que tous les formats de séquence habituellement rencontrés en bioinformatique[1] puissent être acceptés. Mais, il faudra impérativement que l'attribut *data* de SEQUENCE stocke la séquence au format Staden[2]. Il faudra donc modifier le constructeur

1 Source: http://www.infobiogen.fr/doc/documents.php?cours=formats

2. Format Staden. Le plus ancien et le plus simple : La séquence est écrite sous forme d'une suite de lettres disposées en lignes terminées par un retour-à-la-ligne (80 caractères max/ligne). Exemple :

pour prendre en compte ces différents formats et convertir la chaîne en format Staden avant de l'affecter à *data*. Voici un exemple avec le format FASTA et DNS Strider.

1.4.1. Format Fasta

Dans ce cas, la séquence (donnée sous forme de lignes de 80 caractères maximum) est précédée d'une ligne de titre (nom, définition ...) qui doit commencée par le caractère ">". Cela permet de mettre plusieurs séquences dans un même fichier.

```
>em|U03177|FL03177 Feline leukemia virus clone FeLV-69TTU3-16.
AGATACAAGGAAGTTAGAGGCTAAAACAGGATATCTGTGGTTAAGCACCTG
TGAGGCCAAGAACAGTTAAACCCCGGATATAGCTGAAACAGCAGAAGTTTC
GCCAGCAGTCTCCAGGCTCCCCA
```

Ajoutez les lignes suivantes dans *make*:

```
    if a_sequ.first = '>' then
          from_fasta(a_sequ)
    end
...
```

Puis, ajoutez la routine suivante après *to_staden*:

```
from_fasta(a_sequ : STRING) is
    local
          first_index, end_index : INTEGER
    do
          first_index := a_sequ.lower
          end_index := a_sequ.index_of('%N',first_index)
          data := a_sequ
          data.remove_substring(first_index,end_index)
    end
```

1.4.2. Format DNA STrider

Trois lignes de commentaires précédées du caractère ";". Les lignes suivantes contiennent la séquence. La dernière ligne doit contenir les caractères "//".

```
; ### from DNA Strider ;-)
; DNA sequence  pir:ccho (1-104) , 104 bases, 7DA79498 checksum.
;
GDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGFTYTD
ANKNKGITWKEETLMEYLENPKKYIPGTKMIFAGIKKKTEREDLIAYLKK
ATNE
//
```

```
do
...
    elseif a_sequ.first = ';' then
          from_dna_strider(a_sequ)
    end
end
```

```
SESLRIIFAGTPDFAARHLDALLSSGHNVVGVFTQPDRPAGRGKKLMPSPVKVLAEEKGL
PVFQPVSLRPQENQQLVAELQADVMVVVAYGLILPKAVLEMPRLGCINVHGSLLPRWRGA
APIQRSLWAGDAETGV
```

```
from_dna_strider(a_sequ : STRING) is
    local
          first_index, end_index : INTEGER
    do
          first_index := a_sequ.reverse_index_of(';',a_sequ.upper)
          end_index := a_sequ.substring_index("//",a_sequ.lower)
          data := a_sequ.substring(first_index + 2,end_index – 1)
    end
```

La fonction *substring* permet d'extraire entre les bornes min_index et max_index de la STRING d'origine, une nouvelle STRING. Il suffit donc de chercher la position du dernier point-virgule dans la STRING ce qui est réalisé avec la fonction *reverse_index_of*. De la même manière, on recherche avec la fonction *substring_index*, la STRING composée des deux caractères "barre oblique" "//". Enfin, il faut ajuster les deux bornes pour ne prendre en compte que la séquence, le début de la séquence est à deux caractères (un saut de ligne '%N' puis le premier caractère) du point-virgule et la fin de la séquence se termine un caractère avant le "//".

Exercice: Améliorez les deux routines précédentes (*from_fasta* et *from_dna_strider*) pour supprimer tous les sauts de ligne dans *data*. De plus, modifiez la fonction *to_staden* pour ajouter des sauts de ligne tous les 80 caractères.

Exercice: Implantez une routine *from_staden* qui accepte un format Staden plus "souple" avec une séquence dans laquelle on trouve tout type de caractères (espace, tabulation, tiret, nombre). Pour l'implantation, utilisez une boucle qui teste les caractères un par un.

```
  1 VLSPADKTNV KAAWGKVGAH AGEYGAEALE RMFLSFPTTK TYFPHFDLSH
 51 GSAQVKGHGK KVADALTNAV AHVDDMPNAL SALSDLHAHK LRVDPVNFKL
101 LSHCLLVTLA AHLPAEFTPA VHASLDKFLA SVSTVLTSKY R
```

Exercice: Implantez une fonction *complementary_strand* qui retourne une STRING correspondant au brin complémentaire d'une séquence nucléique. La convention d'écriture veut qu'une séquence nucléique doit toujours se lire dans le sens 5' vers 3'.

```
complementary_strand : STRING is
    do
    end
```

Exercice: Implantez une procédure *statistics* qui affiche le pourcentage de toutes les lettres présentes dans la séquence et classées par ordre alphabétique.

1.5. Cas pratique... un peu plus sophistiqué

Dans les fichiers au format Protein DataBank (PDB), les références bibliographiques (dont seul le cas du titre est traité ici) sont mises en forme de la façon suivante:

```
JRNL        TITL   STRUCTURE OF HUMAN OXYHAEMOGLOBIN AT 2.1 ANGSTROMS   1HHOA  2
JRNL        TITL 2 RESOLUTION                                           1HHOA  3
```

La position des mots clés dans la ligne est fixe et est décrite dans les spécifications du format, il faut donc découper la ligne en morceaux. Pour une meilleure compréhension, ajoutons une réglette:

```
0        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
|-------+---------+---------+---------+---------+---------+---------+---------|
```

```
JRNL        TITL   STRUCTURE OF HUMAN OXYHAEMOGLOBIN AT 2.1 ANGSTROMS   1HHOA  2
JRNL        TITL 2 RESOLUTION                                           1HHOA  3
```

On souhaite extraire de ces deux lignes, le titre de l'article. Ce qui se fait en testant le descripteur "JRNL" en début de ligne (non montré dans cette implantation) , puis on recherche le deuxième mot clé "TITL" avant de lire la portion de ligne comprise entre les caractères 20 à 70. De plus, il faudra tester si le titre est réparti sur plusieurs lignes et concaténer les différentes parties.

```
1    if line.substring(13,16).is_equal("TITL") then
2          if line.substring(17,18).is_integer then
3                title.append(line.substring(20,70) )
4                title.right_adjust
5                title.add_last(' ')
6          else
7                create title.copy(line.substring(20,70) )
8                title.right_adjust
9                title.add_last(' ')
10         end
11   end
```

Le synopsis de la routine précédente est:

Ligne 1: On teste si l'objet *line* contient le mot clé TITL

Ligne 2: Si OUI, on teste la présence d'un INTEGER en position [17-18]

Lignes 7-9: Si NON, on crée un objet *title* qu'on initialise avec la sous-STRING en position [20-70]. De plus, on élimine tous les caractères espace inutiles à la fin sauf un.

Lignes 3-5: Si OUI, on a affaire à un titre sur plusieurs lignes, on concatène la $n^{ième}$ ligne avec l'objet *title* existant.

Les routines de STRING utilisées sont:

is_equal permet de comparer deux STRING entre elles.

Remarque: La comparaison des objets (classes de type référence) entre eux se fait au moyen de la routine is_equal; A l'EXCEPTION, des classes de type expanded (REAL, INTEGER, CHARACTER) qui utilisent "=".

substring(i1, i2 : INTEGER) extrait une sous-chaîne de caractères comprise entre i1 et i2

copy est un constructeur qui permet de copier une STRING dans **Current**

append concatène une string à la fin de **Current**

add_last(ch : CHARACTER) ajoute un caractère à la fin de **Current**

right_adjust supprime tous les espaces inutiles à droite.

is_integer: BOOLEAN: teste si **Current** est un INTEGER

L'objet *title* contient ainsi la ligne suivante.

```
STRUCTURE OF HUMAN OXYHAEMOGLOBIN AT 2.1 ANGSTROMS RESOLUTION
```

2. UNICODE_STRING

<A FAIRE>

3. TUPLE

Les TUPLEs sont des petites collections (« fourre-tout » ou *n-uplet* en informatique) d'objets divers. Par exemple, TUPLE[INTEGER,REAL,STRING] est un triplet contenant une suite composée d'un INTEGER, puis d'un REAL et d'un STRING. Contrairement aux autres collections, les TUPLEs possèdent une syntaxe particulière pour leur création utilisant des crochets « [ ] ».

```
property_of(a_residu : CHARACTER) : TUPLE[STRING,REAL] is
      -- Retourne le code à 3 lettres du résidu et sa masse
do
      -- Implémentation plus que partielle puisque seul le cas de Ala
      -- est traité :-)
      if a_residu = 'A' then
            Result := [ "ALA", 89.09 ]
      else
            Result := ["XXX",110.0 ]
      end
end
```

Pour l'utilisation de ces TUPLEs, on peut accéder aux divers éléments constituant le TUPLE par first, second, third,...

```
.......
make is
local
      tup : TUPLE[STRING,REAL]
do
      tup := property_of('A')
      io.put_string(tup.first + « » + tup.second.out + "%N")
end
```

4. Des tableaux, en veux-tu? en voilà!

L'utilisation des collections se fait en trois temps:

(i) On déclare la collection souhaitée en plaçant entre crochets, le(s) type(s) d'objet(s) contenus dans la collection. Ex: ARRAY[INTEGER], LINKED_LIST[REAL] sont des collections contenant respectivement des INTEGER et des REAL.

(ii)On doit réserver de la place mémoire à la collection en créant (instanciant) la collection au moyen de la commande **create**.

(iii)On remplit la collection.

4.1. ARRAY

Les ARRAYs sont des tableaux à une seule dimension qui peuvent être extensibles en taille. De plus, l'indexation du tableau peut être quelconque (on peut envisager des tableaux dont le premier objet est stocké à l'indice -4).

L'utilisation des ARRAYs est relativement similaire à celle des STRINGs; en particulier, pour les routines d'accès en lecture et écriture.

4.1.1. Exemple

La routine *make* initialise et remplit le tableau *sequ* de classe ARRAY :qui contient divers masses d'acides aminés. La première étape (ligne 4) est de créer une instance de la classe ARRAY[INTEGER] (l'objet) contenant trois cellules et dont l'indice de la première cellule est 1. Ensuite, on remplit le tableau (lignes 7, 8 et 9) par la méthode *put* en donnant comme arguments la valeur à placer dans la cellule du tableau et l'indice de la cellule. Le programmeur a oublié (?) une valeur (ligne 11) et il force l'extension du tableau pour y mettre sa quatrième valeur dans la cellule d'indice 4.

```
1   make is
2         -- Initialisation du tableau des masses des résidus
3   local
4         sequ :ARRAY[STRING] — Déclaration du tableau
5   do
6         create sequ.make(1,3) -- On créé un tableau de 3 éléments
7         sequ.put("GLU",1)
8         sequ.put("ASP", 2)
9         sequ.put("PHE",3)
10        io.put_string(sequ.item(2) ) -- ASP
11  end
```

De la même façon, qu'il existe une facilité de création des STRING, on peut créer et initialiser un tableau de type ARRAY avec des double chevrons "<<" et ">>". Le tableau commence obligatoirement à l'indice 1 et les divers éléments du tableau doivent être séparés par des virgules. La routine init_weights devient alors:

```
    init_weights is
          -- Initialisation du tableau des masses des résidus
    local
          weights :ARRAY[REAL] — Déclaration du tableau
    do
          -- Tableau commence à l'indice 1
```

```
        weights := << 89.9, 121.16, 133.10,3 >>

        io.put_real(weights.item(1) )  -- = 89.9
    end
```

Attention: Lorsque vous utilisez un tableau d'objet de classe de type référence, vous devez d'abord créer (instancier) l'objet avant de le placer dans le tableau.

Exemple:

```
    sequences : ARRAY[SEQUENCE]
    ......
    create sequences.make(1,3)

    create seq1.with(Protein,"AERTY")
    sequences.put(seq1,1)
    sequences.put(create {SEQUENCE}.with(Protein,"YYYAGH") , 2)
```

Quelques routines bien pratiques ... pour l'accès aux cellules du tableau

```
lower : INTEGER
    indice de début du tableau.

upper : INTEGER
    indice de fin du tableau.

count: INTEGER
    nombre de cellules dans le tableau.

item(index : INTEGER) : T
    Retourne le contenu de la cellule d'indice index.

put(value : T; index : INTEGER)
    Place l'objet value dans la cellule d'indice index
```

Exemple d'utilisation

```
print_contents(arr : ARRAY[STRING]) is
local
    i : INTEGER
do
    from i := arr.lower
    until i > arr.upper
    loop
        if arr.item(i) /= Void then
            io.put_string("#" + i.out + arr.item(i) );
            io.put_new_line
        else
            io.put_string("#" + i.out + "Void%N" )
        end
        i := i + 1
    end
end
```

Quelques routines bien pratiques ... pour rechercher des éléments du tableau.

ARRAY....une collection redimensionnable

Il est possible d'étendre la taille d'un ARRAY en utilisant la routine *add_last*, *add_first* et *force*.

```
create sequ.make(1,3)
sequ.put("SER", 1)
sequ.put("PHE", 2)
sequ.put("THR", 3)

sequ.add_last("ALA") -- SER PHE THR ALA
sequ.add_first("TRP") -- TRP SER PHE THR ALA
sequ.force("PRO",5) — TRP SER PHE THR ALA PRO
```

ARRAY: de drôles de constructeurs

Une des erreurs fréquentes est la mauvaise utilisation des constructeurs *make* et *with_capacity*. Voici un exemple typique où on crée un tableau de quatre éléments et on souhaite ajouter un à un les objets (ici des STRINGs) dans le tableau.

```
make is
    local
          arr : ARRAY[STRING]
    do
          create arr.make(1,4) -- four cells are created
          arr.add_last("SER")
          arr.add_last("PHE")
          arr.add_last("THR")
          print_contents(arr)
    end
end
```

Le résultat est:

```
#1 Void
#2 Void
#3 Void
#4 Void
#5 SER
#6 PHE
#7 THR
```

Dans cet exemple, on espère créer un tableau *arr* vide et y ajouter successivement SER, PHE, THR en positions respectives 1, 2 et 3. Or, le constructeur *make* crée un tableau vide ET réserve de la place pour y stocker les objets, ce qui signifie que *arr.count* est égal à 4 et donc l'utilisation de *arr.add_last* provoque l'extension du tableau *arr* à partir de la position 4 + 1 = 5.

Pour obtenir le résultat escompté, il faut utiliser le constructeur *with_capacity* qui crée un tableau vide sans réserver de place pour stocker les objets.

L'exemple doit être corrigé de la façon suivante:

```
make is
    local
          arr : ARRAY[STRING]
    do
          create arr.with_capacity(4,1) -- four cells are created
          arr.add_last("SER")
          arr.add_last("PHE")
          arr.add_last("THR")
```

```
            print_contents(arr)
        end
    end
```

Le résultat est:

```
#1 SER
#2 PHE
#3 THR
```

Attention!! : Pour une raison obscure, l'ordre et la nature des arguments dans les constructeurs *make* et *with_capacity* est DIFFERENTE. Pour *make*, on donne d'abord l'indice inférieur puis l'indice supérieur. Pour *with_capacity*, on donne d'abord la capacité du tableau (le nombre total d'objets dans le tableau) puis l'indice inférieur.

Un des moyens de s'en souvenir est de regarder le nombre d'objets dans le tableau en utilisant la méthode *count*.

```
make is
    local
        arr1, arr2 : ARRAY[STRING]
    do
        create arr1.with_capacity(4,1) -- four cells are created
        create arr2.make(1,4) -- four cells are created
        io.put_integer(arr1.count) -- count = 0
        io.put_integer(arr2.count) -- count = 4
    end
```

Il est donc conseillé – au moins dans un premier temps – d'utiliser le constructeur *make* avec la méthode *put*, et le constructeur *with_capacity* avec *add*, *add_first*, *add_last*, *force*.

Les autres ARRAY....

Les classes de tableaux sont regroupés dans le *cluster* lib/storage/collection. On y trouve:

1. ARRAY
2. FAST_ARRAY est identique à ARRAY à l'exception commence obligatoirement à l'indice 0 et qui permet des accès aux cellules du tableau beaucoup plus rapide que ARRAY.
3. RING_ARRAY fonctionne comme une pile c'est à dire que les accès au début et à la fin du tableau ont été optimisés.

4.2. QUEUE et STACK

4.3. LINKED_LIST

Les LINKED_LISTs s'utilisent de la même façon que les ARRAYs mais par contre, sont plus efficaces pour des insertions et déletions au milieu du tableau. La traversée du début à la fin du tableau est plus performante que de la fin au début.

Les classes de tableaux sont regroupés dans le *cluster* lib/storage/collection. On y trouve:

LINKED_LIST

TWO_WAY_LINKED_LIST

DICTIONARY

Attention: Depuis SmartEiffel version 2.0, il existe deux implantations de DICTIONARY: HASHED_DICTIONARY et AVL_DICTIONARY (*algorithme de Adelson-Velsky and Landis; 1962*)

Les DICTIONARY aussi appelés tableaux à indexation ou « tables de hachage » sont des collections utilisant une clé unique pour accéder directement au contenu d'une cellule de la table. Leur déclaration est donc composée du type de dictionnaire HASHED_DICTIONARY ou AVL_DICTIONARY suivi de la classe correspondant aux objets stockés et d'une deuxième classe indiquant la nature de la clé. Par exemple, le dictionnaire *three_to_one* contient des objets de type CHARACTER et des clés de type STRING:

```
three_to_one : HASHED_DICTIONARY[CHARACTER, STRING]
```

Pour créer et remplir un dictionnaire, on peut utiliser le constructeur *make* et la méthode *put* de la même façon que pour les ARRAY à la différence près que l'indice est remplacé par la clé.

```
local
    three_to_one : HASHED_DICTIONARY[CHARACTER, STRING]
do
...
create three_to_one.make
three_to_one.put('A', "ALA")
...
```

ou utiliser la création "manifeste" suivante:

```
three_to_one := { HASHED_DICTIONARY[CHARACTER, STRING]
          <<
          'A', "ALA"; 'C', "CYS"; 'D', "ASP"; 'E', "GLU";
          'F', "PHE"; 'G', "GLY"; 'H', "HIS"; 'I', "ILE";
          'K', "LYS"; 'L', "LEU"; 'M', "MET"; 'N', "ASN";
          'P', "PRO"; 'Q', "GLN"; 'R', "ARG"; 'S', "SER";
          'T', "THR"; 'V', "VAL"; 'W', "TRP"; 'Y', "TYR"
          >>
          }
```

Pour trouver l'objet associé à une clé, on utilise la méthode at et pour trouver une clé correspondant à un objet donné la méthode key_at.

```
three_to_one.at("CYS") -- returns the value 'C' with key "CYS"

three_to_one.key_at('Q') -- returns the key GLN of the value Q
```

Application: conversion de séquence de code à trois lettres en code à une lettre

La méthode *three_to_one* convertit une séquence contenant des codes à trois lettres séparées par des espaces en une STRING de code à une lettre:

```
make is
    do
          io.put_string(three_to_one("ALA ARG GLY THR TYR") + "%N")
    end
```

Le programme doit retourner la STRING suivante:

```
ARGTY
```

L'implantation de la routine *three_to_one* est réalisée au moyen d'un dictionnaire:

```
three_to_one(sequ : STRING) : STRING is
    local
          dict : HASHED_DICTIONARY[CHARACTER, STRING]
          words : ARRAY[STRING]
          i : INTEGER
    do
          dict := { HASHED_DICTIONARY[CHARACTER, STRING]
                    <<
                    'A', "ALA"; 'C', "CYS"; 'D', "ASP"; 'E', "GLU";
                    'F', "PHE"; 'G', "GLY"; 'H', "HIS"; 'I', "ILE";
                    'K', "LYS"; 'L', "LEU"; 'M', "MET"; 'N', "ASN";
                    'P', "PRO"; 'Q', "GLN"; 'R', "ARG"; 'S', "SER";
                    'T', "THR"; 'V', "VAL"; 'W', "TRP"; 'Y', "TYR"
                    >>
                    }
          create Result.make_empty
          words := sequ.split -- Convert the STRING in ARRAY[STRING]
          from i := words.lower
          until i > words.upper
          loop
                Result.append_character(dict.at(words.item(i) ) )
                i := i + 1
          end
    end
```

Exercice: Implantez une fonction *one_to_three* qui retourne une STRING contenant la séquence primaire exprimée en code à trois lettres. Pensez à "couper" la chaîne de caractères quand celle-ci fait 80 caractères.

Exemple:

```
io.put_string(sequence.one_to_three("VLSPAD"); io.put_new_line
```

A l'exécution, le résultat doit s'afficher sous la forme suivante:

```
VAL LEU SER PRO ALA ASP
```

Du flux et du reflux

STREAM

TEXT_FILE_READ