Allocation dynamique et liste chaînée
Corrigé

Objectifs

— Comprendre et savoir utiliser les pointeurs

— Savoir implanter une structure de données dynamique simplement chaînée

Exercice 1 . . . . . *1*
Exercice 2 : Liste simplement chaînée *2*
Exercice 3 : Module Liste *11*

Exercice 1 En s'appuyant sur une schématisation des pointeurs, décrire l'effet du programme ci-après. On signalera les instructions erronées et on les ignorera.

```
1  PROCÉDURE Exercice_Pointeurs EST
2
3      --! Définition de types
4      TYPE T_Ptr_Réel EST POINTEUR SUR Réel
5
6      TYPE  T_Complexe EST ENREGISTREMENT
7          Pr : Réel
8          Pi : Réel
9      FIN ENREGISTREMENT
10
11     TYPE T_Ptr_Complexe EST POINTEUR SUR T_Complexe
12
13     --! Déclaration des variables
14     p1 : T_Ptr_Réel
15     p2 : T_Ptr_Réel
16     q1 : T_Ptr_Complexe
17     q2 : T_Ptr_Complexe
18
19 DÉBUT
20     p1 <-- NULL           --  1
21     p2 <-- NEW Réel       --  2
22
23     p1^ <-- 1.1           --  3
24     p2^ <-- 2.2           --  4
25
26     p1 <-- p2             --  5
27
28     ÉCRIRE (p1^)          --  6
29
30     p2^ <-- 3.3           --  7
31     ÉCRIRE (p1^)          --  8
32
```

```
33      q1 <-- NEW T_Complexe --  9
34      q1^.Pr <--p1^         -- 10
35      q1^.Pi <-- p2^        -- 11
36      ÉCRIRE (q1^)          -- 12
37      q2 <-- q1             -- 13
38      ÉCRIRE (q2^.Pi)       -- 14
39
40  FIN Exercice_Pointeurs
```

Solution :

À la fin de l'exécution de cette procédure, on constate que les deux blocs de mémoire alloués dynamiquement n'ont pas été libérés. Si un ramasse-miettes (*garbage collector* en anglais) est disponible, elle sera automatiquement libérée. Dans le cas contraire, cette mémoire est perdue pour le programme car le programmeur ne l'a pas libérée. Elle sera toutefois rendue à la fin de l'exécution du programme puisque toute la mémoire qui lui a été attribuée sera libérée. Mais attention, si cette procédure est appelée régulièrement dans un programme qui s'exécute très longtemps (système d'exploitation d'un serveur, d'un téléphone..., équipement réseau tel qu'un routeur, etc) le programme finira par manquer de mémoire.

Ce qu'il faut retenir de cet exercice :

- L'allocation dynamique consiste à ajouter deux opérations : 1) allouer dynamiquement de la mémoire (`new`) qui retourne l'adresse de la zone mémoire allouée et 2) libérer une zone mémoire qui libère une zone mémoire allouée explicitement.
- Si un ramasse-miettes est disponible, la libération de la mémoire se fait automatiquement et le programmeur n'a pas en s'en soucier.
- En l'absence de ramasse-miettes, le programmeur **doit** libérer la mémoire dynamique **dès** qu'il ne l'utilise plus. La libérer plus tôt est une erreur car il manipulera une zone mémoire libérée dont le contenu ne peut plus être garanti. La libérer plus tard fait que cette mémoire n'est pas disponible pour le système alors qu'elle pourrait l'être. Ceci risque de provoquer un échec d'une future allocation dynamique de mémoire.
- L'intérêt de la mémoire dynamique est qu'elle continue à exister après la fin de l'exécution du sous-programme qui l'a créée contrairement à une variable locale pour laquelle la mémoire est automatiquement libérée quand le sous-programme se termine. Ainsi on a normalement un sous-programme qui alloue de la mémoire dynamique et un autre qui la libère. Si le même sous-programme alloue de la mémoire et la libère, il suffirait d'utiliser une variable locale.

Exercice 2 : Liste simplement chaînée

Considérons le type `T_Liste` définissant une liste chaînée d'entiers.

```
1   TYPE T_Liste EST POINTEUR SUR T_Cellule
2
3   TYPE T_Cellule EST ENREGISTREMENT
4       Élément : Entier       -- Élément rangé dans la cellule
5       Suivante : T_Liste     -- Accès à la cellule suivante
6   FIN ENREGISTREMENT
```

1. Spécifier, tester et implanter les sous-programmes suivants. Par tester, on entend identifier les cas de test à faire et les données de test correspondantes.

On adoptera un style programmation défensive.

Solution : Voici la spécification du module Listes (nous anticipons sur l'exercice suivant) qui comprend les opérations décrites ci-après.

```
1  generic
2      type T_Element is private;
3
4  package Listes is
5
6      type T_Liste is limited private;
7
8      Indice_Error: Exception;     -- Un indice est invalide
9      Element_Absent_Error: Exception;     -- Élément non trouvé
10
11
12     -- Initialiser une liste.
13     procedure Initialiser (Liste: out T_Liste) with
14         Post => Taille (Liste) = 0;
15
16
17     -- Détruire une liste et libérer toutes les ressources qu'elle utilise.
18     -- Une liste détruite ne doit plus être utilisée.
19     procedure Detruire (Liste: in out T_Liste);
20
21
22     -- Ajouter un nouvel élément au début d'une liste.
23     procedure Ajouter_Debut (Liste: in out T_Liste; Element: T_Element) with
24         Post => Taille (Liste) > 0 and then Ieme (Liste, 0) = Element;
25
26
27     -- Retourner le premier élément d'une liste.
28     -- Exception : Element_Absent_Error si la liste est vide
29     function Premier (Liste: in T_Liste) return T_Element;
30
31
32     -- Retourner la taille d'une liste.
33     function Taille (Liste: in T_Liste) return Integer;
34
35
36     -- Afficher les éléments d'une liste en révélant la structure interne.
37     -- Par exemple : -->[1]-->[3]-->[1]-->[2]--E
38     --! Cette opération serait plutôt utilisée à des fin de mise au point et
39     --! pourrait rester locale au module.  Pour l'utilisateur du module
40     --! une procédure qui affiche [1, 3, 1, 2] serait plus utile.
41     generic
42         with procedure Afficher_Element (Element: in T_Element);
43     procedure Afficher (Liste: in T_Liste);
44
```

```
45
46      -- Retourner vrai ssi Element est dans Liste.
47      function Est_Present (Liste: in T_Liste; Element: in T_Element) return Boolean;
48
49
50      -- Suppprimer la première occurrence de Element dans Liste.
51      -- Exception : Element_Absent_Error si l'élément n'est pas trouvé.
52      procedure Supprimer (Liste: in out T_Liste; Element: in T_Element);
53
54
55      -- Insérer un nouvel élément (Nouveau) dans la liste (Liste) après un
56      -- élément existant (Element).
57      -- Exception : Element_Absent_Error si Element n'est pas dans la liste
58      procedure Inserer_Apres (Liste: in out T_Liste; Nouveau, Element: in T_Element);
59
60
61      -- Retourner l'élément à la position Indice dans la Liste.
62      -- Le premier élément est à l'indice 0.
63      -- Exception : Indice_Error si l'indice n'est pas valide
64      function Ieme (Liste: in T_Liste; Indice: in Integer) return T_Element;
65
66
67      -- Supprimer l'élément à la position Indice dans la Liste.
68      -- Le premier élément est à l'indice 0.
69      -- Exception : Indice_Error si l'indice n'est pas valide
70      procedure Supprimer_Ieme (Liste: in out T_Liste; Indice: in Integer);
71
72
73      -- Procédure de test de la liste.
74      generic
75          Un, Deux, Trois : T_Element;    -- Trois éléments différents
76          with procedure Afficher_Element (Element: in T_Element);
77      procedure Tester;
78
79
80  private
81
82      type T_Cellule;
83
84      type T_Liste is access T_Cellule;
85
86      type T_Cellule is
87          record
88              Element: T_Element;
89              Suivante: T_Liste;
90          end record;
91
92      function Cellule_Contenant (Element: T_Element; Liste: in T_Liste) return T_Liste with
93          Post => Cellule_Contenant'Result /= null
94                  and then Cellule_Contenant'Result.all.Element = Element;
```

```
95
96          --! Spécifier ici, dans la partie private, ce sous-programme n'est
97          --! pas accessible des utilisateurs du module mais le sera de tous
98          --! sous-programme du corps du module.
99
100 end Listes;
```

1. Initialiser une liste. Cette liste est alors vide.
   Solution :

```
1  procedure Initialiser (Liste: out T_Liste) is
2  begin
3      Liste := null;
4  end Initialiser;
```

2. Ajouter un entier en début d'une liste.
   Solution :

```
1  procedure Ajouter_Debut (Liste: in out T_Liste; Element: T_Element) is
2  begin
3      Liste := new T_Cellule'(Element, Liste);
4  end Ajouter_Debut;
```

3. Obtenir l'entier en début d'une liste.
   Solution :

```
1  function Premier (Liste: in T_Liste) return T_Element is
2  begin
3      if Liste = null then
4          raise Element_Absent_Error;
5      end if;
6
7      return Liste.all.Element;
8  end Premier;
```

4. Obtenir la taille (le nombre d'entiers) d'une liste (version itérative et version récursive).
   Solution : Version récursive :

```
1  function Taille_Recursive (Liste: in T_Liste) return Integer is
2  begin
3      if Liste = null then
4          return 0;
5      else
6          return 1 + Taille_Recursive (Liste.all.Suivante);
7      end if;
8  end Taille_Recursive;
```

   Version Itérative :

```
function Taille_Iterative (Liste: in T_Liste) return Integer is
    Curseur: T_Liste;            -- pour parcourir les cellules
    Nombre_Cellules: Integer;    -- nombre de cellules parcourues
begin
    Nombre_Cellules := 0;
    Curseur := Liste;
    while Curseur /= null loop
        Nombre_Cellules := Nombre_Cellules + 1;
        Curseur := Curseur.all.Suivante;
    end loop;
    return Nombre_Cellules;
end Taille_Iterative;
```

5. Afficher les entiers d'une liste (version itérative et version récursive). On affichera la liste sous la forme suivante (exemple avec la liste `[1, 3, 1, 2]`) :

```
-->[1]-->[3]-->[1]-->[2]--E
```

Solution : Version récursive :

```
    generic
        with procedure Afficher_Element (Element: in T_Element);
    procedure Afficher_Recursive (Liste: in T_Liste);


    procedure Afficher_Recursive (Liste: in T_Liste) is
    begin
        if Liste = null then
            Put ("--E");
        else
            -- Afficher le premier élément
            Put ("-->[");
            Afficher_Element (Liste.all.Element);
            Put ("]");

            -- Afficher les autres éléments
            Afficher_Recursive (Liste.all.Suivante);
        end if;
    end Afficher_Recursive;
```

Version Itérative :

```
generic
    with procedure Afficher_Element (Element: in T_Element);
procedure Afficher_Iterative (Liste: in T_Liste);


procedure Afficher_Iterative (Liste: in T_Liste) is
    Curseur: T_Liste;
begin
    Curseur := Liste;
    while Curseur /= null loop
```

```
11          Put ("-->[");
12          Afficher_Element (Curseur.all.Element);
13          Put ("]");
14          Curseur := Curseur.all.Suivante;
15      end loop;
16      Put ("--E");
17  end Afficher_Iterative;
```

6. Indiquer si un entier e est présent dans une liste (version itérative et version récursive).

   Solution : Version récursive :

```
1   function Est_Present_Recursive (Liste: in T_Liste; Element: in T_Element) return Boolean i
2   begin
3       if Liste = Null then
4           return False;
5       elsif Liste.all.Element = Element then
6           return True;
7       else
8           return Est_Present_Recursive (Liste.all.Suivante, Element);
9       end if;
10  end Est_Present_Recursive;
```

   Version récursive (avec des expressions booléennes plutôt que des conditionnelles) :

```
1   function Est_Present_Recursive_Bis (Liste: in T_Liste; Element: in T_Element) return Boole
2   begin
3       return Liste /= Null and then (Liste.all.Element = Element
4           or else Est_Present_Recursive_Bis (Liste.all.Suivante, Element));
5   end Est_Present_Recursive_Bis;
```

   Version Itérative :

```
1   function Est_Present_Iterative (Liste: in T_Liste; Element: in T_Element) return Boolean i
2       Curseur: T_Liste;
3   begin
4       Curseur := Liste;
5       while Curseur /= Null and then Curseur.all.Element /= Element loop
6           Curseur := Curseur.all.Suivante;
7       end loop;
8       return Curseur /= null;
9   end Est_Present_Iterative;
```

   Version en utilisant `Cellule_Contenant` (sous-programme défini un peu plus loin) :

```
1       function Est_Present_Avec_Cellule (Liste: in T_Liste; Element: in T_Element) return Bo
2       begin
3           return Cellule_Contenant (Element, Liste) /= null;
4               --! S'il n'y a pas d'exception, on a forcément cette condition vraie
5       exception
6           when Element_Absent_Error =>
7               return False;
8       end Est_Present_Avec_Cellule;
```

7. Supprimer un entier e dans une liste (version itérative et version récursive).
   Solution : Version récursive :

```
procedure Supprimer_Recursive (Liste: in out T_Liste; Element: in T_Element) is
    A_Detruire: T_Liste;
begin
    if Liste = null then
        raise Element_Absent_Error;
    elsif Liste.all.Element = Element then
        A_Detruire := Liste;
        Liste := Liste.all.Suivante;
        Free (A_Detruire);
    else
        Supprimer_Recursive (Liste.all.Suivante, Element);
    end if;
end Supprimer_Recursive;
```

   Version Itérative :

```
procedure Supprimer_Recursive (Liste: in out T_Liste; Element: in T_Element) is
    A_Detruire: T_Liste;
begin
    if Liste = null then
        raise Element_Absent_Error;
    elsif Liste.all.Element = Element then
        A_Detruire := Liste;
        Liste := Liste.all.Suivante;
        Free (A_Detruire);
    else
        Supprimer_Recursive (Liste.all.Suivante, Element);
    end if;
end Supprimer_Recursive;
```

8. Obtenir l'adresse de (le pointeur sur) la première cellule d'une liste qui contient l'entier e.
   Solution : Version récursive :

```
function Cellule_Contenant_Recursive (Element: T_Element; Liste: in T_Liste) return T_List
begin
    if Liste = null then
        raise Element_Absent_Error;
    elsif Liste.all.Element = Element then
        return Liste;
    else
        return Cellule_Contenant_Recursive (Element, Liste.all.Suivante);
    end if;
end Cellule_Contenant_Recursive;
```

   Version Itérative :

```
function Cellule_Contenant_Iterative (Element: T_Element; Liste: in T_Liste) return T_List
    Curseur: T_Liste;
begin
```

```
4       Curseur := Liste;
5       while Curseur /= null and then Curseur.all.Element /= Element loop
6           Curseur := Curseur.all.Suivante;
7       end loop;
8       if Curseur = null then
9           raise Element_Absent_Error;
10      else
11          return Curseur;
12      end if;
13  end Cellule_Contenant_Iterative;
```

9. Insérer un entier e après la première occurrence d'un entier f dans une liste.
   Solution :

```
1   procedure Inserer_Apres (Liste: in out T_Liste ; Nouveau, Element: in T_Element) is
2       Reference: T_Liste;
3   begin
4       Reference := Cellule_Contenant (Element, Liste);
5           --! peut lever Element_Absent_Error que l'on laisse se propager.
6       Reference.all.Suivante := new T_Cellule'(Nouveau, Reference.all.Suivante);
7   end Inserer_Apres;
```

10. Obtenir l'entier en position i dans une liste. Le premier entier est à la position 0.
    Solution : Version récursive :

```
1   function Ieme_Recursive (Liste: in T_Liste; Indice: in Integer) return T_Element is
2   begin
3       if Liste = null or Indice < 0 then
4           raise Indice_Error;
5       elsif Indice = 0 then
6           return Liste.all.Element;
7       else
8           return Ieme_Recursive (Liste.all.Suivante, Indice - 1);
9       end if;
10  end Ieme_Recursive;
```

    Version Itérative :

```
1   function Ieme_Iterative (Liste: in T_Liste; Indice: in Integer) return T_Element is
2       Curseur: T_Liste;
3       Index: Integer;
4   begin
5       Curseur := Liste;
6       Index := Indice;
7       while Curseur /= null and Index > 0 loop
8           Curseur := Curseur.all.Suivante;
9           Index := Index - 1;
10      end loop;
11      if Curseur = null or Index < 0 then
12          raise Indice_Error;
13      else
14          return Curseur.all.Element;
```

```
15      end if;
16  end Ieme_Iterative;
```

11. Supprimer l'entier à la position i dans une liste.

Solution : Version récursive :

```
1   procedure Supprimer_Ieme_Recursive (Liste: in out T_Liste; Indice: in Integer) is
2       A_Detruire: T_Liste;
3   begin
4       if Liste = null or Indice < 0 then
5           raise Indice_Error;
6       elsif Indice = 0 then
7           A_Detruire := Liste;
8           Liste := Liste.all.Suivante;
9           Free (A_Detruire);
10      else
11          Supprimer_Ieme_Recursive (Liste.all.Suivante, Indice - 1);
12      end if;
13  end Supprimer_Ieme_Recursive;
```

Version Itérative :

```
1       procedure Supprimer_Ieme_Iterative (Liste: in out T_Liste; Indice: in Integer) is
2           A_Detruire: T_Liste;
3           Curseur: T_Liste;
4       begin
5           if Indice <= 0 then        -- suppression au début ?
6               if Indice < 0 or Liste = null then
7                   raise Indice_Error;
8               else
9                   A_Detruire := Liste;
10                  Liste := Liste.all.Suivante;
11                  Free (A_Detruire);
12              end if;
13          else
14              -- Trouver le (I-1)eme élément
15              Curseur := Liste;
16              for I in 1..Indice - 1 loop
17                  if Curseur = null then
18                      raise Indice_Error;
19                  else
20                      Curseur := Curseur.all.Suivante;
21                  end if;
22              end loop;
23
24              -- Supprimer l"élément
25              if Curseur = null or else Curseur.all.Suivante = null then
26                  raise Indice_Error;
27              else
28                  A_Detruire := Curseur.all.Suivante;
29                  Curseur.all.Suivante := A_Detruire.all.Suivante;
```

```
30                  Free (A_Detruire);
31              end if;
32
33          end if;
34      end Supprimer_Ieme_Iterative;
```

Version Itérative (avec un While) :

```
1       procedure Supprimer_Ieme_Iterative_While (Liste: in out T_Liste; Indice: in Integer) i
2           A_Detruire: T_Liste;
3           Curseur: T_Liste;
4           Index: Integer;
5       begin
6           if Indice <= 0 then         -- suppression au début ?
7               if Indice < 0 or Liste = null then
8                   raise Indice_Error;
9               else
10                  A_Detruire := Liste;
11                  Liste := Liste.all.Suivante;
12                  Free (A_Detruire);
13              end if;
14          else
15              -- Trouver le (I-1)eme élément
16              Index := 0;
17              Curseur := Liste;
18              while Curseur /= null and then Index < Indice - 1 loop
19                  Curseur := Curseur.all.Suivante;
20                  Index := Index + 1;
21              end loop;
22
23              -- Supprimer l"élément
24              if Curseur = null or else Curseur.all.Suivante = null then
25                  raise Indice_Error;
26              else
27                  A_Detruire := Curseur.all.Suivante;
28                  Curseur.all.Suivante := A_Detruire.all.Suivante;
29                  Free (A_Detruire);
30              end if;
31
32          end if;
33      end Supprimer_Ieme_Iterative_While;
```

Exercice 3 : Module Liste

Les sous-programmes précédents peuvent être regroupés au sein d'un module générique.

1. Définir l'interface d'un module générique `P_Liste` permettant de gérer des listes chaînées linéaires simples avec encapsulation.

2. Donner la structure générale du corps de ce module.