PARTIE 3
TDA :
TYPE DE DONNES ABSTRAIT

ENSEIGNANTS:
MME. FEYROUZ HAMDAOUI, M. SOFIENE BEN AHMED ET M. MOEZ HAMMAMI

2025/2026 –SU1

Polytechnique Sousse EUR-ACE®

1

TYPES DE DONNEES ABSTRAITS (TDA)

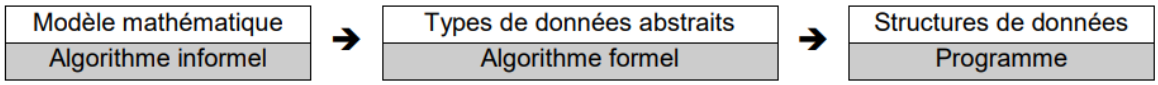


Figure 2.1 : **Processus de résolution d'un problème**

2

TYPES DE DONNEES ABSTRAITS (TDA)

- Un type de données abstraits est constitué par:
  - un modèle mathématique
  - et les différentes opérations définies sur ce modèle.
- Pour représenter le modèle mathématique sous-jacent à un TDA, il faut recourir à des structures de données, qui sont des ensembles de variables, a priori de types différents, reliées de multiples façons.

3

LISTES

ENSEIGNANTS:
MME. FEYROUZ HAMDAOUI, M. SOFIENE BEN AHMED ET M. MOEZ HAMMAMI

2025/2026 –SU1

Poly technique SOUSSE EUR-ACE®

4

LISTES: DÉFINITION (1/5)

- D'un point de vue mathématique, une liste est une suite, vide ou non, d'éléments d'un type donné (que nous désignerons par TypeElem).
- Il est habituel de représenter une telle suite par : a1, a2, …, an où n est positif ou nul, et où chaque ai est de type TypeElem. Le nombre n d'éléments s'appelle la longueur (ou la taille) de la liste.
  - Si $n \geq 1$, on dit que a1 est le premier élément et an le dernier.
  - Si $n = 0$, on parle de liste vide, sans éléments.

5

LISTES: DÉFINITION (2/5)

- Pour construire un TDA à partir de la notion mathématique de type liste, il nous faut définir un ensemble d'opérations sur des objets de type Liste.
- Examinons à présent un sous-ensemble représentatif d'opérations (primitives) sur les listes.
- Dans ce qui suit, L est une liste d'objets de type TypeElem, E est un objet de ce type. Ceci est la convention que nous adopterons dans tout ce chapitre.

6

LISTES: DÉFINITION (3/5)

1. Procédure **Init** (Var L : Liste)
   -- Précond :
   -- Postcond : transforme la liste L en liste vide

2. Fonction **Taille** (L : Liste) : Entier
   -- Précond :
   -- Postcond : Taille(L) = donne le nombre d'éléments de la liste L

3. Fonction **Vide** (L : Liste) : Booléen
   -- Précond :
   -- Postcond : vérifie si la liste L est vide $\Rightarrow$ Vide(L) = (Taille(L) = 0)

4. Fonction **Premier** (L : Liste) : TypeElem
   -- Précond : Non Vide(L)
   -- Postcond : Premier(L) = premier élément de la liste L

5. Fonction **Dernier** (L : Liste) : TypeElem
   -- Précond : Non Vide(L)
   -- Postcond : Dernier(L) = dernier élément de la liste L

7

LISTES: DÉFINITION (4/5)

6. Fonction **Reste** (L : Liste) : Liste
   -- Précond : Non Vide(L)
   -- Postcond : renvoie toute la liste L sans le premier élément
   -- Si L contient $a_1, a_2, \ldots a_n$, Reste(L) = $a_2, a_3, \ldots a_n$

7. Fonction **Face** (L : Liste) : Liste
   -- Précond : Non Vide(L)
   -- Postcond : renvoie toute la liste L sans le dernier élément
   -- Si L contient $(a_1, a_2, \ldots a_n)$, Face(L) = $(a_1, a_2, \ldots a_{n-1})$

8. Procédure **InsérerDébut** (E : TypeElem, Var L : Liste)
   -- Précond : le réservoir n'est pas plein
   -- Postcond : L' ayant tous les éléments de la liste L et en tête un élément contenant E
   -- Si L contient $(a_1, a_2, \ldots a_n)$, elle devient $(E, a_1, a_2, \ldots a_n)$
   -- Premier(L') = E
   -- Reste(L') = L et Taille(L') = Taille(L) + 1

8

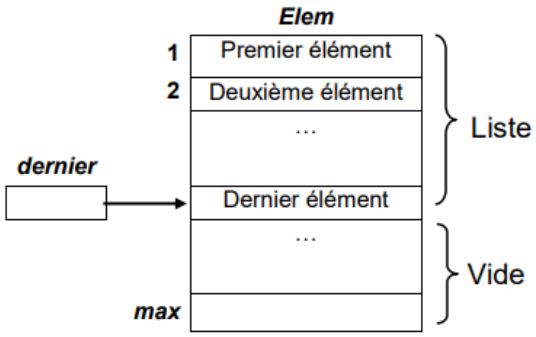
LISTES: DÉFINITION (5/5)

9. Procédure **InsérerFin** (Var L : Liste, E : TypeElem)
   -- Précond : le réservoir n'est pas plein
   -- Postcond : L' ayant tous les éléments de la liste L et en queue un élément contenant E
   -- Si L contient $(a_1, a_2, \ldots a_n)$, elle devient $(a_1, a_2, \ldots a_n, E)$
   -- Dernier(L') = E
   -- Face(L') = L et Taille(L') = Taille(L) + 1

10. Procédure **SupprimerTête** (Var L : Liste)
   -- Précond : Non Vide(L)
   -- Postcond : L' ayant tous les éléments de la liste L sans le premier élément
   -- Si L contient $(a_1, a_2, \ldots a_n)$, elle devient $(a_2, \ldots a_n)$
   -- L' = Reste(L) et Taille(L') = Taille(L) - 1

9

LISTES: MISE EN ŒUVRE
I. LISTES CONTIGÜES OU INDEXEES (1/6)



10

LISTES: MISE EN ŒUVRE
I. LISTES CONTIGÜES OU INDEXEES (2/6)

```
DefConst
    max (Entier) = 100
DefType
    Liste = Structure
        elem : Tableau [1..max] de TypeElem
        dernier : 0..max
    Fin Structure
```

-- Le type Liste consiste en un enregistrement
-- à deux champs. Le premier *elem* est un
-- tableau dont la taille est suffisante pour
-- contenir les listes les plus grandes que l'on
-- désire traiter. Le second champ *dernier* est
-- un entier (0..max) indiquant la position du
-- dernier élément du tableau.

Implémenter les opérations fondamentales sur une liste indexée

11

LISTES: MISE EN ŒUVRE
I. LISTES CONTIGÜES OU INDEXEES (3/6)

1. **Procédure *Init*** (VAR L : Liste)
   -- Précond :
   -- Postcond : transforme la liste L en liste vide
   Début
   L.dernier ← 0
   Fin

2. **Fonction *Taille*** (L : Liste) : Entier
   -- Précond :
   -- Postcond : Taille(L) = donne le nombre d'éléments de la liste L
   Début
   Taille ← L.dernier
   Fin

3. **Fonction *Vide*** (L : Liste) : Booléen
   -- Précond :
   -- Postcond : vérifie si la liste L est vide ⇒ Vide(L) = (Taille(L) = 0)
   Début
   Vide ← (L.dernier = 0) -- ou Vide ← (Taille(L) = 0)
   Fin

4. Fonction ***Premier*** (L : Liste) : TypeElem
   -- Précond : Non Vide(L)
   -- Postcond : Premier(L) = premier élément de la liste L
   Début
   Premier ← L.elem[1]
   Fin

LISTES: MISE EN ŒUVRE I. LISTES CONTIGÜES OU INDEXEES (4/6)

5. Fonction ***Dernier*** (L : Liste) : TypeElem

```
-- Précond    : Non Vide(L)
-- Postcond   : Dernier(L) = dernier élément de la liste L
Début
    Dernier ← L.elem[L.dernier]
Fin
```

6. Fonction ***Reste*** (L : Liste) : Liste

```
-- Précond    : Non Vide(L)
-- Postcond   : renvoie toute la liste L sans le premier élément
--              Si L contient (a1, a2, … an), Reste(L) = (a2, a3,… an)
DefVar
    p (Liste)
    i (Entier)
Début
    p ← L
    Pour i de 1 à Taille(L) – 1 Faire
        p.elem[i] ← p.elem[i+1]
    Fin Pour
    p.dernier ← Taille(L) – 1      -- ou (p.dernier – 1)
    Reste ← p
Fin
```

LISTES: MISE EN ŒUVRE I. LISTES CONTIGÜES OU INDEXEES (5/6)

7. Procédure ***InsérerDébut*** (E : TypeElem, Var L : Liste)

```
-- Précond    : le réservoir n'est pas plein
-- Postcond   : L' ayant tous les éléments de la liste L et en tête un élément contenant E
--              Si L contient (a1, a2, … an), elle devient (E, a1, a2, … an)
--              Premier(L') = E
--              Reste(L') = L et Taille(L') = Taille(L) + 1
DefVar
    i (Entier)
Début
    Pour i de Taille(L) +1 à 2 Pas –1 Faire   -- ou i de L.dernier +1 à 2
        -- décaler tous les éléments d'un rang vers la fin de L
        L.elem[i] ← L.elem[i-1]
    Fin Pour
    L.dernier ← Taille(L) + 1     -- ou L.dernier + 1
    L.elem[1] ← E
Fin
```

8. Procédure ***InsérerFin*** (Var L : Liste, E : TypeElem)

```
-- Précond    : le réservoir n'est pas plein
-- Postcond   : L' ayant tous les éléments de la liste L et en queue un élément contenant E
--              Si L contient a1, a2, … an, elle devient a1, a2, … an, E
--              Dernier(L') = E
--              Reste(L') = L et Taille(L') = Taille(L) + 1
Début
    L.dernier ← Taille(L) + 1     -- ou L.dernier + 1
    L.elem[L.dernier] ← E
Fin
```

14

LISTES: MISE EN ŒUVRE
I. LISTES CONTIGÜES OU INDEXEES (6/6)

9. Procédure ***SupprimerTête*** (Var L : Liste)

```
-- Précond    : Non Vide(L)
-- Postcond   : L' ayant tous les éléments de la liste L sans le premier élément
--              Si L contient (a1, a2, … an), elle devient (a2, … an)
--              L' = Reste(L)  et  Taille(L') = Taille(L) - 1

Début
    L ← Reste(L)
Fin
```

10. **Fonction *Pleine*** (L : Liste) : Booléen

```
-- Précond    :
-- Postcond   : vérifie si la liste L est pleine ⇒ Pleine(L) = (Taille(L) = max)
Début
    Pleine ← (L.dernier = max)          -- ou  (Taille(L) = max)
Fin
```

15

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.1. POINTEUR (1/4)

Un pointeur: est une cellule dont la valeur réfère à une autre cellule.

- Dans les représentations graphiques des structures de données, on a l'habitude d'indiquer que la cellule « A » est un pointeur sur la cellule « B » en dessinant une flèche de « A » vers « B ».
- En algorithmique, on peut créer et manipuler une variable pointeur « p » qui pointe sur des cellules d'un type donnée, disons « TypeCellule », par la déclaration

16

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.1. POINTEUR (2/4)

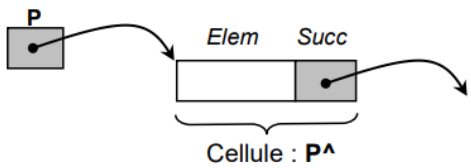
Figure 2.2 : **Déclaration et manipulation algorithmique d'un pointeur de « TypeCellule »**

```
DefType
    TypeCellule = …          -- TypeCellule est appelé type de base (Entier, Réel, Structure,…)
DefVar
    s (TypeCellule)          -- s est une variable statique de type TypeCellule
    p (^TypeCellule)         -- p est un pointeur vers un objet de type TypeCellule
                             -- ^ est le symbole du pointeur
Début
    …                        -- ici, p n'est pas significatif
    Allouer (p)              -- p est un pointeur dont la valeur est l'adresse de p^
                             -- p^ est une variable dynamique (de type TypeCellule)
                             -- créée et pointée par p
    …
    Libérer (p)              -- p^ est détruite et P n'est plus significatif
    ...
Fin
```

17

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.1. POINTEUR (3/4)

Figure 2.3 : **Représentation graphique d'un pointeur P vers une cellule du type TypeCellule**



Supposons que nous disposions d'un réservoir de cellules. Deux procédures permettent la gestion de ce réservoir, ce sont **Allouer** et **Libérer**.

18

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.1. POINTEUR (4/4)

Figure 2.4 : **Spécification des procédures Allouer & Libérer**

Procédure ***Allouer*** *(VAR p : ^TypeCellule)*
-- Précond :
-- Postcond : p contient l'adresse d'une nouvelle cellule allouée.

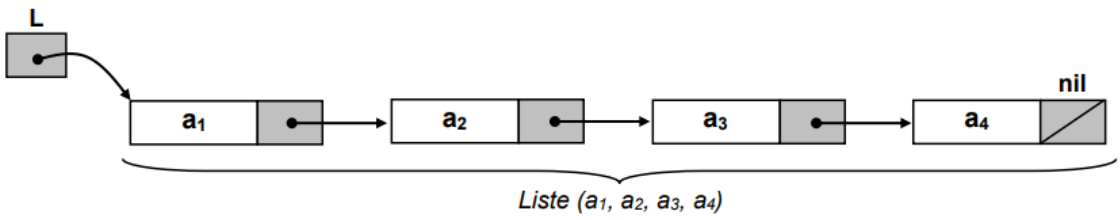
Procédure ***Libérer*** *(p : ^TypeCellule)*
-- Précond :
-- Postcond : rend la cellule d'adresse p au réservoir.

Remarque :

- La procédure « Allouer » permet à chaque appel d'obtenir une nouvelle cellule dont l'adresse sera retournée dans la variable p.
- Ces deux procédures permettent d'obtenir ou de rendre une cellule à la mémoire au fur et à mesure des besoins de l'algorithme.

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (1/6)

Figure 3.2 : Représentation graphique de la liste linéaire chaînée (a_1, a_2, a_3, a_4)



20

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (2/6)

Figure 3.3 : Déclaration algorithmique d'une liste linéaire chaînée

```
DefType
   TypeElem = …                    -- TypeElem est un type simple, structure, vecteur…
   Cellule =  Structure
        elem : TypeElem            -- Information utile
        succ : ^Cellule            -- Référence du successeur
   Fin Structure
   Liste = ^Cellule
DefVar
   L (Liste)                       -- ⇔ L (^Cellule)  : une variable d'accès à la liste
```

Implémenter les opérations fondamentales sur une liste simplement chainée

21

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (3/6)

Primitives fondamentales

pour sur une liste simplement chaînée :

1. **Procédure *Init*** (Var L : Liste)
   -- Précond :
   -- Postcond : transforme la liste L en liste vide

```
Début
   L ← nil
Fin
```

2. **Fonction *Taille*** (L : Liste) : Entier
   -- Précond :
   -- Postcond : Taille(L) = donne le nombre d'éléments de la liste L

```
Début
   Si Vide(L)
      Alors   Taille ← 0
      Sinon   Taille ← 1+ Taille (Reste (L))
   Fin Si
Fin
```

3. **Fonction *Vide*** (L : Liste) : Booléen
   -- Précond :
   -- Postcond : vérifie si la liste L est vide ⇒ Vide(L) = (Taille(L) = 0) ou (L = nil)

```
Début
   Vide ← (L = nil)
Fin
```

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (4/6)

4. Fonction ***Premier*** (L : Liste) : TypeElem
```
-- Précond    : Non Vide(L)
-- Postcond   : Premier(L) = premier élément de la liste L
Début
    Premier ← L^.elem
Fin
```

5. Fonction ***Dernier*** (L : Liste) : TypeElem
```
-- Précond    : Non Vide(L)
-- Postcond   : Dernier(L) = dernier élément de la liste L
Début
    Si (Taille(L) = 1)
        Alors   Dernier ← L^.elem
        Sinon   Dernier ← Dernier (Reste (L))
    Fin Si
Fin
```

6. Fonction ***Reste*** (L : Liste) : Liste
```
-- Précond    : Non Vide(L)
-- Postcond   : renvoie toute la liste L sans le premier élément
--              Si L contient (a1, a2, … an), Reste(L) = (a2, a3,… an)
Début
    Reste ← L^.succ
Fin
```

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (5/6)

7. Procédure ***InsérerDébut*** (E : TypeElem, Var L : Liste)
```
-- Précond    : le réservoir n'est pas plein
-- Postcond   : L' ayant tous les éléments de la liste L et en tête un élément contenant E
--              Si L contient (a1, a2, … an), elle devient (E, a1, a2, … an)
--              Premier(L') = E
--              Reste(L') = L  et  Taille(L') = Taille(L) + 1
DefVar
    c (^Cellule)          -- ou  c (Liste)
Début
    Allouer (c)
    c^.elem ← E
    c^.succ ← L
    L ← c
Fin
```

24

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.2. LISTES SIMPLEMENT CHAINÉES (6/6)

8. Procédure ***InsérerFin*** (Var L : Liste, E : TypeElem)

```
-- Précond    :
-- Postcond   : L' ayant tous les éléments de la liste L et en queue un élément contenant E
--              Si L contient (a1, a2, … an), elle devient (a1, a2, … an, E)
--              Dernier(L') = E
--              Reste(L') = L  et  Taille(L') = Taille(L) + 1
Début
   Si Vide(L)
      Alors InsérerDébut (E, L)
      Sinon InsérerFin (Reste(L), E)
   Fin Si
Fin
```

9. Procédure ***SupprimerTête*** (Var L : Liste)

```
-- Précond    : Non Vide(L)
-- Postcond   : L' ayant tous les éléments de la liste L sans le premier élément
--              Si L contient (a1, a2, … an), elle devient (a2, … an)
--              L' = Reste(L)  et  Taille(L') = Taille(L) - 1
DefVar
   p (Liste)
Début
   p ← L
   L ← Reste(L)
   Libérer (p)
Fin
```

25

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.3. LISTES DOUBLEMENT CHAINÉES (1/4)

Figure 3.4 : Représentation graphique de la liste doublement chaînée (a_1, a_2, a_3, a_4)

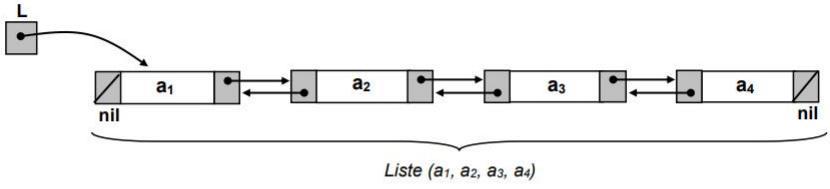


Figure 3.5 : Déclaration algorithmique d'une liste doublement chaînée

```
DefType
   TypeElem = …                    -- TypeElem est un type simple, structure, vecteur…
   Cellule  = Structure
      préd : ^Cellule              -- Référence du prédécesseur
      elem : T                     -- Information utile
      succ : ^Cellule              -- Référence du successeur
   Fin Structure
   Liste = ^Cellule
```

26

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.3. LISTES DOUBLEMENT CHAINÉES (2/4)

1. **Procédure *InsérerDébut*** (E : TypeElem, Var L : Liste)

```
-- Précond    : L possède un chaînage bidirectionnel
-- Postcond   : L' ayant tous les éléments de la liste L et en tête un élément contenant E
--              Si L contient (a1, a2, … an), elle devient (E, a1, a2, … an)
--              Premier(L') = E
--              Reste(L') = L
DefVar
    c (^Cellule)
Début
    Allouer (c)
    c^.elem ← E
    c^.pred ← nil
    c^.succ ← L
    Si Non Vide (L) Alors
        L^.pred ← c
    Fin Si
    L ← c
Fin
```

27

LISTES: MISE EN ŒUVRE
II.3. LISTES DOUBLEMENT CHAINÉES (3/4)

2. **Procédure *Insérer*** (Var L : Liste, E : TypeElem, K : Entier, Var Ok : Booléen)

```
-- Précond    : L possède un chaînage bidirectionnel et K > 0
-- Postcond   : Si L contient (a1, a2, … an), elle devient (a1, a2, … ak-1, E, ak,…an) et Ok
--              Si K = Taille(L) + 1, la liste L devient (a1, a2, … an, E) et Ok
--              Si K = 1, la liste devient (E, a1, a2, … an) et Ok
--              Sinon l'opération est impossible et Non Ok
DefVar
    pKmoins1, c (^Cellule)
Début
    Ok ← Faux
    Si (K=1)
        Alors   Ok ← Vrai
                InsérerDébut (E, L)
        Sinon   pKmoins1 ← Pointeur (L, K-1)          -- retourne le pointeur qui pointe vers la
                Si Non Vide (pKmoins1) Alors           -- cellule se trouvant à la position k-1
                    Ok ← Vrai
                    Allouer (c)
                    c^.elem ← E
                    c^.pred ← pKmoins1
                    c^.succ ← pKmoins1^.succ          -- ou c^.succ ← Reste (pKmoins1)
                    Si Non Vide (pKmoins1^.succ) Alors -- ou Non Vide (Reste (pKmoins1))
                        PKmoins1^.Succ^.Pred ← C
                    Fin Si
                    pKmoins1^.succ ← c
                Fin Si
    Fin Si
Fin
```

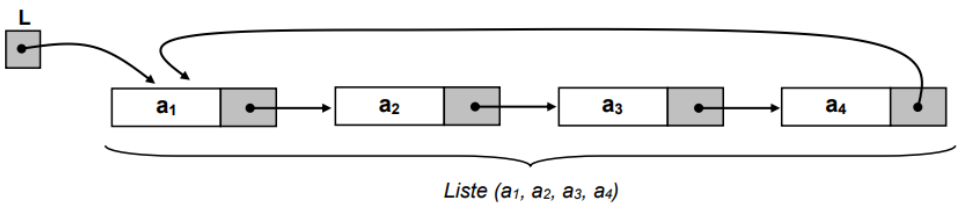
LISTES: MISE EN ŒUVRE
II.3. LISTES DOUBLEMENT CHAINÉES (4/4)

3. **Procédure *Supprimer*** (Var L : Liste, K : Entier, Var Ok : Booléen)

```
-- Précond   : L possède un chaînage bidirectionnel et K > 0
-- Postcond  : Si L contient a1, a2, … an, elle devient a1, a2, … ak-1, ak+1,…an
--             Si K = Taille(L), la liste L devient a1, a2, … an-1
--             Si K = 1, la liste devient a2, … an
--             Sinon l'opération est impossible.
DefVar
    pK (^Cellule)
Début
    Ok ← Faux
    pK ← Pointeur (L, K)
    Si Non Vide (pK) Alors
        Ok ← Vrai
        Si (pK = L)
            Alors   L ← L^.succ                          -- ou L ← Reste (L)
            Sinon   pK^.pred^.succ ← pK^.succ            -- ou pK^.pred^.succ ← Reste (pK)
        Fin Si
        Si Non Vide (pK^.succ) Alors                     -- ou Non Vide (Reste (pK))
            pK^.succ^.pred ← pK^.pred
        Fin Si
    Fin Si
Fin
```

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.4. LISTES CIRCULAIRES (1/2)

Figure 3.8 : Représentation graphique de la liste circulaire (a_1, a_2, a_3, a_4)



Liste (a_1, a_2, a_3, a_4)

30

LISTES: MISE EN ŒUVRE
II. LISTES CHAINEES
II.4. LISTES CIRCULAIRES (2/2)

```
-- Parcours d'une liste circulaire
Procédure Parcours (L : Liste)
   -- Précond  :  Non Vide (L)
   -- Postcond :  Traitement des éléments de la liste
   DefVar
      p (Liste)
      Stop (Booléen)
   Début
      p ← L
      Stop ← Faux
      TantQue (Non Stop)  Faire
         Traiter (p)              -- c'est une procédure quelconque de traitement d'une cellule d'adresse donnée.
         p ← Reste (p)
         Stop ← (p = L)
      Fin TantQue
   Fin
```

LISTES PARTICULIÈRES: PILES ET FILES

ENSEIGNANTS:
MME. FEYROUZ HAMDAOUI, M. SOFIENE BEN AHMED ET M. MOEZ HAMMAMI

2025/2026 –SU1

Poly technique SOUSSE
EUR-ACE®

32

PILES: DÉFINITION

- La structure de pile est une structure de liste particulière permettant d'exprimer le comportement d'une pile d'assiettes : on peut ajouter et enlever des assiettes au sommet de la pile ; toute insertion ou retrait d'une assiette en milieu de pile est une opération qui comporte de risque (la pile s'écroule).
- La stratégie de gestion d'une pile est dernier arrivé, premier servi. En anglais on dira « Last In First Out », plus connu sous le nom de « **LIFO** ». Le dernier élément d'une pile est appelé **Base** de la pile ; le premier élément est appelé **Sommet** de la pile. Les opérations de mise à jour (insertion et suppression), d'après la définition, se font à partir du sommet.

33

PRIMITIVES

1. **Procédure *Init*** (Var P : Pile)
   -- Précond :
   -- Postcond : transforme la pile P en pile vide

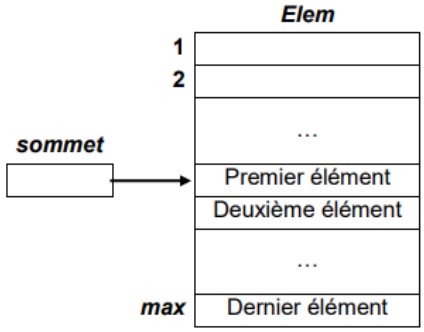
2. **Fonction *Vide*** (P : Pile) : Booléen
   -- Précond :
   -- Postcond : retourne vrai si la pile P est vide et faux sinon

3. **Fonction *Sommet*** (P : Pile) : Booléen
   -- Précond : Non Vide (P)
   -- Postcond : retourne (sans le dépiler) l'élément au sommet de la pile P

4. **Procédure *Empiler*** (Var P : Pile, E : TypeElem)
   -- Précond :
   -- Postcond : P' ayant tous les éléments de la pile P et en tête un élément contenant E
   -- Si P contient $(a_1, a_2, \ldots a_n)$, elle devient $(E, a_1, a_2, \ldots a_n)$
   -- Sommet (P') = E

5. **Procédure *Dépiler*** (Var P : Pile, E : TypeElem)
   -- Précond : Non Vide (P)
   -- Postcond : P' ayant tous les éléments de la pile P sans le sommet
   -- Si P contient $(a_1, a_2, \ldots a_n)$, elle devient $(a_2, \ldots a_n)$ et $E = a_1$

REPRESENTATION CONTIGÜE D'UNE PILE



35

REPRESENTATION CONTIGÜE D'UNE PILE

```
DefConst
    max (Entier) = 100
DefType
    Pile = Structure
        sommet : 0..max+1
        elem : Tableau [1..max] de TypeElem
    Fin Structure
```

-- Le type Pile consiste en un enregistrement
-- à deux champs. Le premier *elem* est un
-- tableau dont la taille est suffisante pour
-- contenir les piles les plus grandes que l'on
-- désire traiter. Le second champ *sommet* est
-- un entier (0..max+1) indiquant la position du
-- premier élément du tableau.

36

REPRESENTATION CONTIGÜE D'UNE PILE

1. **Procédure *Init*** (Var P : Pile)
   -- Précond :
   -- Postcond : transforme la pile P en pile vide
   Début
   P.sommet ← max +1
   Fin

2. **Fonction *Vide*** (P : Pile) : Booléen
   -- Précond :
   -- Postcond : retourne vrai si la pile P est vide et faux sinon
   Début
   Vide ← (P.sommet > max)
   Fin

3. **Fonction *Sommet*** (P : Pile) : Booléen
   -- Précond : Non Vide (P)
   -- Postcond : retourne (sans le dépiler) l'élément au sommet de la pile P
   Début
   Sommet ← P.elem[P.sommet]
   Fin

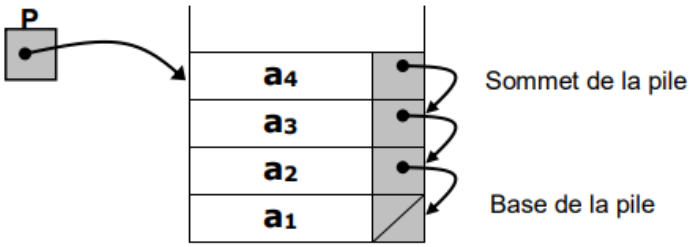
REPRESENTATION CONTIGÜE D'UNE PILE

4. **Procédure *Empiler*** (Var P : Pile, E : TypeElem)
   -- Précond : Non Pleine (P)
   -- Postcond : P' ayant tous les éléments de la pile P et en tête un élément contenant E
   -- Si P contient ($a_1, a_2, \ldots a_n$), elle devient ($E, a_1, a_2, \ldots a_n$)
   -- Sommet (P') = E
   Début
   P.sommet ← P.sommet – 1
   P.elem[P.sommet] ← E
   Fin

5. **Procédure *Dépiler*** (Var P : Pile, Var E : TypeElem)
   -- Précond : Non Vide (P)
   -- Postcond : P' ayant tous les éléments de la pile P sans le sommet
   -- Si P contient ($a_1, a_2, \ldots a_n$), elle devient ($a_2, \ldots a_n$) et E = a_1
   Début
   E ← P.elem[P.sommet]
   P.sommet ← P.sommet + 1
   Fin

6. **Fonction *Pleine*** (P : Pile) : Booléen
   -- Précond :
   -- Postcond : retourne vrai si la pile P est pleine et faux sinon
   Début
   Pleine ← (P.sommet = 1)
   Fin

REPRESENTATION CHAINEE D'UNE PILE



39

REPRESENTATION CHAINEE D'UNE PILE

```
DefType
    TypeElem = …                    -- TypeElem est un type simple, structure, vecteur…
    Cellule =  Structure
          elem : TypeElem           -- Information utile
          succ : ^Cellule           -- Référence du successeur
    Fin Structure
    Pile = ^Cellule

DefVar
    P (Pile)                        --  une variable d'accès à la Pile
```

40

REPRESENTATION CHAINEE D'UNE PILE

1. **Procédure *Init*** (Var P : Pile)

```
-- Précond    :
-- Postcond   : transforme la pile P en pile vide
Début
    Allouer (P)                    -- création d'une cellule de tête
    P ← nil
Fin
```

2. **Fonction *Vide*** (P : Pile) : Booléen

```
-- Précond    :
-- Postcond   : retourne vrai si la pile P est vide et faux sinon
Début
    Vide ← (P = nil)
Fin
```

3. **Fonction *Sommet*** (P : Pile) : Booléen

```
-- Précond    : Non Vide (P)
-- Postcond   : retourne (sans le dépiler) l'élément au sommet de la pile P
Début
    Sommet ← P^.elem
Fin
```

REPRESENTATION CHAINEE D'UNE PILE

4. **Procédure *Empiler*** (Var P : Pile, E : TypeElem)

```
-- Précond    : Non Pleine (P)
-- Postcond   : P' ayant tous les éléments de la pile P et en tête un élément contenant E
--              Si P contient (a1, a2, … an), elle devient (E, a1, a2, … an)
--              Sommet (P') = E
DefVar
    c (^Cellule)          -- ou  c (Pile)
Début
    Allouer (c)
    c^.elem ← E
    c^.succ ← nil
    P^.succ ← c
    P ← c
Fin
```

5. **Procédure *Dépiler*** (Var P : Pile, Var E : TypeElem)

```
-- Précond    : Non Vide (P)
-- Postcond   : P' ayant tous les éléments de la pile P sans le sommet
--              Si P contient (a1, a2, … an), elle devient (a2, … an) et E = a1
Début
    E ← P^.elem
    P ← P^.succ
Fin
```

FILES: DÉFINITION

Une file est un type particulier de liste, où les éléments sont insérés en queue et supprimés en tête. Le nom vient des files d'attente à un guichet, où le premier arrivé est le premier servi, ce que justifie le terme anglo-saxon de « **FIFO** » (First In First Out). Elles sont d'un usage très répandu dans la programmation système.

Les opérations sur une file sont analogues à celles définies sur les piles, la principale différence provenant du fait que les insertions se font en fin de liste plutôt qu'en début.

43

PRIMITIVES

1. **Procédure *Init*** (Var F : File)
   -- Précond :
   -- Postcond : transforme la pile F en file vide

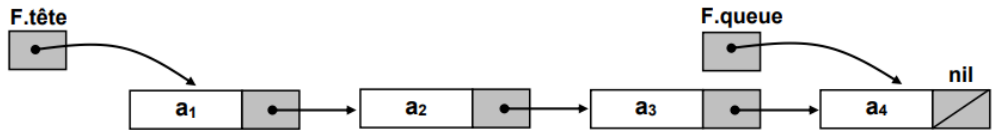
2. **Fonction *Vide*** (F : File) : Booléen
   -- Précond :
   -- Postcond : retourne vrai si la file F est vide et faux sinon

3. **Fonction *Tête*** (F : File) : Booléen
   -- Précond : Non Vide (F)
   -- Postcond : retourne (sans le défiler) l'élément au sommet de la file F

4. **Procédure *Enfiler*** (Var F : File, E : TypeElem)
   -- Précond :
   -- Postcond : F' ayant tous les éléments de la file F et en queue un élément contenant E
   -- Si F contient $(a_1, a_2, \ldots a_n)$, elle devient $(a_1, a_2, \ldots a_n, E)$

5. **Procédure *Défiler*** (Var F : File, Var E : TypeElem)
   -- Précond : Non Vide (F)
   -- Postcond : F' ayant tous les éléments de la file F sans le premier
   -- Si F contient $(a_1, a_2, \ldots a_n)$, elle devient $(a_2, \ldots a_n)$ et $E = a_1$

MISE EN ŒUVRE DES FILES PAR POINTEURS



45

MISE EN ŒUVRE DES FILES PAR POINTEURS

```
DefType
    TypeElem = …                    -- TypeElem est un type simple, structure, vecteur…
    Cellule =  Structure
        elem    : TypeElem          -- Information utile
        succ    : ^Cellule          -- Référence du successeur
    Fin Structure
    File = Structure
        tête    : ^Cellule          -- tête pointe sur l'élément de tête
        queue   : ^Cellule          -- queue pointe sur l'élément de queue
    Fin Structure
DefVar
    F (File)                        -- une variable d'accès à la file
```

46

MISE EN ŒUVRE DES FILES PAR POINTEURS

1. **Procédure *Init*** (Var F : File)

```
-- Précond    :
-- Postcond   : transforme la file F en file vide
Début
    Allouer (F.tête)          -- création d'une cellule de tête
    F.tête ← nil
    F.tête ← F.queue          -- l'en-tête est à la fois la dernière et la première cellule.
Fin
```

2. **Fonction *Vide*** (F : File) : Booléen

```
-- Précond    :
-- Postcond   : retourne vrai si la file F est vide et faux sinon
Début
    Vide ← (F.tête = F.queue)
Fin
```

3. **Fonction *Tête*** (F : File) : Booléen

```
-- Précond    : Non Vide (F)
-- Postcond   : retourne (sans le défiler) l'élément au sommet de la file F
Début
    Tête ←  F.tête^.Elem
Fin
```

47

MISE EN ŒUVRE DES FILES PAR POINTEURS

4. **Procédure *Enfiler*** (Var F : File, E : TypeElem)

```
-- Précond    :
-- Postcond   : F' ayant tous les éléments de la file F et en queue un élément contenant E
--              Si F contient (a1, a2, … an), elle devient (a1, a2, … an, E)
--              Premier(F') = a1
DefVar
    c (^Cellule)          -- ou  c (File)
Début
    Allouer (c)
    c^.elem ← E
    c^.succ ← nil
    F.queue^.succ ← c
    F.queue ← c
Fin
```

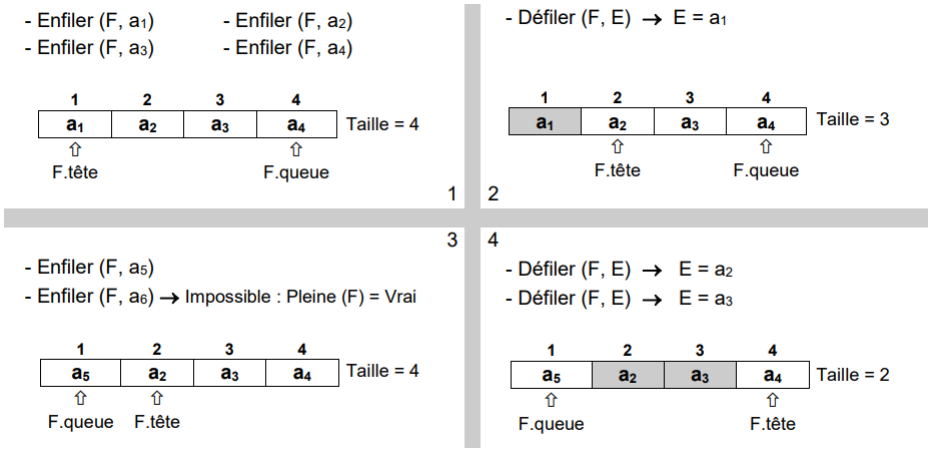
```
-- Version 2
Début
    Allouer (F.queue^.succ)
    F.queue ← F.queue^.succ
    F.queue^.elem ← E
    F.queue^.succ ← nil
Fin
```

5. **Procédure *Défiler*** (Var F : File, Var E : TypeElem)

```
-- Précond    : Non Vide (F)
-- Postcond   : F' ayant tous les éléments de la file F sans le premier élément
--              Si F contient (a1, a2, … an), elle devient (a2, … an) et E = a1
Début
    E ← F.tête^.elem
    F.tête ← F.tête^.succ
Fin
```

48

IMPLANTATION CIRCULAIRE DES FILES



49

IMPLANTATION CIRCULAIRE DES FILES

```
DefConst
    max (Entier) = 100
DefType
    File = Structure
        elem    : Tableau [1..max] de TypeElem
        tête    : 1..max
        queue   : 0..max
        taille  : 0..max
    Fin Structure
```

50

IMPLANTATION CIRCULAIRE DES FILES

1. **Procédure *Init*** (Var F : File)

```
-- Précond     :
-- Postcond    : transforme la file F en file vide
Début
    F.tête ← 1
    F.queue ← 0
    F.taille ← 0
Fin
```

2. **Fonction *Vide*** (F : File) : Booléen

```
-- Précond     :
-- Postcond    : retourne vrai si la file F est vide et faux sinon
Début
    Vide ← (F.taille = 0)
Fin
```

51

IMPLANTATION CIRCULAIRE DES FILES

3. **Fonction *Pleine*** (F : File) : Booléen

```
-- Précond     :
-- Postcond    : retourne vrai si la file F est pleine et faux sinon
Début
    Pleine ← (F.taille = max)
    -- ou bien
    -- Pleine ←  (F.tête = 1  Et  F.queue = max)  Ou  (F.queue < max  Et  F.tête = F.queue + 1)
Fin
```

4. **Fonction *Tête*** (F : File) : Booléen

```
-- Précond     : Non Vide (F)
-- Postcond    : retourne (sans le défiler) l'élément au sommet de la file F
Début
    Tête ←  F.tête^.Elem
Fin
```

52

IMPLANTATION CIRCULAIRE DES FILES

5. **Procédure *Enfiler*** (Var F : File, E : TypeElem)

```
-- Précond   :
-- Postcond  : F' ayant tous les éléments de la file F et en queue un élément contenant E
--             Si F contient (a1, a2, … an), elle devient (a1, a2, … an, E)
--             Premier(F') = a1
Début
    F.taille ← F.taille + 1
    Si (F.queue < max) Alors F.queue ← F.queue + 1
                       Sinon F.queue ← 1
    Fin Si
    F.elem[F.queue] ← E
Fin
```

6. **Procédure *Défiler*** (Var F : File, Var E : TypeElem)

```
-- Précond   : Non Vide (F)
-- Postcond  : F' ayant tous les éléments de la file F sans le premier élément
--             Si F contient (a1, a2, … an), elle devient (a2, … an) et E = a1
Début
    F.taille ← F.taille – 1
    E ← F.elem[F.tête]
    Si (F.tête = F.queue) Alors     F.tête ← 1
                                    F.queue ← 0
                          SinonSi   (F.tête < max)  Alors   F.tête ← F.tête + 1
                                                    Sinon   F.tête ← 1
    Fin Si
Fin
```