

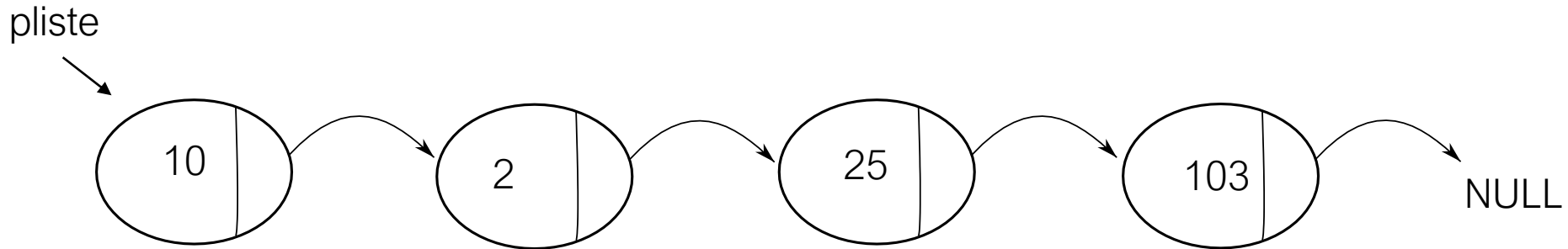
# INFORMATIQUE 3

## II. LES FILES ET LES PILES



# Rappel et objectifs

- Nous avons vu au cours dernier les listes chaînées.



- Les listes chaînées permettent d'ajouter/ de supprimer des éléments n'importe où dans la liste.
- Les **pires** et les **files** sont des objets informatiques qui permettent d'ajouter/ supprimer des éléments dans une liste en répondant à des règles plus spécifiques que les listes chaînées.

# I. Piles

# Piles : principe

- Une **pile** est une liste d'éléments qui sont « **empilés** » :
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- On l'appelle aussi **LIFO** : « last in, first out »
- Pensez à une pile d'assiettes !



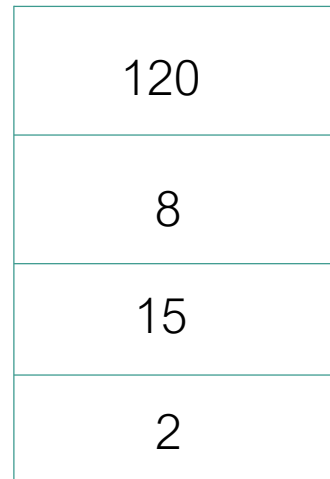
# Piles : principe

- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Exemple d'une pile d'entiers :

120
8
15
2

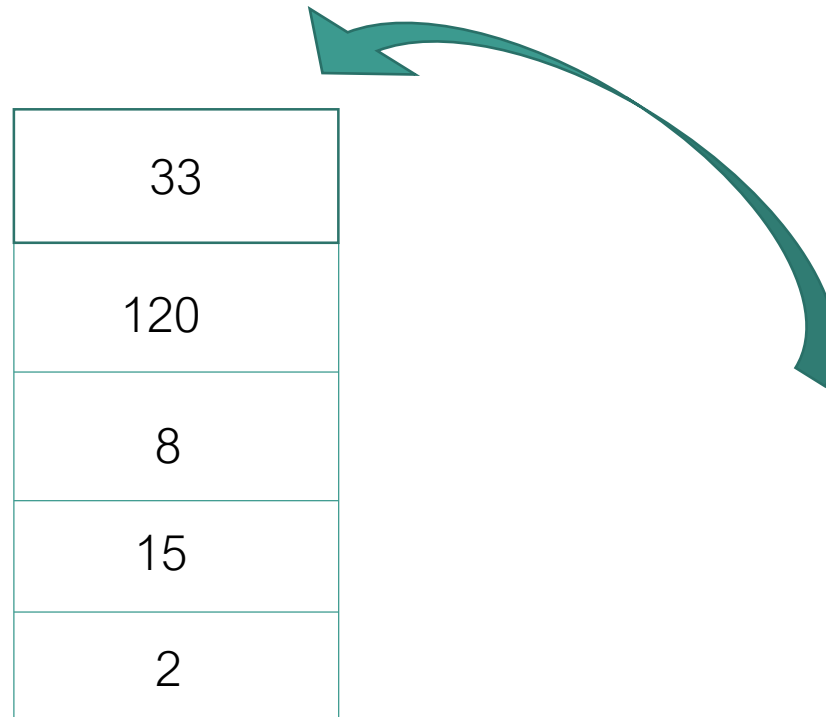
# Piles : principe

- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Ajout d'un nouvel élément : **empilage**



# Piles : principe

- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Empilage : le nouvel élément va au sommet de la pile



# Piles : principe

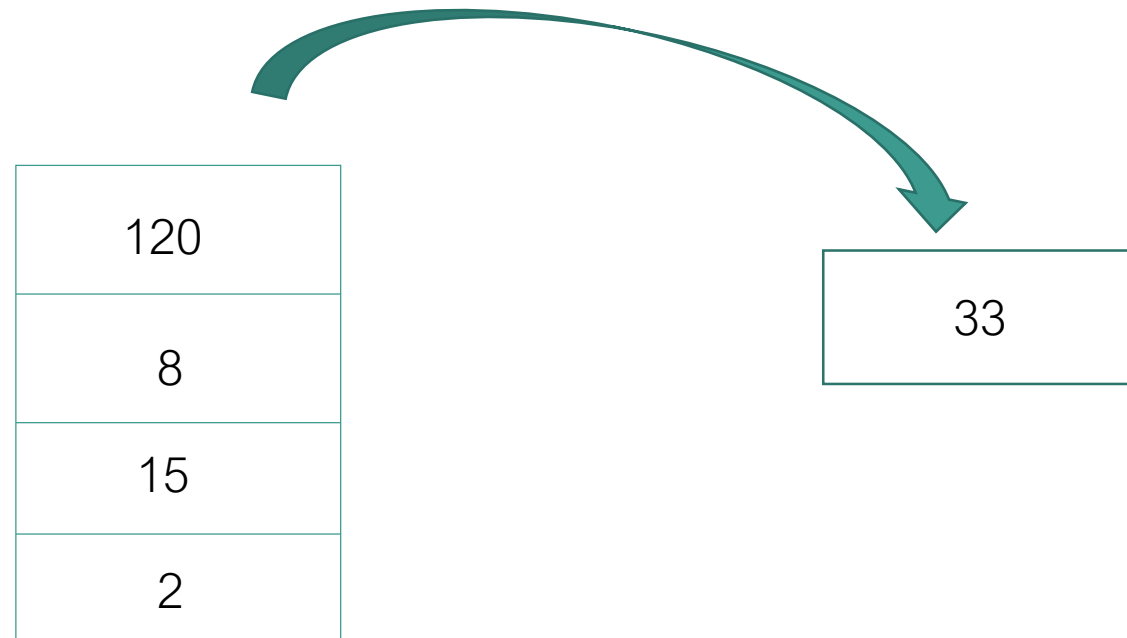
- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Suppression de deux éléments : **dépilage**

33
120
8
15
2



# Piles : principe

- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Dépilage de deux éléments :



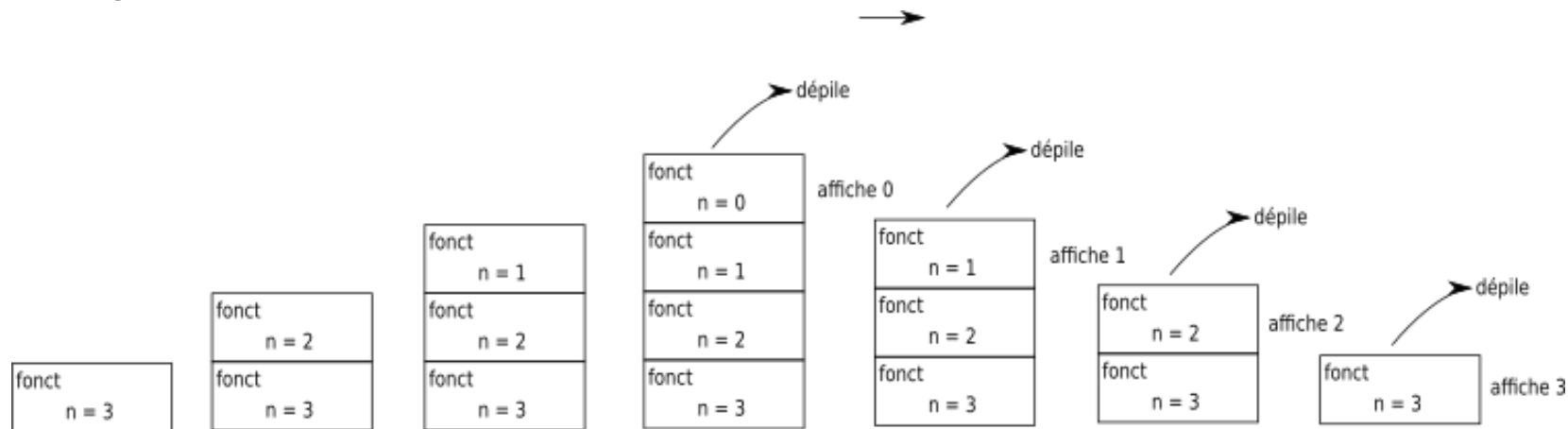
# Piles : principe

- Une pile est une liste d'éléments qui sont « empilés »:
  - Un élément est toujours ajouté en haut de la pile
  - Un élément est toujours retiré en haut de la pile
- Dépilage de deux éléments :



# Piles : utilisation

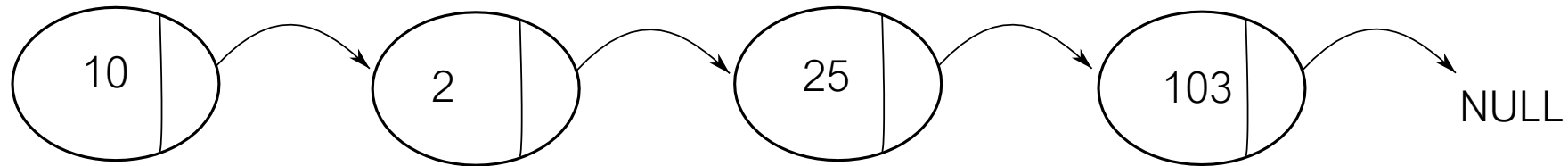
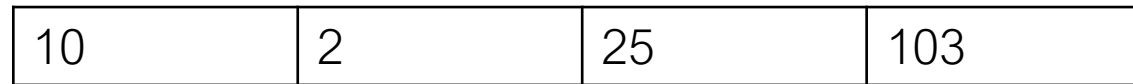
- Les piles sont très utilisées en informatique. On en trouve notamment:
  - Pour permettre des retours en arrière à l'utilisateur :
    - L'action « undo » (ctrl+z)
    - Retour arrière dans un navigateur web
  - Pour la gestion des appels de fonctions récursives non-terminale



- Dans les processeurs pour gérer, entre autre, l'appel à des fonctions quelconques.
- Et également pour simuler des problèmes mathématiques ou de la vie réelle!

# Piles : Implémentation d'une pile

- Une pile peut être implémentée de deux manières:
  - A l'aide d'un tableau (**pile statique**)
  - A l'aide d'une liste chaînée (**pile dynamique**)



- Pourquoi utiliser l'un ou l'autre ?

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau **tabPile** permettant de stocker les éléments de la pile.
  - La taille maximale **N** du tableau
  - Un entier **tete** indiquant l'indice du haut de la pile (remplissage).

```
Structure PileStat :  
    tabPile : tableau d'Element de taille 'taille' (constante)  
    (taille : entier          // taille du tableau)  
    tete :entier // indice indiquant le haut de la pile
```

OU

```
Structure PileStat :  
    tabPile : pointeur sur Element // si allocation dynamique  
    taille : entier          // taille du tableau  
    tete :entier // indice indiquant le haut de la pile
```

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau `tabPile` permettant de stocker les éléments de la pile.
  - La taille maximale `N` du tableau
  - Un entier `tete` indiquant l'indice du haut de la pile (remplissage).
- Exemple

10	2	25	103	/	/	/
----	---	----	-----	---	---	---

Ici `tabPile` est de `taille`  
`tete =`

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau **tabPile** permettant de stocker les éléments de la pile.
  - La taille maximale **N** du tableau
  - Un entier **tete** indiquant l'indice du haut de la pile (remplissage).
- Exemple

10	2	25	103	?	?	?
----	---	----	-----	---	---	---

↑  
tete

Ici **tabPile** est de **taille** 7  
**tete** = 3

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau `tabPile` permettant de stocker les éléments de la pile.
  - La taille maximale `N` du tableau
  - Un entier `tete` indiquant l'indice du haut de la pile (remplissage).
- Exemple

10	2	25	103	55	16	147
----	---	----	-----	----	----	-----

↑  
`tete`

Ici `tabPile` est de `taille 7`  
`tete = 3`

Remarque : les cases situées après `tête` ne sont pas vides, mais on ne s'y intéresse pas



# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau **tabPile** permettant de stocker les éléments de la pile.
  - La taille maximale **N** du tableau
  - Un entier **tete** indiquant l'indice du haut de la pile (remplissage).
- Dépiler:

10	2	25	103	55	16	147
----	---	----	-----	----	----	-----

tete

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau **tabPile** permettant de stocker les éléments de la pile.
  - La taille maximale **N** du tableau
  - Un entier **tete** indiquant l'indice du haut de la pile (remplissage).
- Dépiler:

10	2	25	103	55	16	147
----	---	----	-----	----	----	-----

↑  
**tete**

```
val ← tabPile[tete]  
tete ← tete-1  
retourner val
```

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau **tabPile** permettant de stocker les éléments de la pile.
  - La taille maximale **N** du tableau
  - Un entier **tete** indiquant l'indice du haut de la pile (remplissage).
- Empiler : Ajouter 40

10	2	25	103	55	16	147
----	---	----	-----	----	----	-----

↑  
**tete**

# Piles : Implémentation statique

- Une pile implémentée en statique (tableau) est une structure possédant trois variables :
  - Le tableau `tabPile` permettant de stocker les éléments de la pile.
  - La taille maximale `N` du tableau
  - Un entier `tete` indiquant l'indice du haut de la pile (remplissage).
- Empiler : Ajouter 40

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
`tete`

```
tete ← tete+1  
tabPile[tete] ← 40
```

# Piles : Implémentation statique

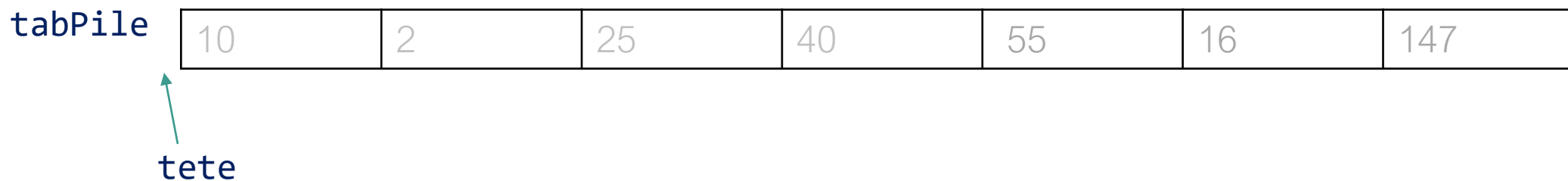
- Initialisation de la pile statique :

```
FONCTION creationPileStat(): PileStat
VARIABLE
    pile: PileStat
DEBUT
    tete(pile) ← ???
    RETOURNER pile
FIN
```

# Piles : Implémentation statique

- Initialisation de la pile statique :

```
FONCTION creationPileStat(): PileStat
VARIABLE
    pile: PileStat
DEBUT
    tete(pile) ← -1      // la pile est vide
    RETOURNER pile
FIN
```



# Piles : Implémentation statique

- Vérification de l'intégrité de la structure **PileStat** :
  - A chaque fois que nous allons utiliser une structure de type **PileStat** dans une fonction, il faudra s'assurer qu'elle n'a pas été corrompue :

```
FONCTION verificationPileStat(pile : pointeur sur PileStat) : entier
VARIABLE
    result ← 0 : entier    // resultat (0 signifie OK)
DEBUT
    SI (???) ALORS
        result ← -1    // données d'entrées invalides
    SINON SI (???) ALORS
        result ← -2    // la structure est corrompue
    FIN SI
    RETOURNER result
FIN
```

# Piles : Implémentation statique

- Vérification de l'intégrité de la structure **PileStat** :
  - A chaque fois que nous allons utiliser une structure de type **PileStat** dans une fonction, il faudra s'assurer qu'elle n'a pas été corrompue :

```
FONCTION verificationPileStat(pile : pointeur sur PileStat) : entier
VARIABLE
    result ← 0 : entier    // resultat (0 signifie OK)
DEBUT
    SI (pile EST EGAL A NULL) ALORS
        result ← -1    // données d'entrées invalides
    SINON SI (???) ALORS
        result ← -2    // la structure est corrompue
    FIN SI
    RETOURNER result
FIN
```



# Piles : Implémentation statique

- Vérification de l'intégrité de la structure **PileStat** :
  - A chaque fois que nous allons utiliser une structure de type **PileStat** dans une fonction, il faudra s'assurer qu'elle n'a pas été corrompue :

```
FONCTION verificationPileStat(pile : pointeur sur PileStat) : entier
VARIABLE
    result ← 0 : entier    // resultat (0 signifie OK)
DEBUT
    SI (pile EST EGAL A NULL) ALORS
        result ← -1    // données d'entrées invalides
    SINON SI (taille(pile) EST INF. OU EGAL A 0)
        OU (???) ALORS
            result ← -2    // la structure est corrompue
    FIN SI
    RETOURNER result
FIN
```

# Piles : Implémentation statique

- Vérification de l'intégrité de la structure **PileStat** :
  - A chaque fois que nous allons utiliser une structure de type **PileStat** dans une fonction, il faudra s'assurer qu'elle n'a pas été corrompue :

```
FONCTION verificationPileStat(pile : pointeur sur PileStat) : entier
VARIABLE
    result ← 0 : entier    // resultat (0 signifie OK)
DEBUT
    SI (pile EST EGAL A NULL) ALORS
        result ← -1    // données d'entrées invalides
    SINON SI (taille(pile) EST INF. OU EGAL A 0)
        OU (tete(pile) EST INF. STRICT. A -1)
        OU (???) ALORS
            result ← -2    // la structure est corrompue
    FIN SI
    RETOURNER result
FIN
```

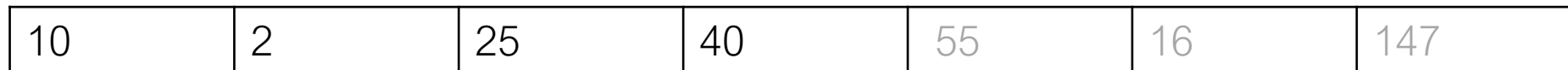
# Piles : Implémentation statique

- Vérification de l'intégrité de la structure **PileStat** :
  - A chaque fois que nous allons utiliser une structure de type **PileStat** dans une fonction, il faudra s'assurer qu'elle n'a pas été corrompue :

```
FONCTION verificationPileStat(pile : pointeur sur PileStat) : entier
VARIABLE
    result ← 0 : entier    // resultat (0 signifie OK)
DEBUT
    SI (pile EST EGAL A NULL) ALORS
        result ← -1    // données d'entrées invalides
    SINON SI (taille(pile) EST INF. OU EGAL A 0)
        OU (tete(pile) EST INF. STRICT. A -1)
        OU (tete(pile) EST SUP. STRICT. A taille(pile)-1) ALORS
            result ← -2    // la structure est corrompue
    FIN SI
    RETOURNER result
FIN
```

# Piles : Implémentation statique

- Empilage :



↑  
tete

# Piles : Implémentation statique

- Empilage :

```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← ???
    SI (???) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            tete(pile) ← ???
            tabPile(pile)[tete] ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Empilage :

```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (???) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            tete(pile) ← ???
            tabPile(pile)[tete] ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Empilage :

```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (result EST EGAL A 0) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            tete(pile) ← ???
            tabPile(pile)[tete] ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Empilage :

```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A taille(pile)-1) ALORS
            result ← 1          // la liste est déjà pleine
        SINON
            tete(pile) ← ???
            tabPile(pile)[tete] ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete



# Piles : Implémentation statique

- Empilage :

```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A taille(pile)-1) ALORS
            result ← 1          // la liste est déjà pleine
        SINON
            tete(pile) ← tete(pile) + 1 // décalage de la tête
            tabPile(pile)[tete] ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Empilage :

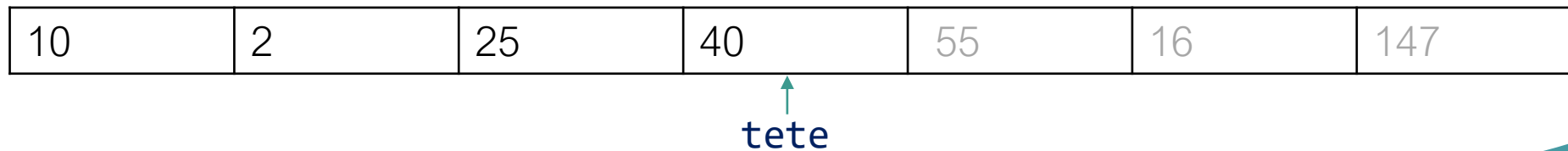
```
FONCTION empilagePileStat(pile : pointeur sur PileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier          // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A taille(pile)-1) ALORS
            result ← 1          // la liste est déjà pleine
        SINON
            tete(pile) ← tete(pile) + 1 // décalage de la tête
            tabPile(pile)[tete] ← elmt // stockage de l'élément
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	<b>'elmt'</b>	16	147
----	---	----	----	---------------	----	-----

↑  
tete

# Piles : Implémentation statique

- Dépilage :



# Piles : Implémentation statique

- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (???) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (???) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            *pelmt ← ???
            tete(pile) ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (pelmt EST EGAL A NULL) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (???) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            *pelmt ← ???
            tete(pile) ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (pelmt EST EGAL A NULL) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (result EST EGAL A 0) ALORS
        SI (???) ALORS
            result ← 1
        SINON
            *pelmt ← ???
            tete(pile) ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (pelmt EST EGAL A NULL) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A -1) ALORS
            result ← 1 // la liste est déjà vide
        SINON
            *pelmt ← ???
            tete(pile) ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

↑  
tete

# Piles : Implémentation statique

- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (pelmt EST EGAL A NULL) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A -1) ALORS
            result ← 1 // la liste est déjà vide
        SINON
            *pelmt ← tabPile(pile)[tete] // récupération élément
            tete(pile) ← ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

10	2	25	40	55	16	147
----	---	----	----	----	----	-----

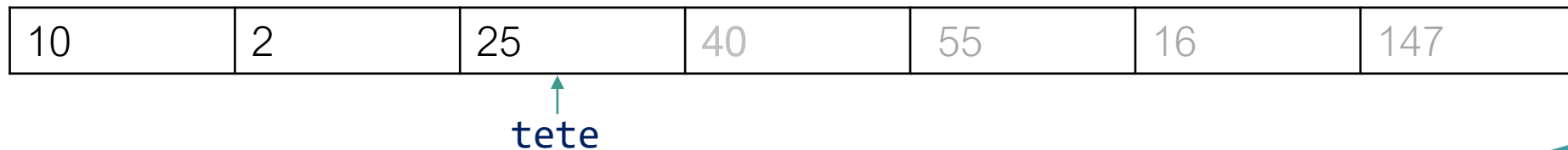
↑  
tete



# Piles : Implémentation statique

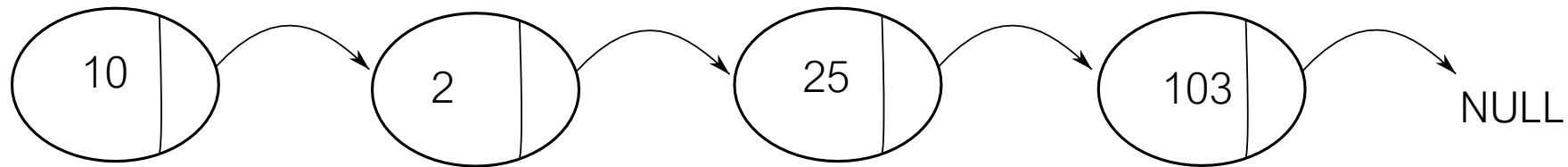
- Dépilage :

```
FONCTION depilagePileStat(pile : pointeur sur PileStat, pelmt : pointeur sur Element) : entier
VARIABLE
    result ← 0 : entier // résultat de l'empilage (0 signifie OK)
DEBUT
    result ← verificationPileStat(pile)
    SI (pelmt EST EGAL A NULL) ALORS
        result ← -1 // données d'entrées invalides
    FIN SI
    SI (result EST EGAL A 0) ALORS
        SI (tete(pile) EST EGAL A -1) ALORS
            result ← 1 // la liste est déjà vide
        SINON
            *pelmt ← tabPile(pile)[tete] // récupération élément
            tete(pile) ← tete(pile) - 1 // décalage de la tête
        FIN SI
    FIN SI
    RETOURNER result
FIN
```



# Piles : Implémentation dynamique

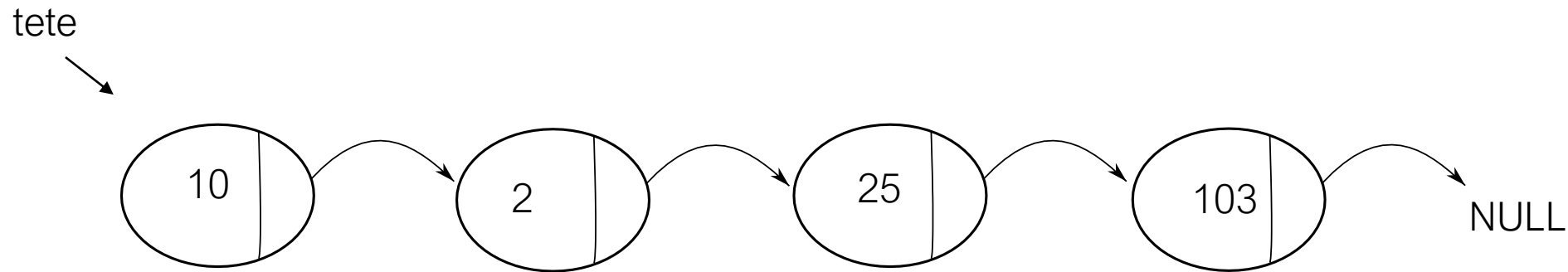
- La pile dynamique est une liste chaînée :



Quel maillon est la tête de la pile?

# Piles : Implémentation dynamique

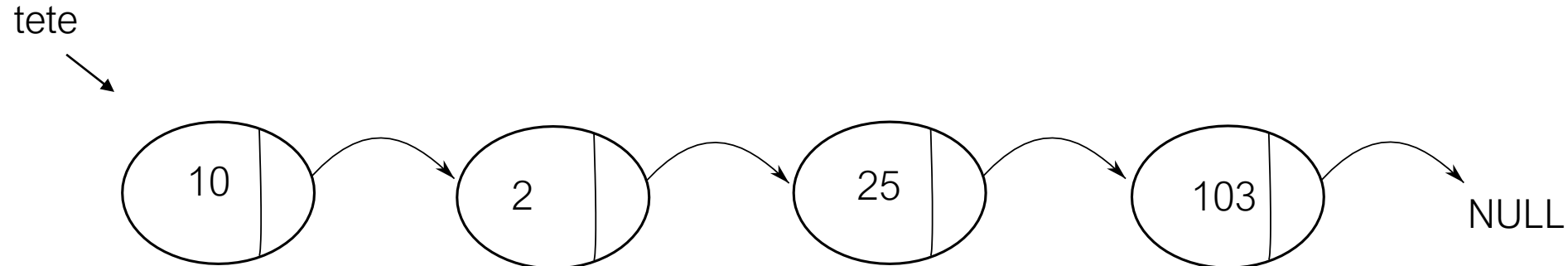
- La pile dynamique est une liste chaînée :



- La tête de la pile est un pointeur indiquant le premier chaînon de la liste chaînée.

# Piles : Implémentation dynamique

- La pile dynamique est une liste chaînée :



```
Structure Chainon :
```

```
    elmt : Element
```

```
    suivant : pointeur sur structure Chainon
```

```
Fonction creationPileDyn(): pointeur sur Chainon
```

```
VARIABLE
```

```
    tete : pointeur sur Chainon
```

```
DEBUT
```

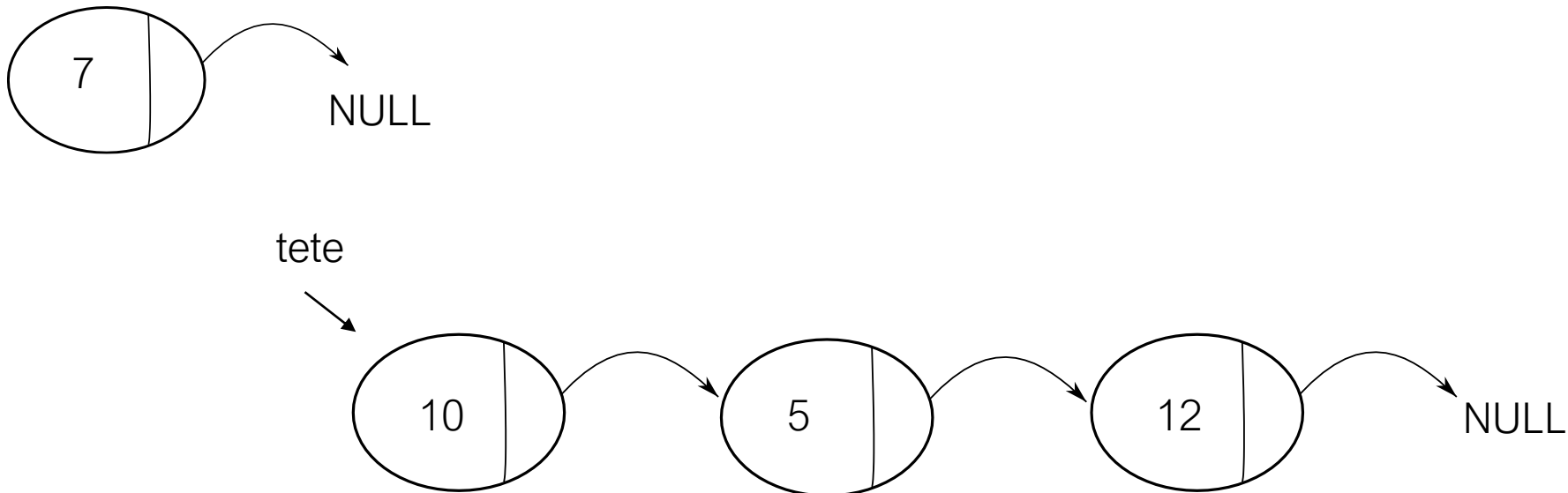
```
    tete <- NULL // pas encore d'element
```

```
    RETOURNER tete
```

```
FIN
```

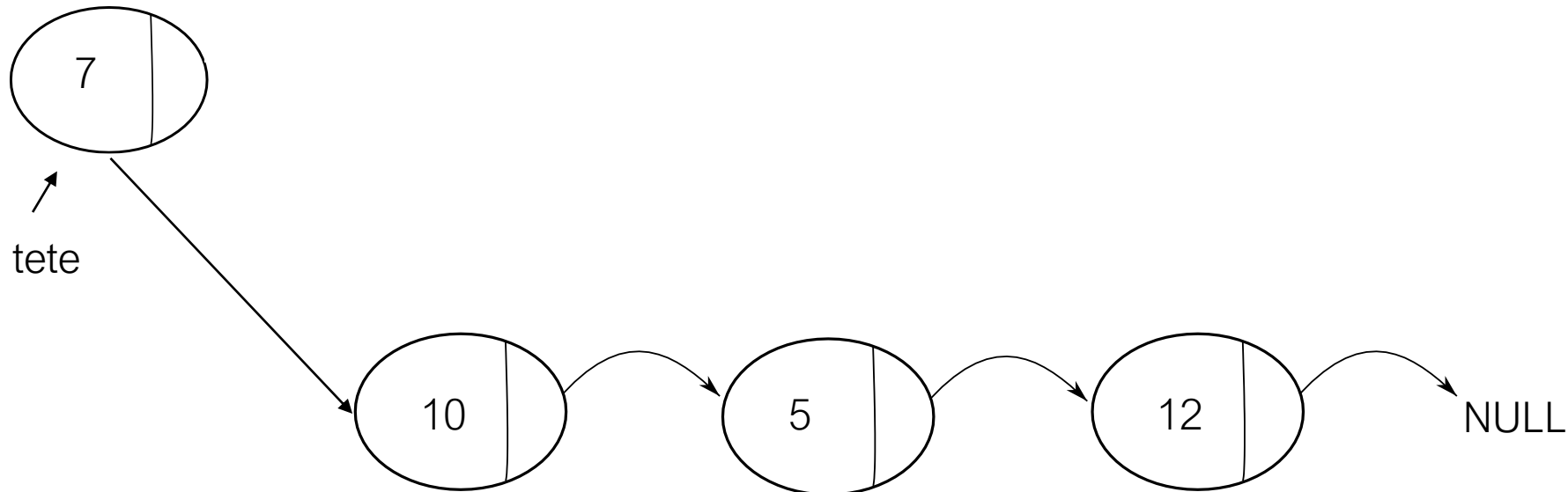
# Piles : Implémentation dynamique

- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Empilage :



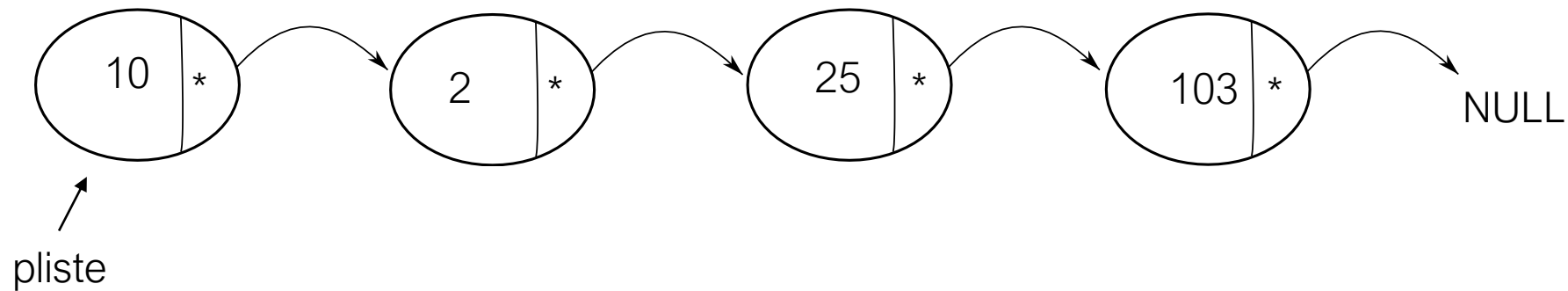
# Piles : Implémentation dynamique

- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Empilage : ajout d'un **Chainon** en début de liste (**insertDebut**)



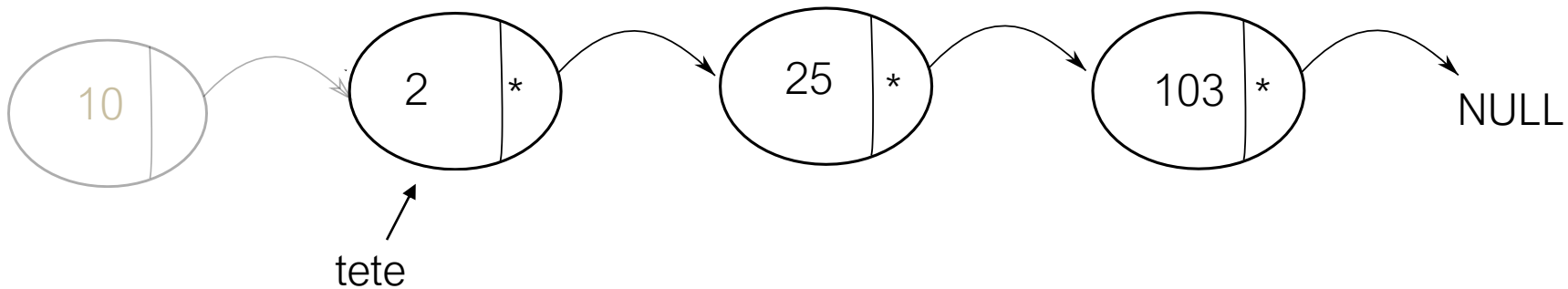
# Piles : Implémentation dynamique

- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Empilage : ajout d'un **Chainon** en début de liste (**insertDebut**).
  2. Dépilage :



# Piles : Implémentation dynamique

- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Empilage : ajout d'un **Chainon** en début de liste (**insertDebut**).
  2. Dépilage : suppression d'un **Chainon** en début de liste (**suppDebut**) et on récupère de la valeur de ce chaînon.





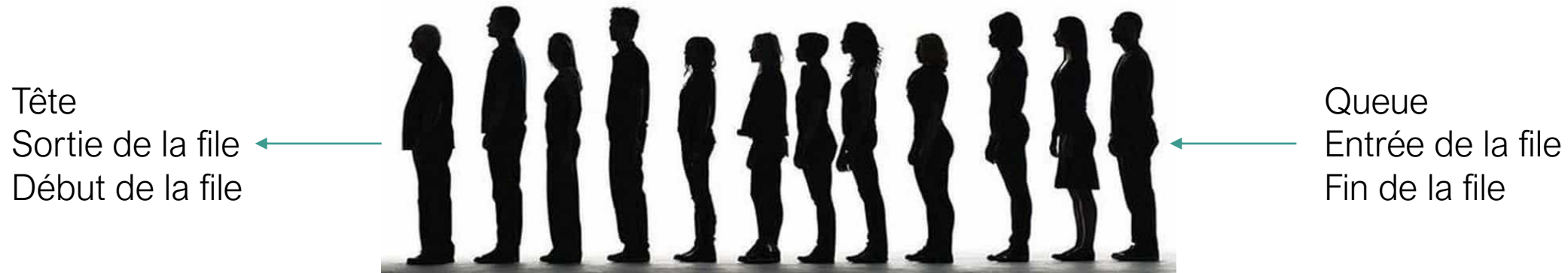
# Piles : Implémentation dynamique

- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Empilage : ajout d'un **Chainon** en début de liste (**insertDebut**).
  2. Dépilage : suppression d'un **Chainon** en début de liste (**suppDebut**) et on récupère de la valeur de ce chaînon.
- Une pile dynamique est une liste chaînée dont les éléments ne peuvent être ajoutés/retirés qu'en début de chaîne.

## II. Files

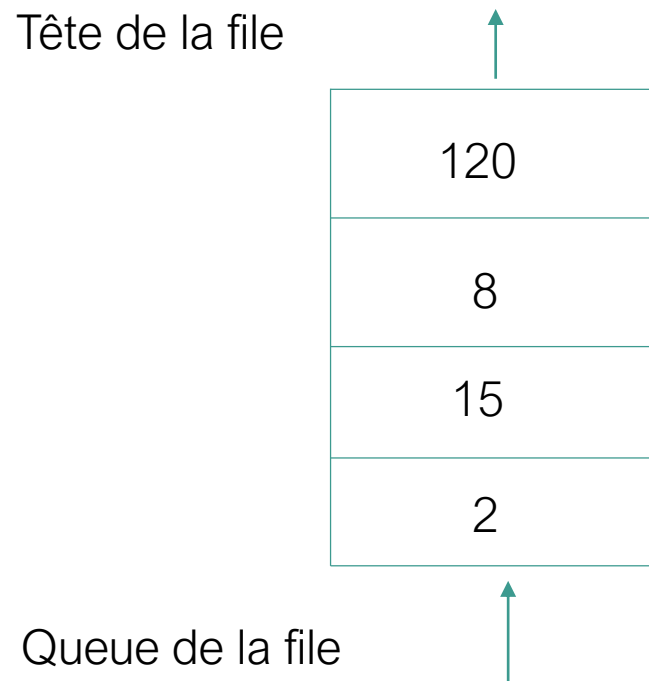
# Files : principe

- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- On l'appelle aussi **FIFO** : « first in, first out »
- Pensez à une file d'attente !



# Files : principe

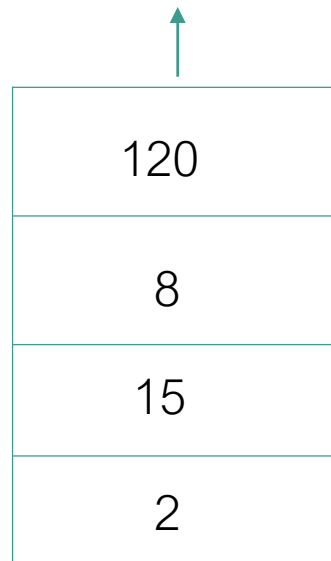
- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple



# Files : principe

- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple : ajouter un élément : enfiler

Tête de la file

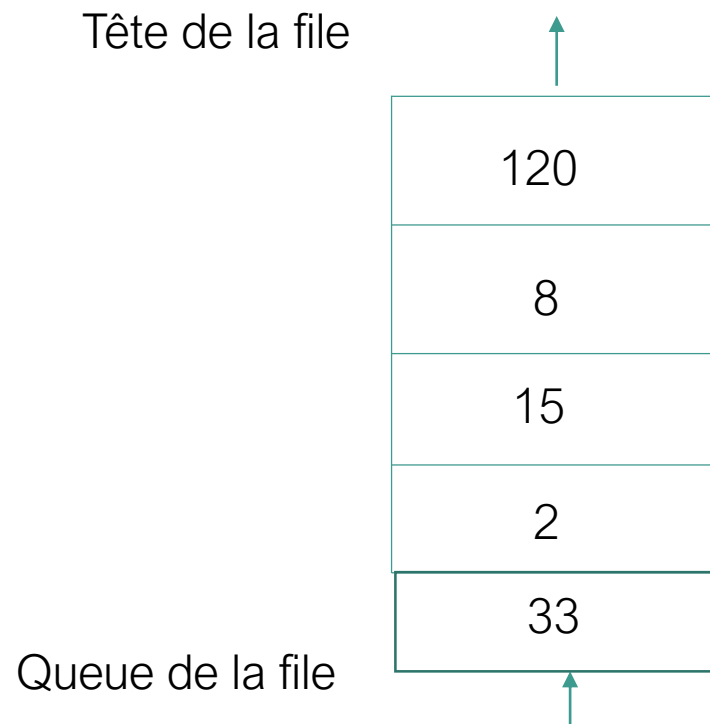


Queue de la file



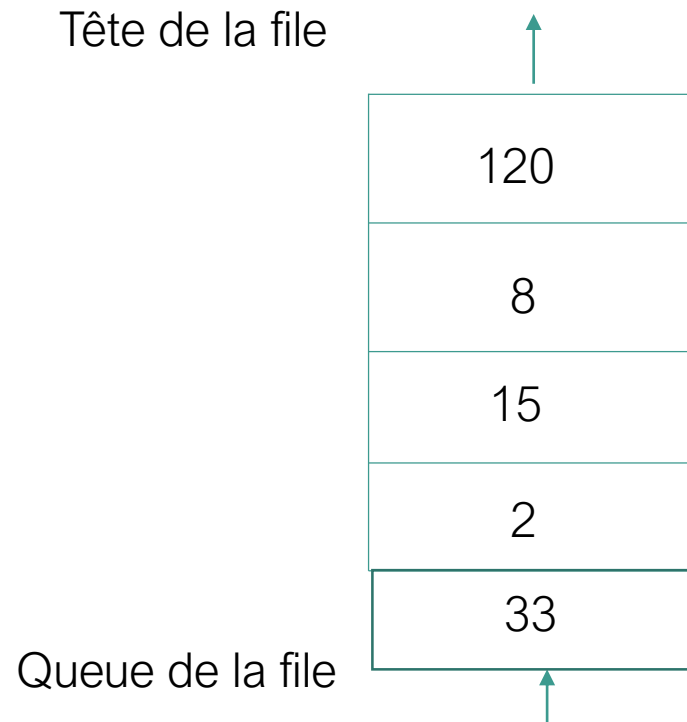
# Files : principe

- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple : enfiler : on place l'élément en bas de la file



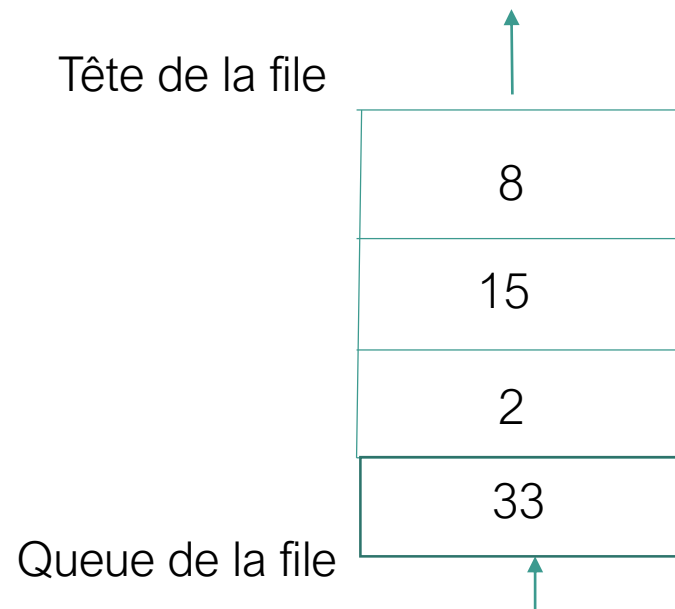
# Files : principe

- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple : supprimer un élément : défiler



# Files : principe

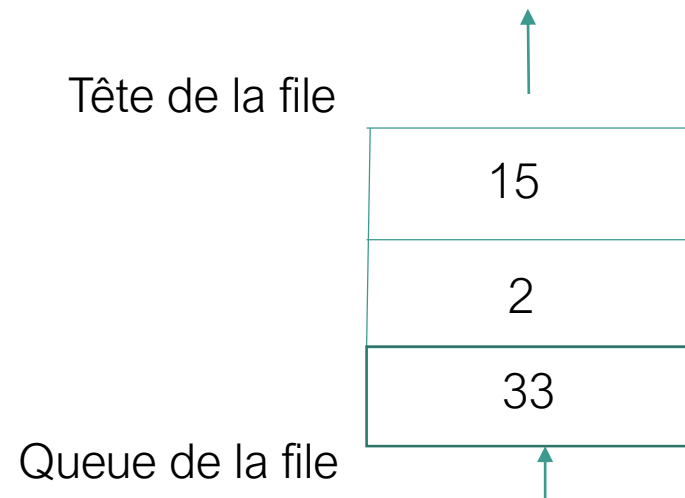
- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple : défiler: l'élément est supprimé en haut de la file





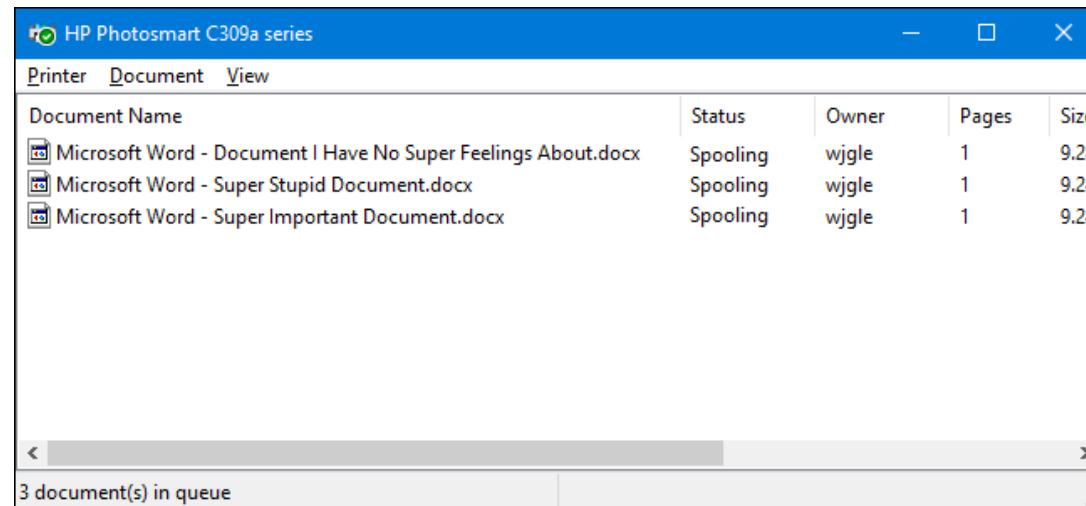
# Files : principe

- Une **file** est une liste d'éléments :
  - Un élément est toujours ajouté en bas de la file
  - Un élément est toujours retiré en haut de la file
- Exemple : défiler : l'élément est supprimé en haut de la file



# Files : utilisation

- Dans l'ordinateur, les files sont principalement utilisées pour stocker des données qui doivent être traitées dans l'ordre d'arrivée.
- Exemple : les tâches d'impression !

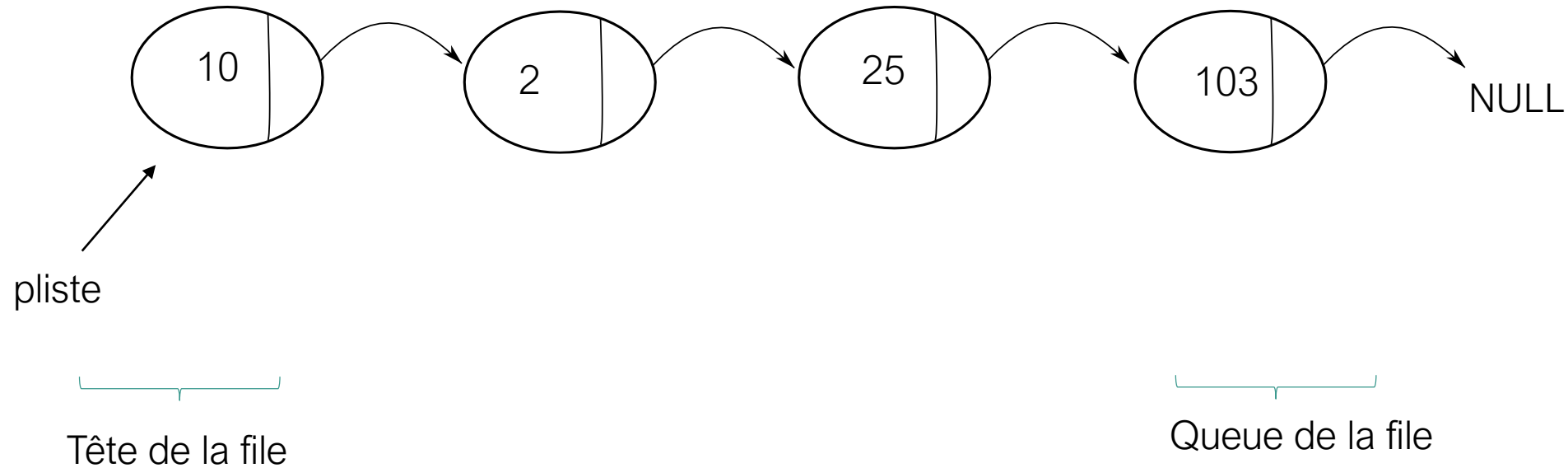


Document Name	Status	Owner	Pages	Size
Microsoft Word - Document I Have No Super Feelings About.docx	Spooling	wjgle	1	9.28
Microsoft Word - Super Stupid Document.docx	Spooling	wjgle	1	9.24
Microsoft Word - Super Important Document.docx	Spooling	wjgle	1	9.24

- En algorithmie la File permet de simuler de nombreux problèmes comme la gestion de file d'attente.

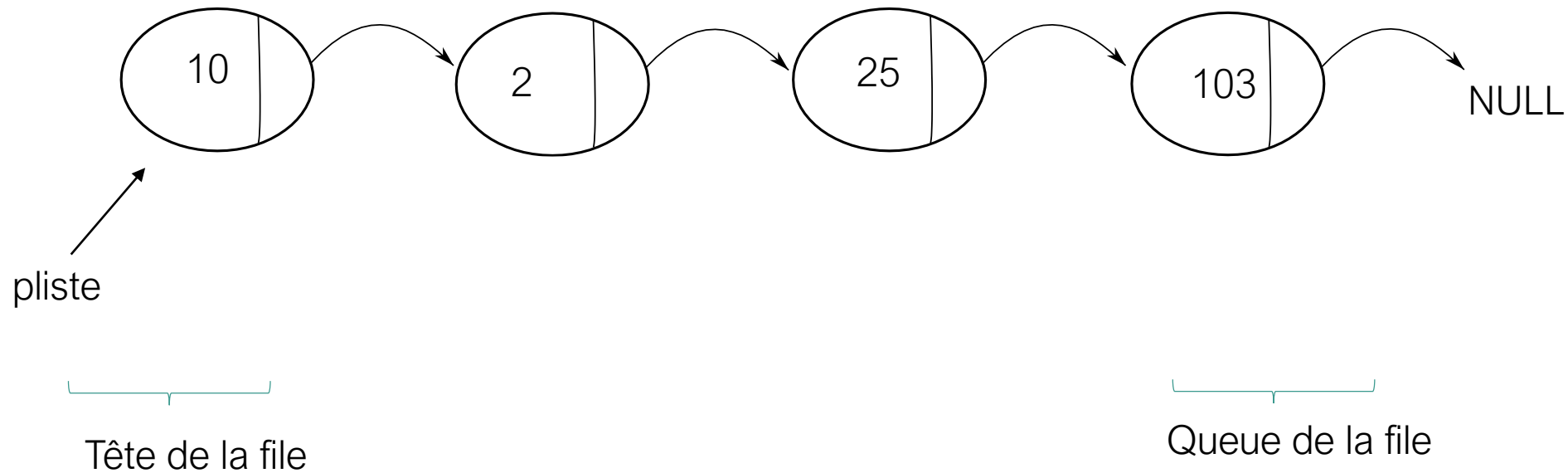
# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.



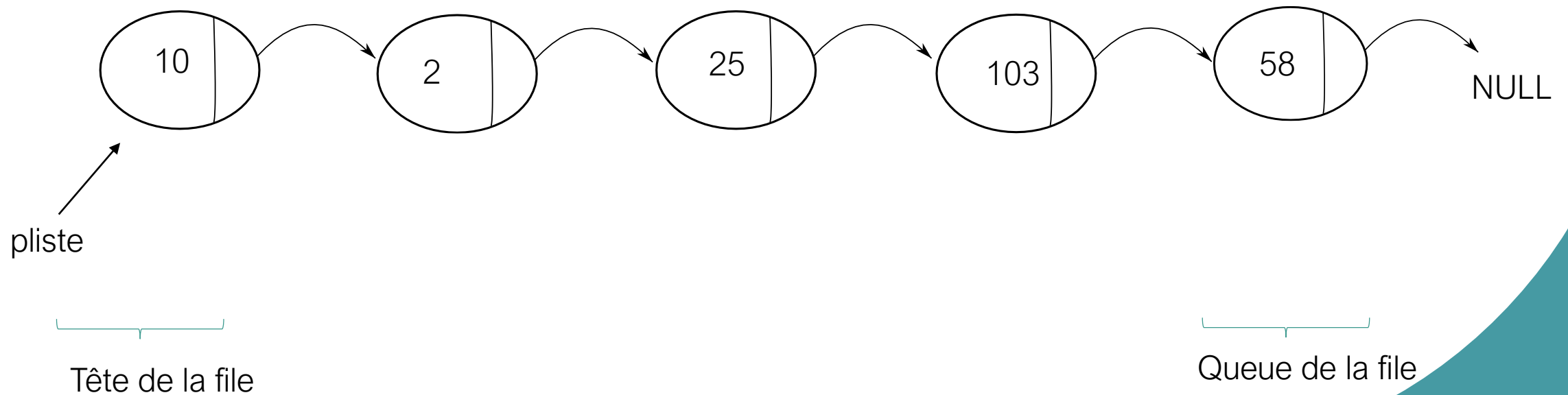
# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.
- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Enfilage :



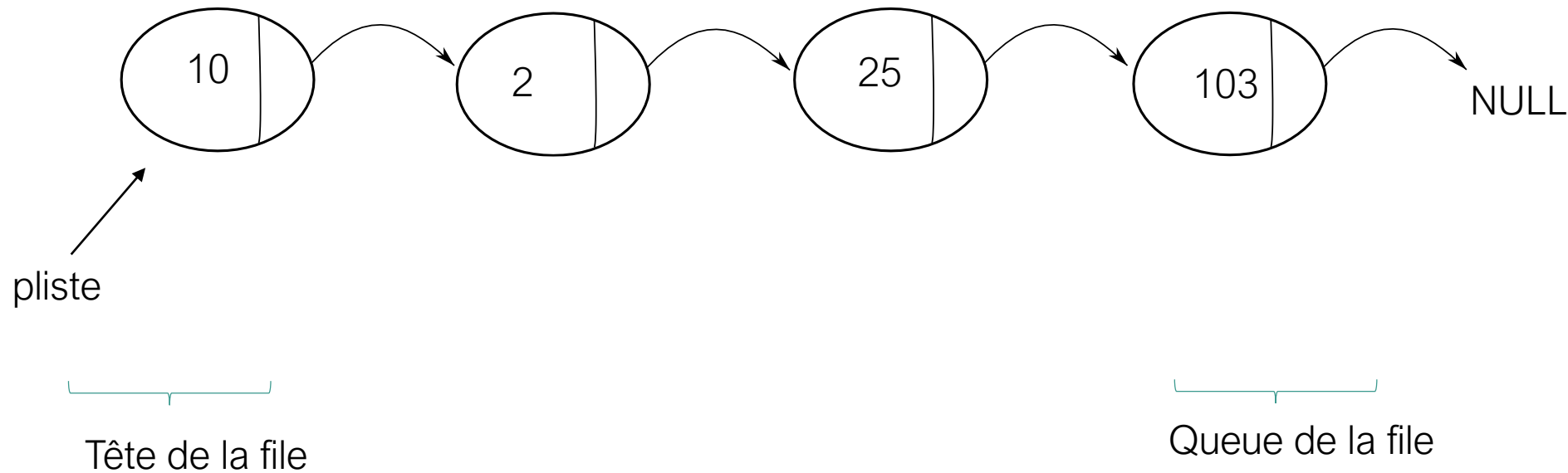
# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.
- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Enfilage : ajout d'un élément en fin de liste (fin de la file / en queue)



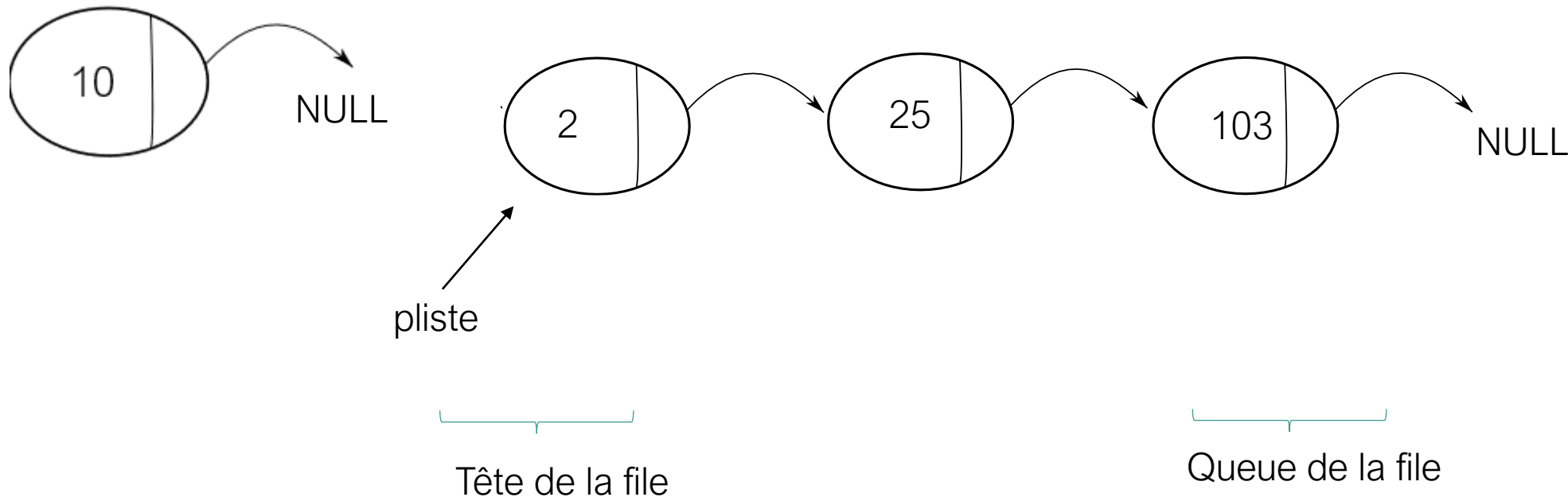
# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.
- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Enfilage : ajout d'un élément en fin de liste (fin de la file / en queue)
  2. Défilage :



# Files dynamiques

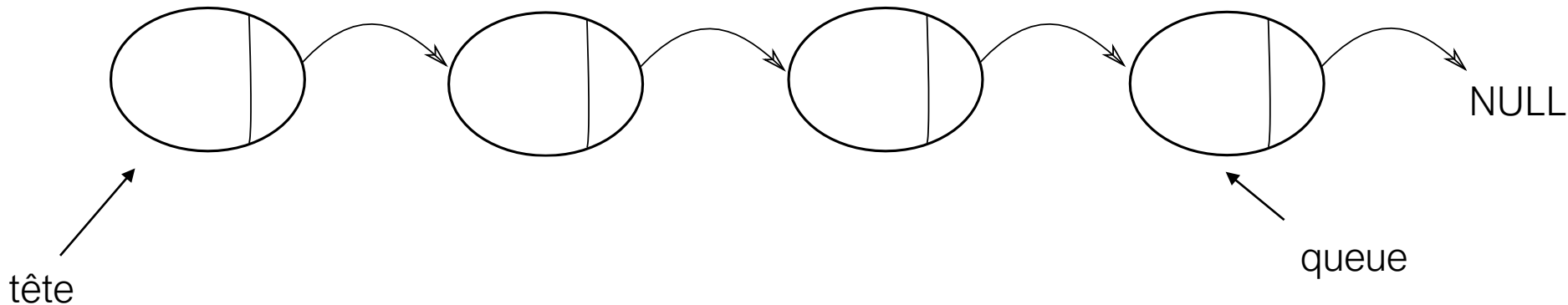
- Les files dynamiques sont gérées à partir de listes chaînées.
- A quelles opérations sur une liste chaînée sont équivalentes les opérations suivantes :
  1. Enfilage : ajout d'un élément en fin de liste (fin de la file / en queue)
  2. Défilage : suppression d'un élément en début de liste (début de file / en tête)  
+ récupération de l'Element



# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.
- Pour ne pas avoir besoin de parcourir la liste chaînée, la pile va exploiter deux pointeurs:

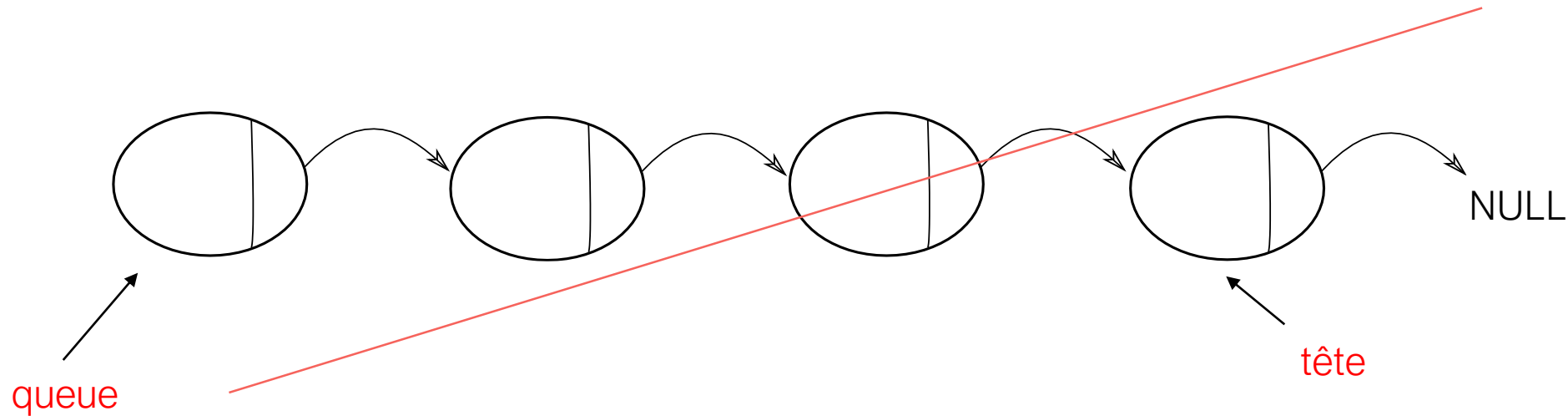
```
Structure FileDyn :  
  tete : pointeur sur Chainon  
  Queue : pointeur sur Chainon
```





# Files dynamiques

- Les files dynamiques sont gérées à partir de listes chaînées.
- Pour ne pas avoir besoin de parcourir la liste chaînée, la pile va exploiter deux pointeurs:

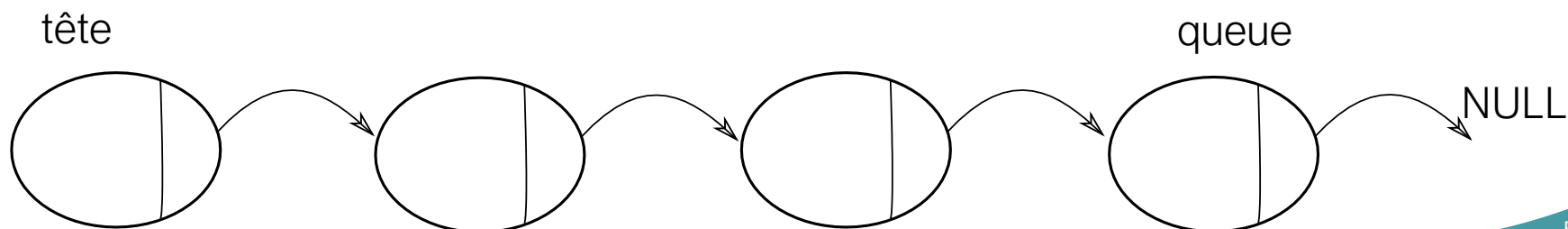


L'inverse est moins optimal !

# Files dynamiques

- Vérification de l'intégrité de la structure FileDyn

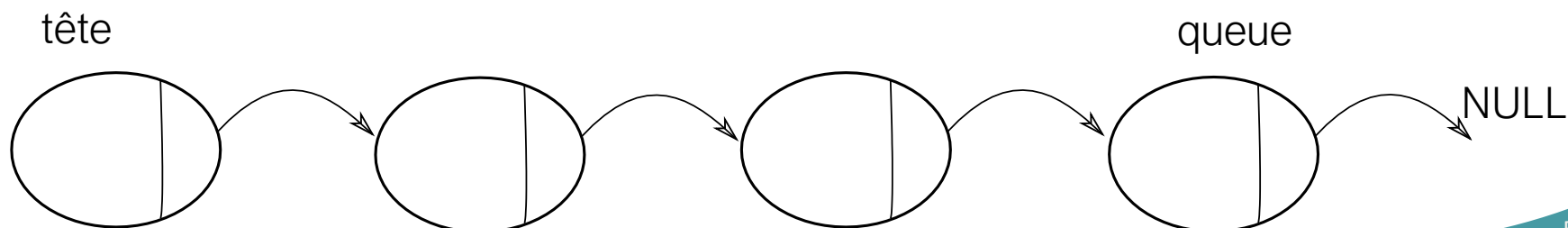
```
FONCTION verificationFileDyn(file: pointeur sur FileDyn) : entier
VARIABLE
    result ← 0 : entier // retour (0 signifie OK)
DEBUT
    SI(???) ALORS // pointeur structure NULL
        result ← -1
    SINON SI(???) ALORS // structure corrompue #1
        result ← -2
    SINON SI(???) ALORS // structure corrompue #2
        result ← -3
    FIN SI
RETOURNER result
FIN
```



# Files dynamiques

- Vérification de l'intégrité de la structure FileDyn

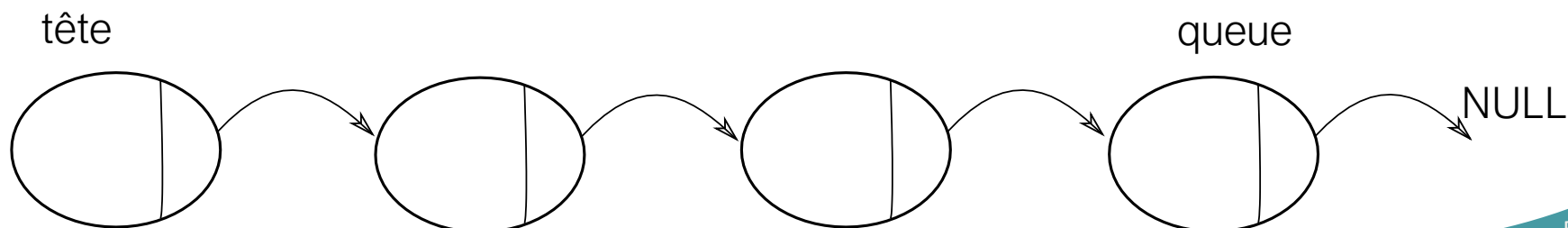
```
FONCTION verificationFileDyn(file: pointeur sur FileDyn) : entier
VARIABLE
    result ← 0 : entier // retour (0 signifie OK)
DEBUT
    SI(file EST EGAL A NULL) ALORS // pointeur structure NULL
        result ← -1
    SINON SI(???) ALORS // structure corrompue #1
        result ← -2
    SINON SI(???) ALORS // structure corrompue #2
        result ← -3
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

- Vérification de l'intégrité de la structure FileDyn

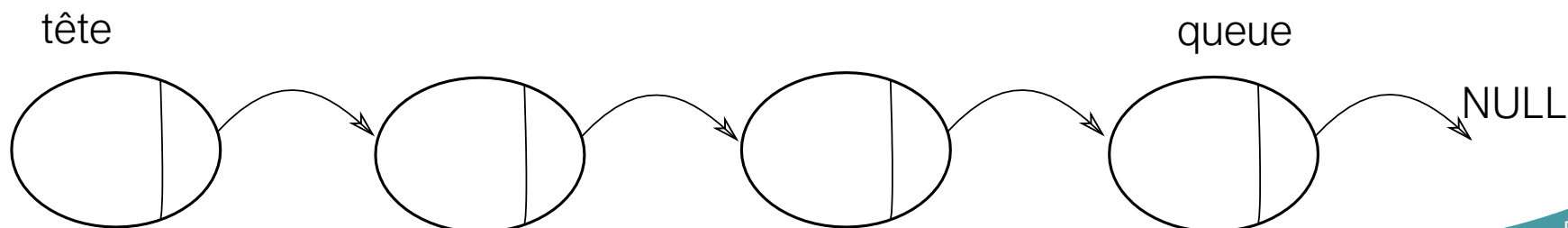
```
FONCTION verificationFileDyn(file: pointeur sur FileDyn) : entier
VARIABLE
    result ← 0 : entier // retour (0 signifie OK)
DEBUT
    SI(file EST EGAL A NULL) ALORS
        result ← -1 // pointeur structure NULL
    SINON SI((tete(file) EGAL A NULL) DIFFERENT DE (queue(file) EGAL A NULL)) ALORS
        result ← -2 // structure corrompue #1
    SINON SI(???) ALORS
        result ← -3 // structure corrompue #2
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

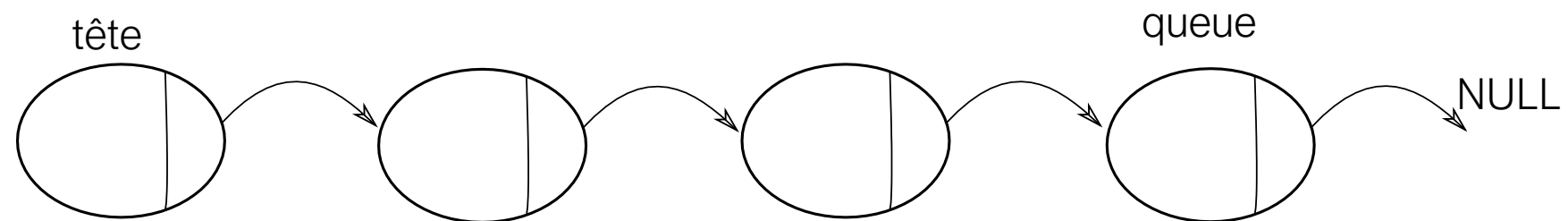
- Vérification de l'intégrité de la structure FileDyn

```
FONCTION verificationFileDyn(file: pointeur sur FileDyn) : entier
VARIABLE
    result ← 0 : entier // retour (0 signifie OK)
DEBUT
    SI(file EST EGAL A NULL) ALORS
        result ← -1 // pointeur structure NULL
    SINON SI((tete(file) EGAL A NULL) DIFFERENT DE (queue(file) EGAL A NULL)) ALORS
        result ← -2 // structure corrompue #1
    SINON SI(suivant(queue(file)) DIFFERENT DE NULL) ALORS
        result ← -3 // structure corrompue #2
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

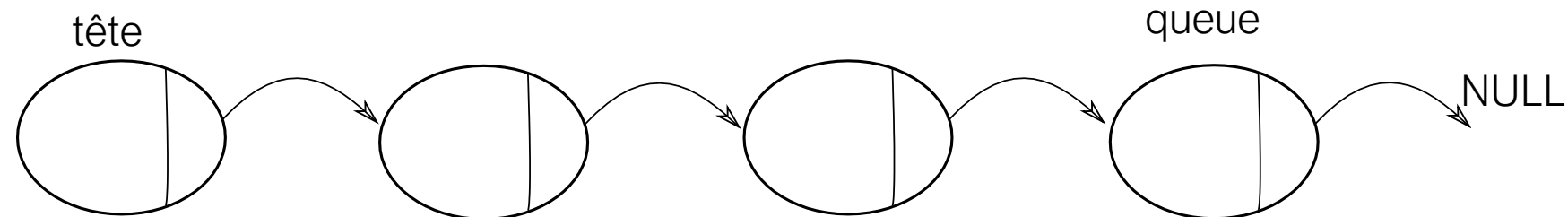
- Enfilage



# Files dynamiques

- Enfilage

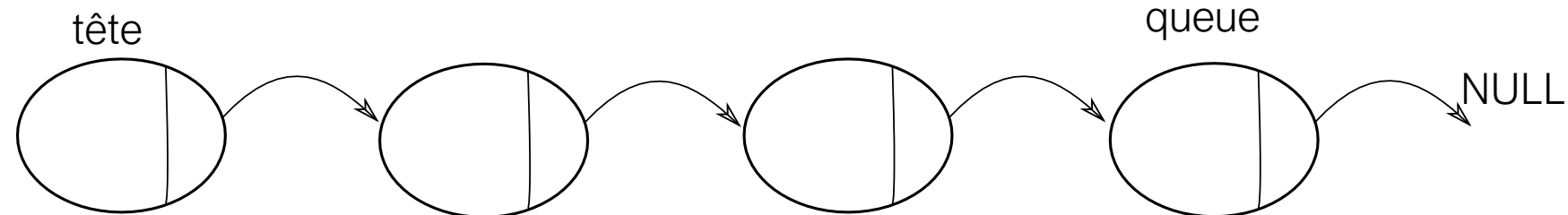
```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ←- ???
    SI (???) ALORS
        nouveau ← ???
        elmt(nouveau) ← ???
        SI (???) // si la file est vide...
            ???
            ???
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (???) ALORS
        nouveau ← ???
        elmt(nouveau) ← ???
        SI (???) // si la file est vide...
            ???
            ???
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

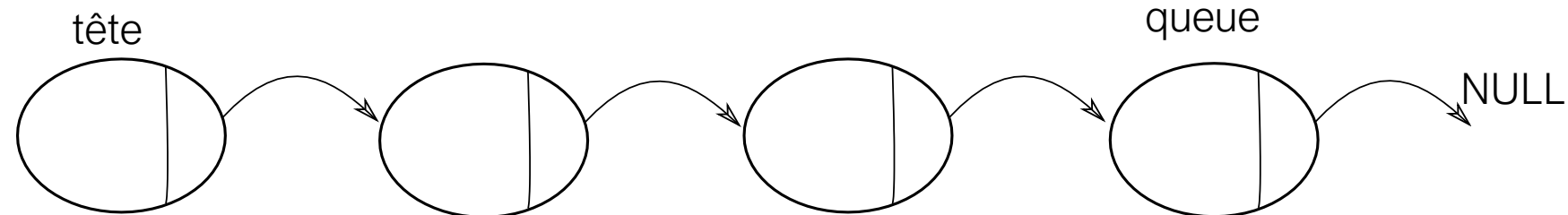




# Files dynamiques

- Enfilage

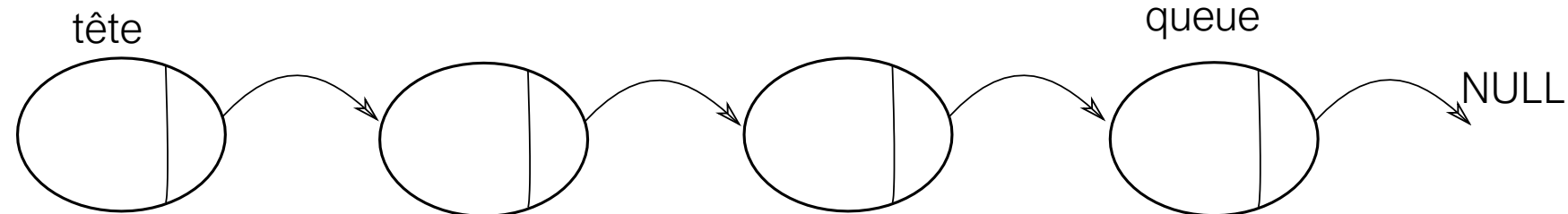
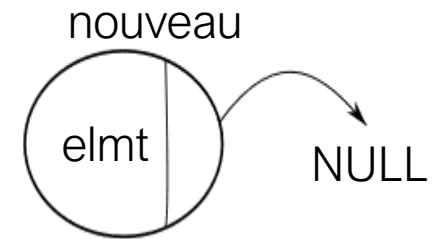
```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← ???
        elmt(nouveau) ← ???
        SI (???) // si la file est vide...
            ???
            ???
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← ???
        SI (???) // si la file est vide..
            ???
            ???
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

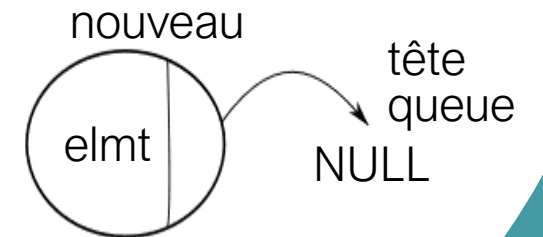


# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← elmt
        SI (queue(file) EST EGAL A NULL)
            ???
            ???
        SINON
            ???
    FIN SI
    RETOURNER result
FIN
```

// si la file est vide..

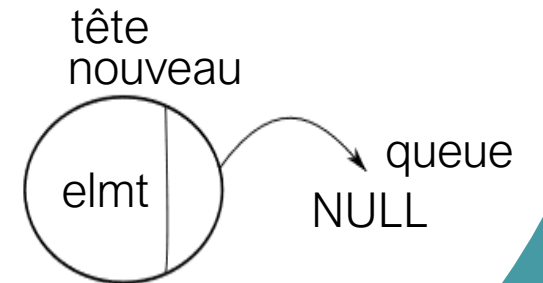


# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← elmt
        SI (queue(file) EST EGAL A NULL)
            tete(file) ← nouveau
            ???
        SINON
            ???
    FIN SI
    RETOURNER result
FIN
```

// si la file est vide..

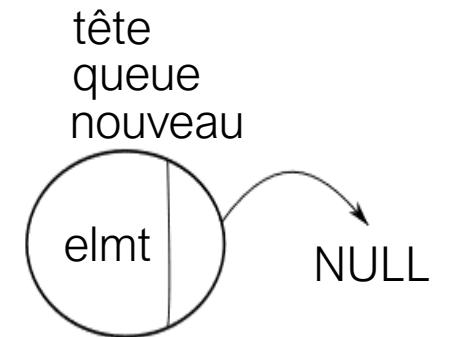


# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← elmt
        SI (queue(file) EST EGAL A NULL)
            tete(file) ← nouveau
            queue(file) ← nouveau
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

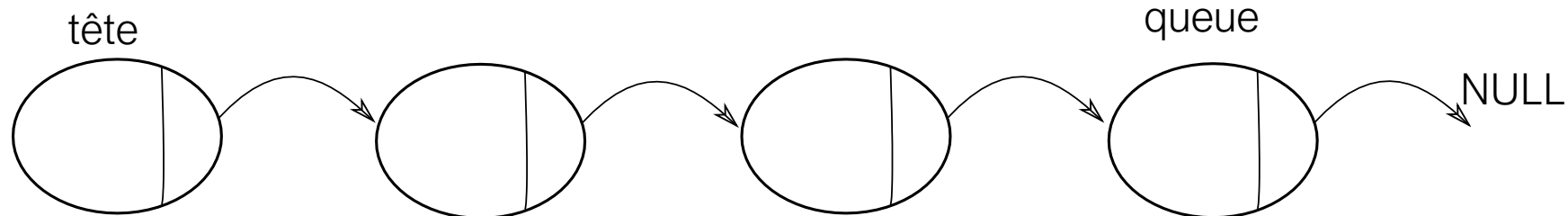
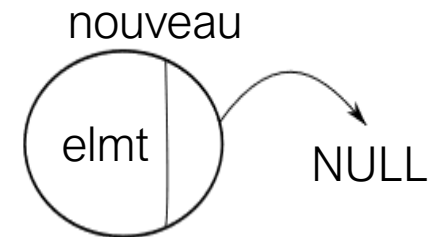
// si la file est vide..



# Files dynamiques

- Enfilage

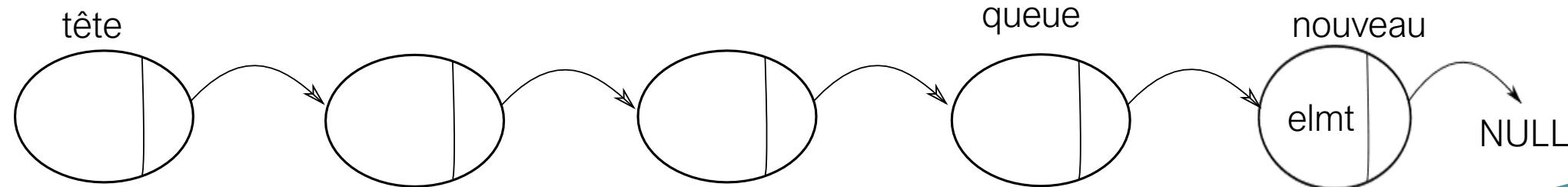
```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← elmt
        SI (queue(file) EST EGAL A NULL) // si la file est vide..
            tete(file) ← nouveau
            queue(file) ← nouveau
        SINON
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```



# Files dynamiques

- Enfilage

```
FONCTION enfilerDyn(file: pointeur sur FileDyn, elmt : Element) : entier
VARIABLE
    nouveau      : pointeur sur Chainon
    result ← 0 : entier
DEBUT
    result ← verificationFileDyn(file)
    SI (result EST SUP. STRICT. A -2) ALORS
        nouveau ← creationChainon()
        elmt(nouveau) ← elmt
        SI (queue(file) EST EGAL A NULL) // si la file est vide..
            tete(file) ← nouveau
            queue(file) ← nouveau
        SINON
            suivant(queue(file)) ← nouveau
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

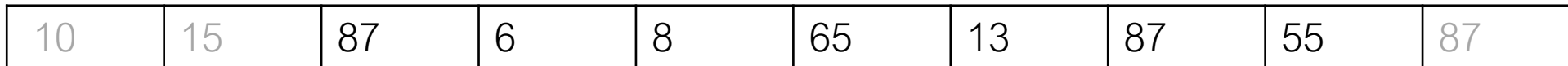


# Files statiques

- La file statique est une file réalisée avec un tableau.
- Le tête et la queue de la file vont être représentées par des indices de ces tableaux indiquant leur positions.

```
Structure FileStat :  
    tabPile : tableau d'Element de taille N  
    tete : entier  
    queue : entier
```

- Exemple :



tete ← 2  
queue ← 8



# Files statiques

- Les opérations de la file statique :
  - Enfiler

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION enfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    ???
    SI(???) ALORS
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION enfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(???) ALORS // données intègres
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION enfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION enfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(queue(file) EST EGAL A taille(file)-1) ALORS
            result ← 1 // file déjà pleine
        SINON
            ???
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete										queue
10	15	87	6	8	65	13	72	55	87	

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION enfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(queue(file) EST EGAL A taille(file)-1) ALORS
            result ← 1 // file déjà pleine
        SINON
            ???
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION EnfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(queue(file) EST EGAL A taille(file)-1) ALORS
            result ← 1 // file déjà pleine
        SINON
            queue(file) ← queue(file) + 1 // décalage queue
            ???
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete					queue				
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler

```
FONCTION EnfilerStat( file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur pile et du contenu de la structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(queue(file) EST EGAL A taille(file)-1) ALORS
            result ← 1 // file déjà pleine
        SINON
            queue(file) ← queue(file) + 1 // décalage queue
            tabFile(file)[queue] ← elmt // stockage élément
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

tete				queue					
10	15	87	6	8	65	13	72	elmt	87



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

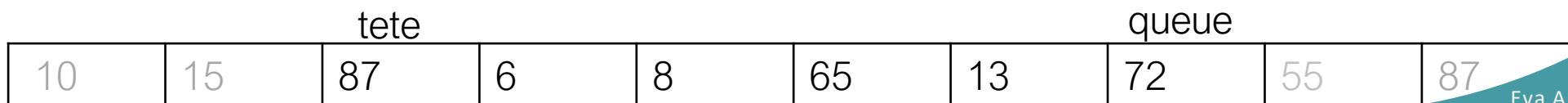
```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    ???
    SI(???) ALORS
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
    FIN SI
    RETOURNER result
FIN
```



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

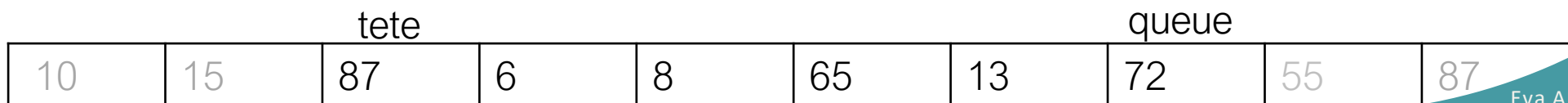
```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(???) ALORS
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
    FIN SI
    RETOURNER result
FIN
```



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(???) ALORS
            result ← 1
        SINON
            ???
            ???
    FIN SI
    RETOURNER result
FIN
```



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(tete(file) EST EGAL A queue(file) + 1) ALORS
            result ← 1 // file déjà vide
        SINON
            ???
            ???
    FIN SI
    RETOURNER result
FIN
```



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(tete(file) EST EGAL A queue(file) + 1) ALORS
            result ← 1 // file déjà vide
        SINON
            ???
            ???
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(tete(file) EST EGAL A queue(file) + 1) ALORS
            result ← 1 // file déjà vide
        SINON
            *pElmt ← tete(file) // récupération élément
            ???
    FIN SI
    RETOURNER result
FIN
```

tete			queue						
10	15	87	6	8	65	13	72	55	87

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler

```
FONCTION defilerStat( file : pointeur sur FileStat, pElmt : pointeur sur
element) : entier
DEBUT
    result ← vérification des pointeurs et contenu de structure
    SI(result EST EGAL A 0) ALORS // données intègres
        SI(tete(file) EST EGAL A queue(file) + 1) ALORS
            result ← 1 // file déjà vide
        SINON
            *pElmt ← tete(file) // récupération élément
            tete(file) ← tete(file) + 1 // décalage tete
    FIN SI
    RETOURNER result
FIN
```

			tete					queue			
10	15	87	6	8	65	13	72	55	87		



# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler
  - Initialisation de la file

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler
  - Initialisation de la file

```
FONCTION initFileStat( file : pointeur sur FileStat) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    SI (???) ALORS
        result ← ???
    FIN SI
    tete(file) ← ???
    queue(file) ← ???
    retourner result
FIN
```

10	15	87	6	8	65	13	87	55	87
----	----	----	---	---	----	----	----	----	----

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler
  - Initialisation de la file

```
FONCTION initFileStat( file : pointeur sur FileStat) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    SI (file EST EGAL A NULL) ALORS
        result ← ???
    FIN SI
    tete(file) ← ???
    queue(file) ← ???
    retourner result
FIN
```

10	15	87	6	8	65	13	87	55	87
----	----	----	---	---	----	----	----	----	----

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler
  - Initialisation de la file

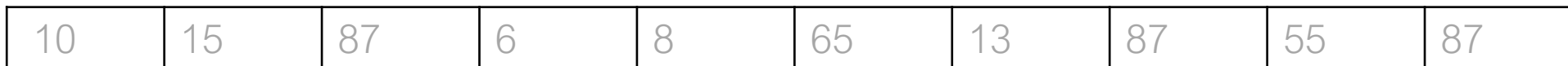
```
FONCTION initFileStat( file : pointeur sur FileStat) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    SI (file EST EGAL A NULL) ALORS
        result ← -1
    FIN SI
    tete(file) ← ???
    queue(file) ← ???
    retourner result
FIN
```

10	15	87	6	8	65	13	87	55	87
----	----	----	---	---	----	----	----	----	----

# Files statiques

- Les opérations de la file statique :
  - Enfiler
  - Défiler
  - Initialisation de la file

```
FONCTION initFileStat( file : pointeur sur FileStat) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    SI (file EST EGAL A NULL) ALORS
        result ← -1
    FIN SI
    tete(file) ← 0
    queue(file) ← -1
    retourner result
FIN
```

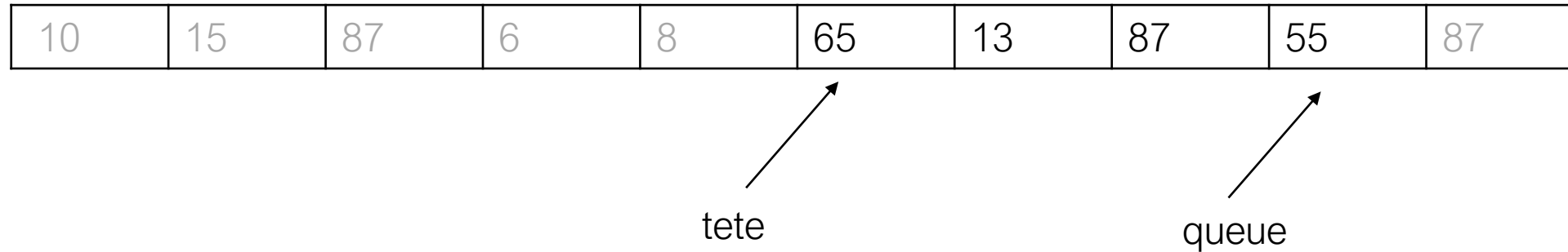


↑  
queue

↑  
tete

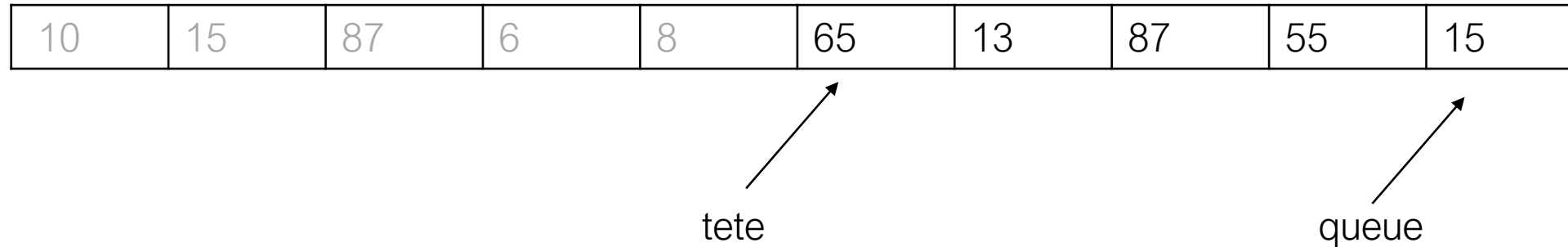
# Files statique : limites

- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Exemple :



# Files statiques : limites

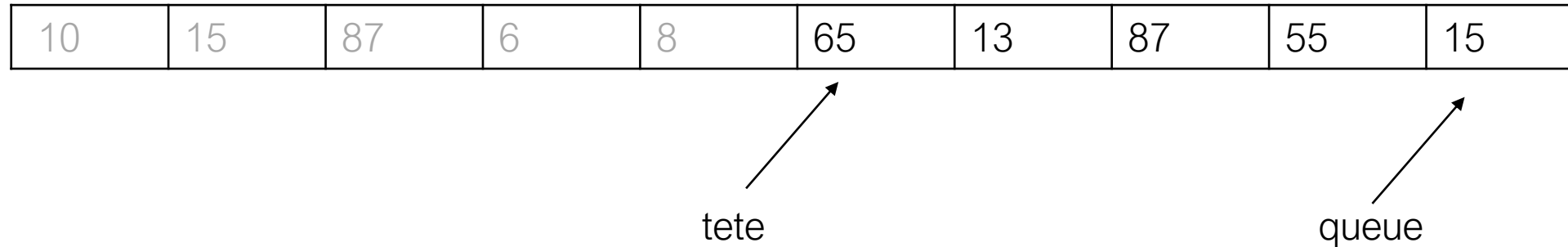
- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Exemple :



On ne peut pas rajouter de nouvel élément!

# Files statiques : limites

- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Exemple :



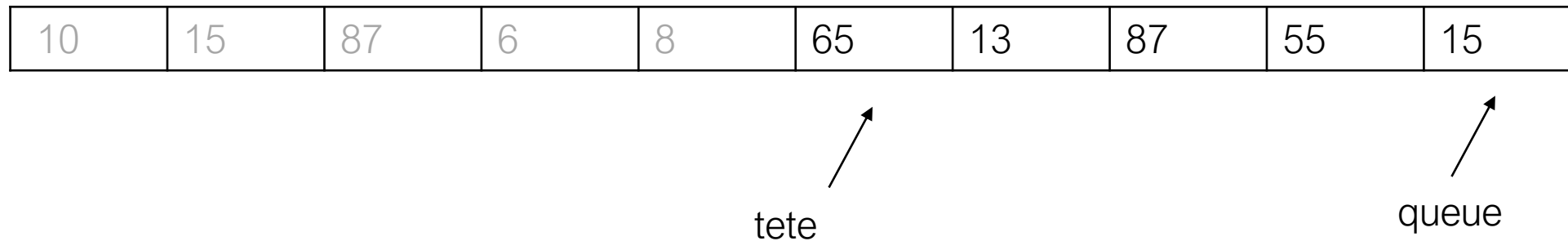
On ne peut pas rajouter de nouvel élément!

- Les indices de tête et de queue se décale en fin de tableau avec les utilisations de la pile : on arrive toujours à ce résultat au bout d'un certain nombre d'opérations!



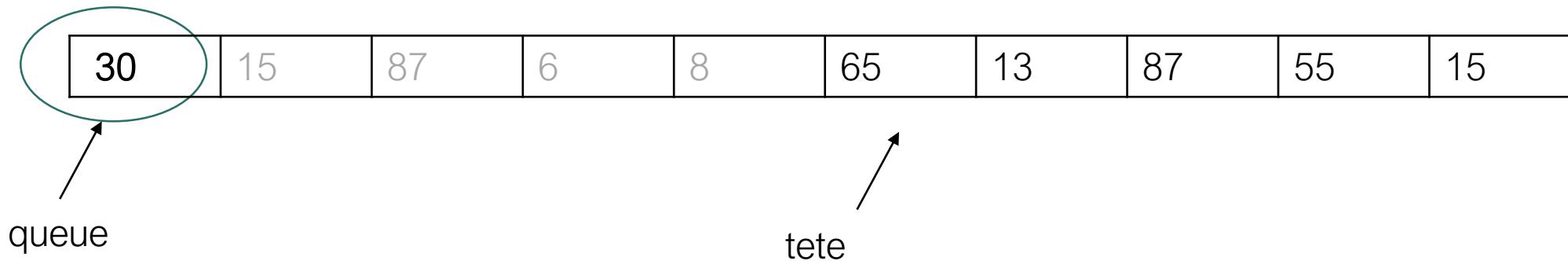
# Files statiques cycliques

- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Solution : utilisation d'une **file cyclique**.
- Principe : les éléments de la file vont « *reboucler* » dans le tableau.



# Files statiques cycliques

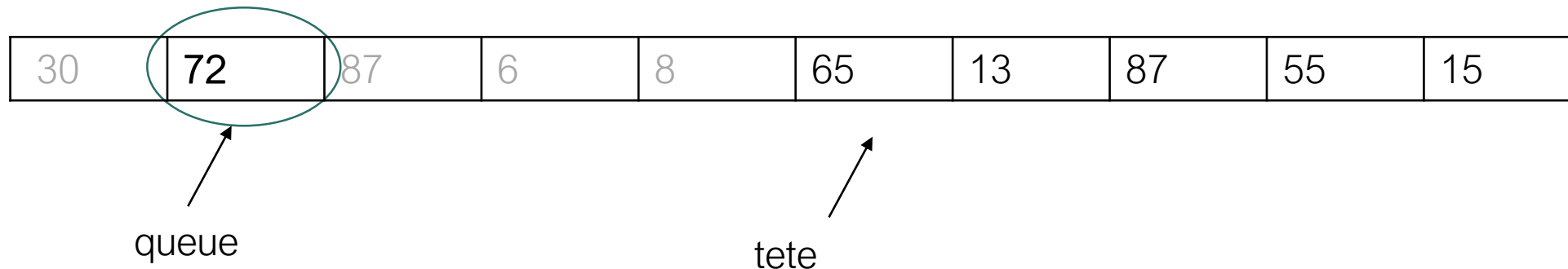
- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Solution : utilisation d'une **file cyclique**.
- Principe : les éléments de la file vont « *reboucler* » dans le tableau.



L'élément supplémentaire est rajouté au début du tableau !

# Files statiques cycliques

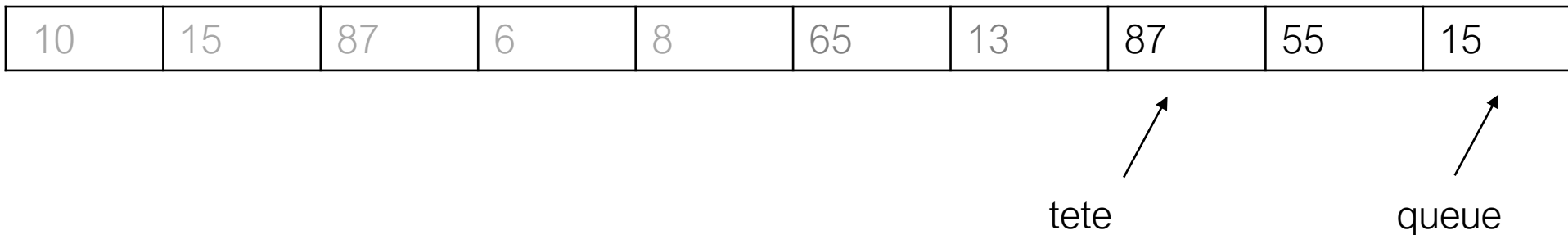
- Les opérations telles que celles définies précédemment peuvent mener à une impossibilité d'utiliser la file même lorsque celle-ci n'est pas pleine.
- Solution : utilisation d'une **file cyclique**.
- Principe : les éléments de la file vont « *reboucler* » dans le tableau.



L'élément supplémentaire est rajouté au début du tableau !

# Files statiques cycliques

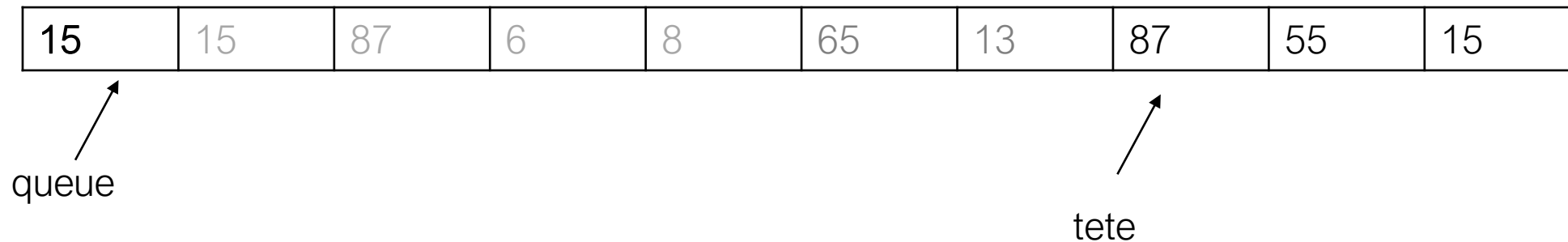
- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :



tete = 7  
queue = 9

# Files statiques cycliques

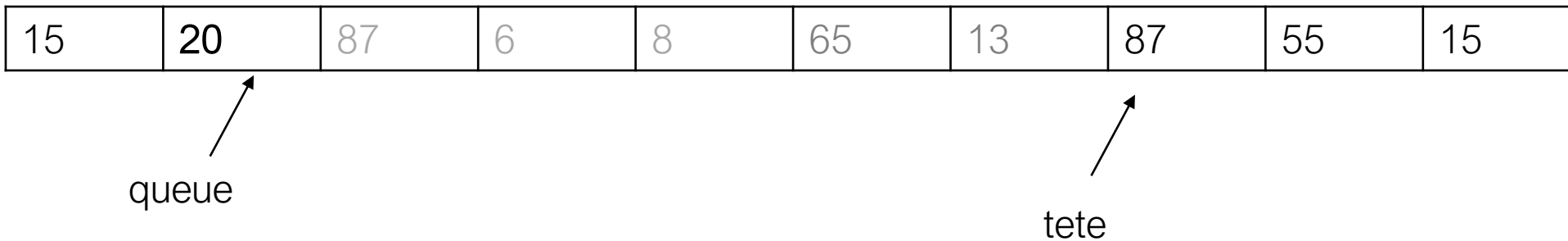
- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :



tete = 7  
queue = 10

# Files statiques cycliques

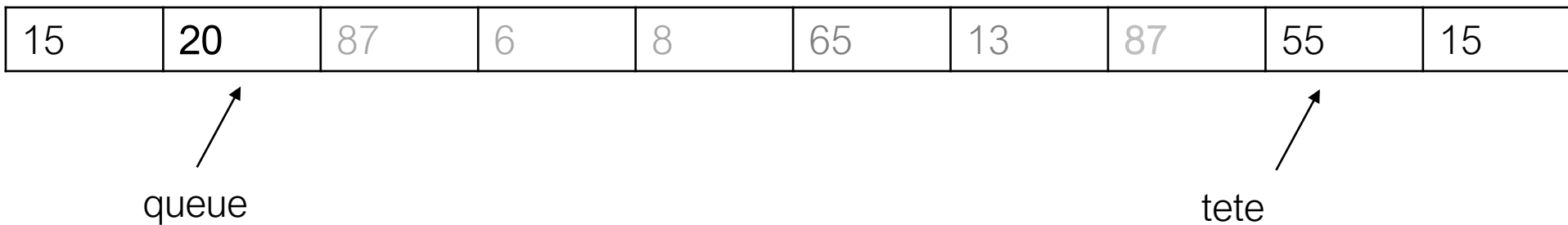
- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :



tete = 7  
queue = 11

# Files statiques cycliques

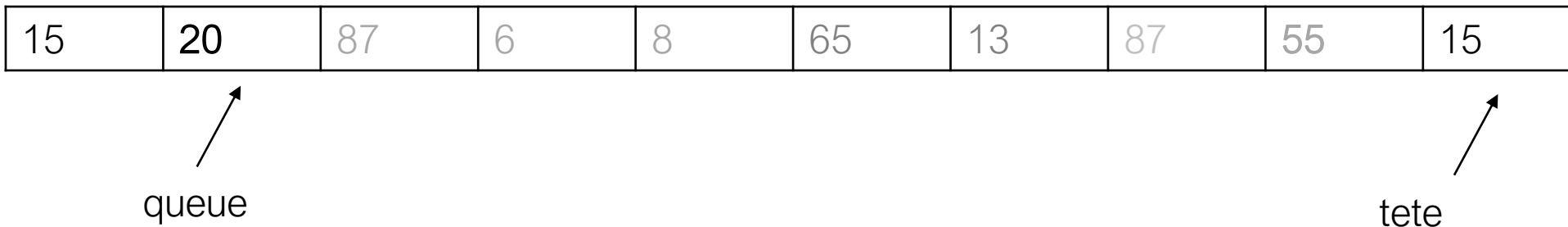
- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :



tete = 8  
queue = 11

# Files statiques cycliques

- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :

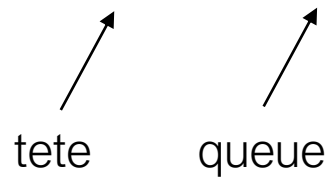


tete = 9  
queue = 11



# Files statiques cycliques

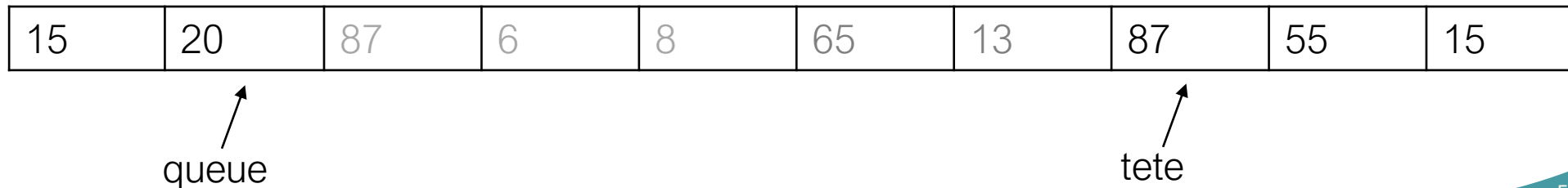
- Dans la file cyclique, les entiers **tete** et **queue** ne vont pas indiquer directement les indices des tableaux.
- **tete** et **queue** indiquent les positions du haut et du bas de la file s'il n'y avait pas de rebouclage (comme si le tableau avait une taille infinie)
- Exemple :



tete = 10  
queue = 11

# Files statiques cycliques

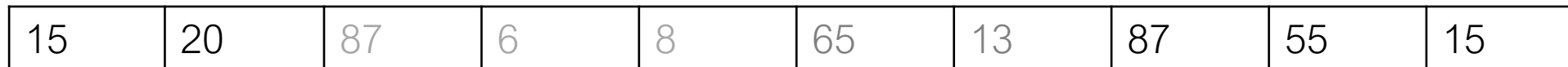
- Opérations dans une file cyclique :
  - Enfilage



# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DDEBUT
    result ← ???
    SI(???) ALORS // file pleine
        result ← ???
    SINON
        ??? // décalage queue
        ??? // stockage élément
    FIN SI
    RETOURNER result
FIN
```



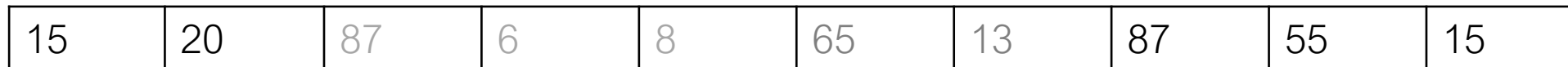
↑  
queue

↑  
tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DDEBUT
    result ← vérification du pointeur de file et contenu de la structure
    SI(???) ALORS // file pleine
        result ← ???
    SINON
        ??? // décalage queue
        ??? // stockage élément
    FIN SI
    RETOURNER result
FIN
```



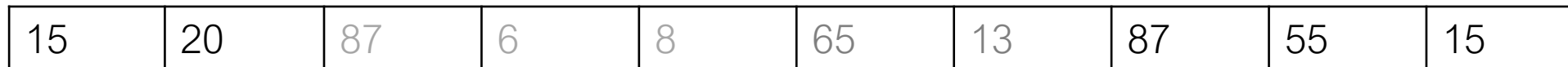
↑  
queue

↑  
tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DDEBUT
    result ← vérification du pointeur de file et contenu de la structure
    SI(queue(file)-tete(file) EGAL A taille(file)-1) ALORS // file pleine
        result ← ???
    SINON
        ??? // décalage queue
        ??? // stockage élément
    FIN SI
    RETOURNER result
FIN
```



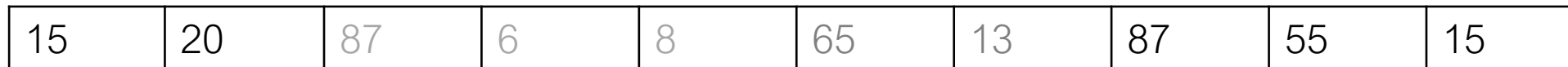
↑  
queue

↑  
tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DDEBUT
    result ← vérification du pointeur de file et contenu de la structure
    SI(queue(file)-tete(file) EGAL A taille(file)-1) ALORS // file pleine
        result ← 1
    SINON
        ??? // décalage queue
        ??? // stockage élément
    FIN SI
    RETOURNER result
FIN
```



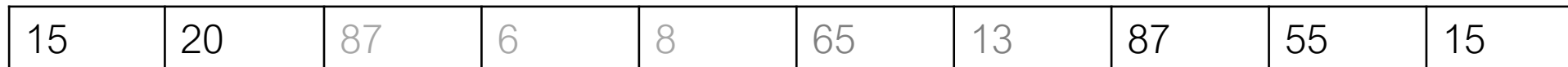
↑  
queue

↑  
tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DDEBUT
    result ← vérification du pointeur de file et contenu de la structure
    SI(queue(file)-tete(file) EGAL A taille(file)-1) ALORS // file pleine
        result ← 1
    SINON
        queue(file) ← queue(file) + 1 // décalage queue
        ??? // stockage élément
    FIN SI
    RETOURNER result
FIN
```



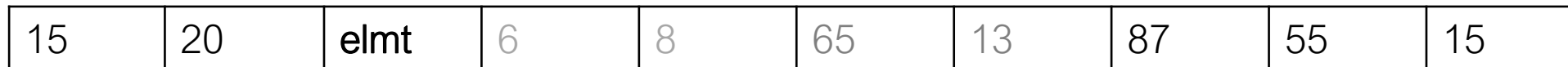
↑  
queue

↑  
tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Enfilage

```
FONCTION enfilerStatCyclique(file : pointeur sur FileStat, elmt : Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification du pointeur de file et contenu de la structure
    SI(queue(file)-tete(file) EGAL A taille(file)-1) ALORS // file pleine
        result ← 1
    SINON
        queue(file) ← queue(file) + 1 // décalage queue
        tabFile(file)[queue(file) MOD taille(file)] ← elmt // stockage élément
    FIN SI
    RETOURNER result
FIN
```



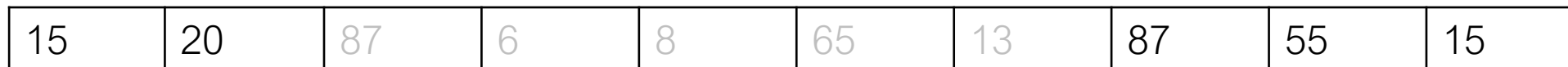
↑  
queue

↑  
tete



# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage



queue

tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage

```
FONCTION defilerStatCyclique(file : ptr sur FileStat, pelmt : ptr sur Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    ???
    SI(???)
        SI(???) ALORS                                     // file vide
            result ← 1
        SINON
            ???                                           // récupération
            ???                                           // décalage
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

15	20	87	6	8	65	13	87	55	15
----	----	----	---	---	----	----	----	----	----

queue

tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage

```
FONCTION defilerStatCyclique(file : ptr sur FileStat, pelmt : ptr sur Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification des pointeurs et contenus des structures
    SI(result EGAL A 0)
        SI(???) ALORS // file vide
            result ← 1
        SINON
            ??? // récupération
            ??? // décalage
    FIN SI
FIN SI
RETOURNER result
FIN
```

15	20	87	6	8	65	13	87	55	15
----	----	----	---	---	----	----	----	----	----

queue

tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage

```
FONCTION defilerStatCyclique(file : ptr sur FileStat, pelmt : ptr sur Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification des pointeurs et contenus des structures
    SI(result EGAL A 0)
        SI(queue(file)-tete(file) EGAL A -1) ALORS // file vide
            result ← 1
        SINON
            ??? // récupération
            ??? // décalage
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

15	20	87	6	8	65	13	87	55	15
----	----	----	---	---	----	----	----	----	----

queue

tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage

```
FONCTION defilerStatCyclique(file : ptr sur FileStat, pelmt : ptr sur Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification des pointeurs et contenus des structures
    SI(result EGAL A 0)
        SI(queue(file)-tete(file) EGAL A -1) ALORS // file vide
            result ← 1
        SINON
            *pelmt ← tabFile(file)[tete(file) MOD taille(file)] // récupération
            ??? // décalage
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

15	20	87	6	8	65	13	87	55	15
----	----	----	---	---	----	----	----	----	----

queue

tete

# Files statiques cycliques

- Opérations dans une file cyclique :
  - Défilage

```
FONCTION defilerStatCyclique(file : ptr sur FileStat, pelmt : ptr sur Element) : entier
VARIABLE
    result ← 0 : entier
DEBUT
    result ← vérification des pointeurs et contenus des structures
    SI(result EGAL A 0)
        SI(queue(file)-tete(file) EGAL A -1) ALORS // file vide
            result ← 1
        SINON
            *pelmt ← tabFile(file)[tete(file) MOD taille(file)] // récupération
            tete(file) ← tete(file) + 1 // décalage
        FIN SI
    FIN SI
    RETOURNER result
FIN
```

15	20	87	6	8	65	13	87	55	15
----	----	----	---	---	----	----	----	----	----

queue

tete

# Résumé

- Les **pires** et les **files** sont des objets algorithmiques permettant de stocker des données en répondant à des critères d'insertion et de suppression précis.
- Pile : dernier entré, premier sorti (last IN, first OUT → LIFO).
- File : premier entré, premier sorti (First In, First Out → FIFO).
- Il existe plusieurs manières d'implémenter les piles et les files selon les résultats souhaités (statique, dynamique, cyclique).
- Les piles et les files sont très utilisées par l'ordinateur pour son fonctionnement, mais elles sont également régulièrement présentes dans certains algorithmes classiques (voir cours suivant !).