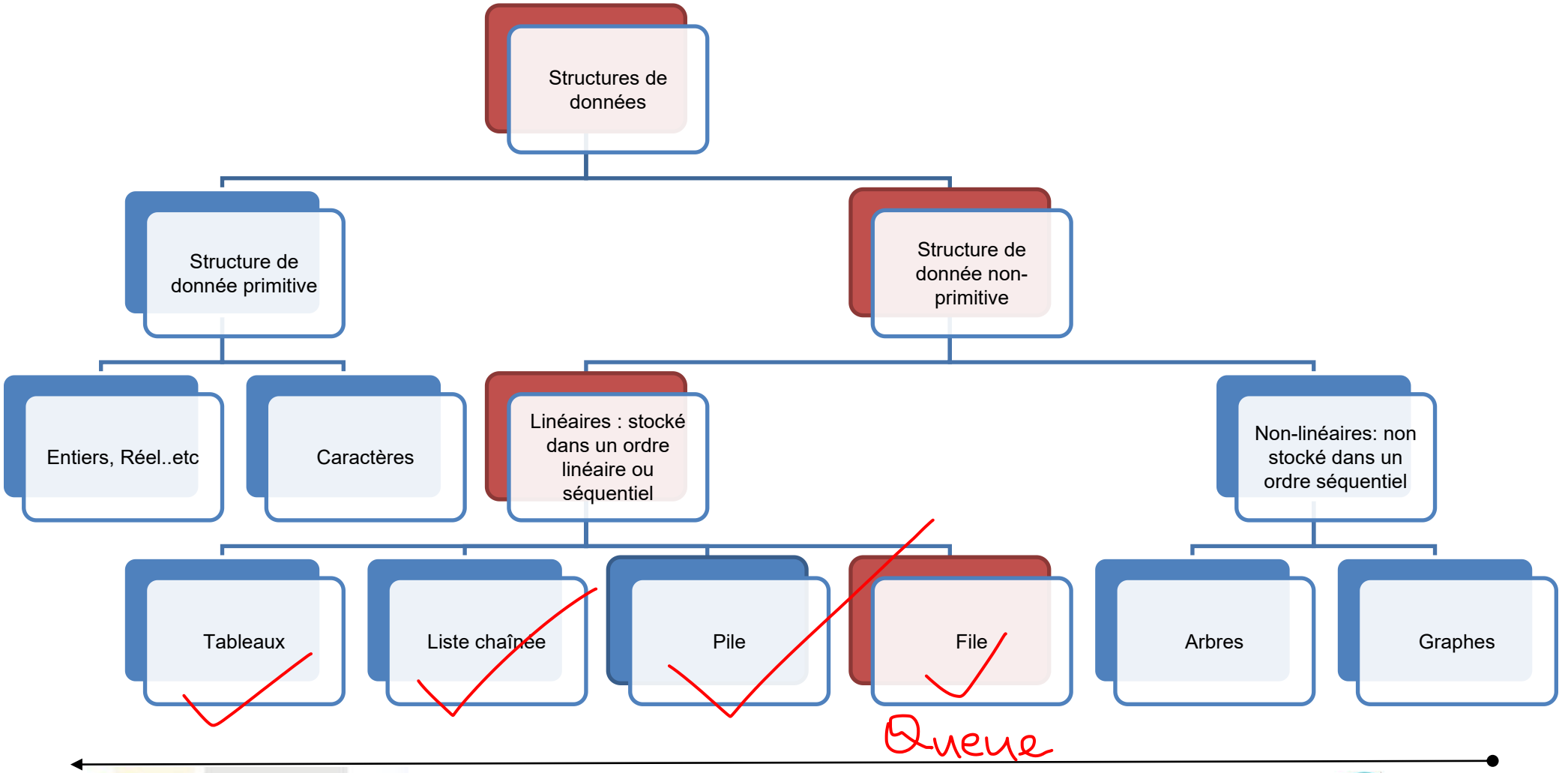
Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

CM 9 : Les Files (1)

CLASSIFICATION DES STRUCTURES DE DONNÉES

- **Structures de données primitives et non primitives**



PLAN

- Introduction
- Définition
- Applications
- Les Files
  - Représentation contiguë
  - Représentation chaînée
- Les services (Opérations) des Files
    - Création
    - Ajout
    - Suppression
    - Consultation/recherche

Introduction (1)

Introduction (2)

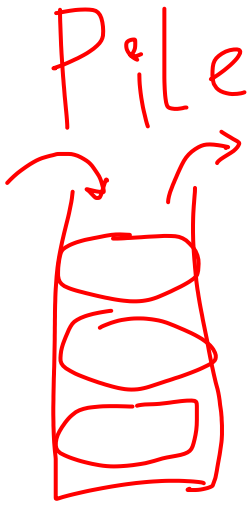
Introduction (3)

- La file est une structure permettant, comme un tableau ou une pile, des éléments ayant tous le même type.
- Comme dans une pile, l'ordre dans lequel les éléments d'une file sont accessibles dépend de l'ordre dans lequel ils ont été ajouté.
- La spécificité d'une file est que l'élément accessible est toujours le *plus ancien*.
- C'est le principe d'une file d'attente à un guichet, ou la personne servie, celle qui se trouve en *tête* de file, est celle qui attend depuis le plus longtemps dans la file, tandis qu'un nouvel arrivant sera placé en *queue* de file.

DÉFINITION (1)

File (Queue en anglais)

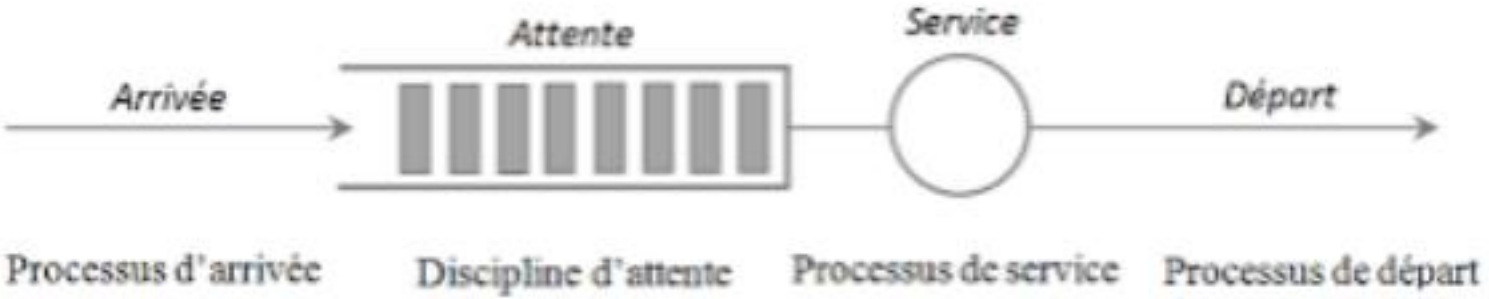
- Un espace où on peut accumuler des éléments
- On ne peut retirer que le plus vieux des éléments ajouté.
- Exemple: Distributeur de boissons gazeuses



DÉFINITION (2)

- **QU'EST-CE QU'UNE FILE ?**
  - Une file est une structure de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du premier arrivé premier sorti
    - ou encore FIFO (First In First Out).

Pile LIFO (Last In First Out)



APPLICATIONS

De nombreuse applications s'appuient sur l'utilisation d'une file, on peut citer:

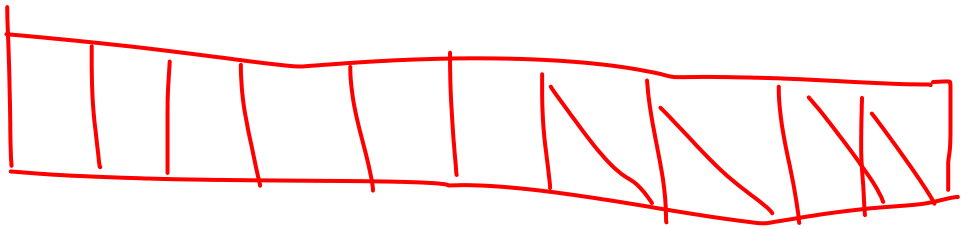
- Les files sont largement utilisées comme listes d'attente pour une seule ressource partagée comme une imprimante, un disque, un processeur.
- Les files d'attente sont utilisées pour transférer des données entre deux processus
- Les files d'attente sont utilisées comme tampons sur les lecteurs MP3 et les lecteurs de CD portables, les listes de lecture iPod.
- Les files d'attente sont utilisées dans le système d'exploitation pour gérer les interruptions. Si les interruptions doivent être traitées dans l'ordre d'arrivée, une file d'attente FIFO est la structure de données appropriée.

Représentation d'une File

- **Les primitives de gestion des Files**
  - Initialiser : cette fonction crée une file vide.
  - EstVide : renvoie 1 si la file est vide, 0 sinon.
  - EstPleine : renvoie 1 si la file est pleine, 0 sinon.
  - AccederTete : cette fonction permet l'accès à l'information contenue dans la tête de file.
  - Enfiler : cette fonction permet d'ajouter un élément à la queue de la file.
    - La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.
  - Defiler : cette fonction supprime le début de la file. L'élément supprimé est retourné par la fonction Defiler pour pouvoir être utilisé.
  - Vider : cette fonction vide la file.
  - Detruire : cette fonction permet de détruire la file.

Représentation d'une File

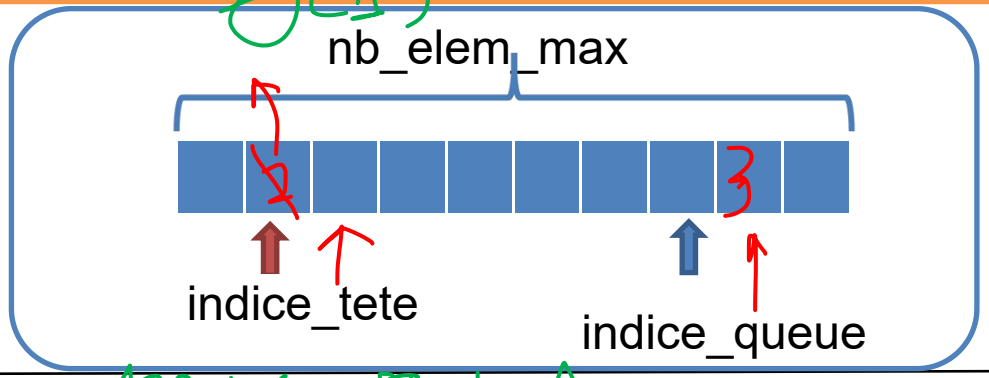
- **Représentation contiguë (par tableau)**
  - Les éléments de la file sont rangés dans un tableau
  - Deux indices sont nécessaires pour indiquer respectivement la tête et la queue de la file



IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Gestion naïve**
  - Pour implémenter une file sous forme de tableau, on crée la structure de données suivante.

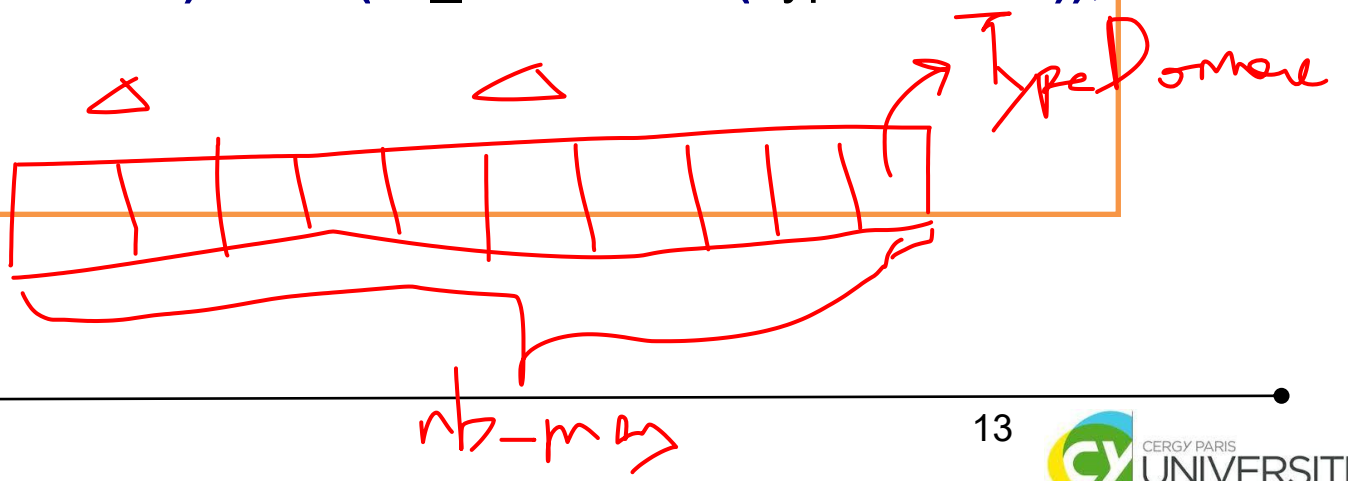
```
typedef float TypeDonnee;
typedef struct
{
  int nb_elem_max; /* nombre d'éléments du tableau */
  int indice_tete, indice_queue; /* indice de tête et queue */
  TypeDonnee *tab; /* tableau des éléments */
} File;
```



IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Créer une file vide.**
  - La fonction permettant de créer une file vide est la suivante :

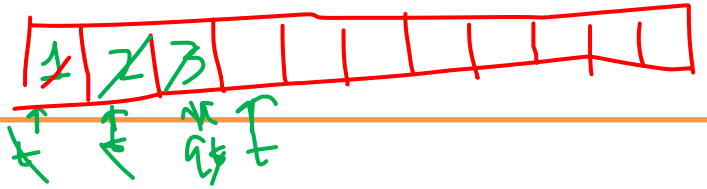
```
File Initialiser(int nb_max)
{
  File filevide;
  filevide.indice_tete=0; /* la file est vide */
  filevide.indice_queue=-1;
  filevide.nb_elem_max = nb_max; /* capacité nb_max */
  /* allocation des éléments : */
  filevide.tab = (TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
  return filevide;
}
```



IMPLÉMENTATION SOUS FORME DE TABLEAU

- **File vide,**
  - La fonction permettant de savoir si la file est vide est la suivante. La fonction renvoie
    - 1 si le nombre d'éléments est égal à 0.
    - La fonction renvoie 0 dans le cas contraire.

```
int EstVide(File F)
{
  /* renvoie 1 si l'indice de queue est plus petit */
  return (F.indice_tete == F.indice_queue+1) ?
1 : 0;
}
```



ind_queue = -1 ??

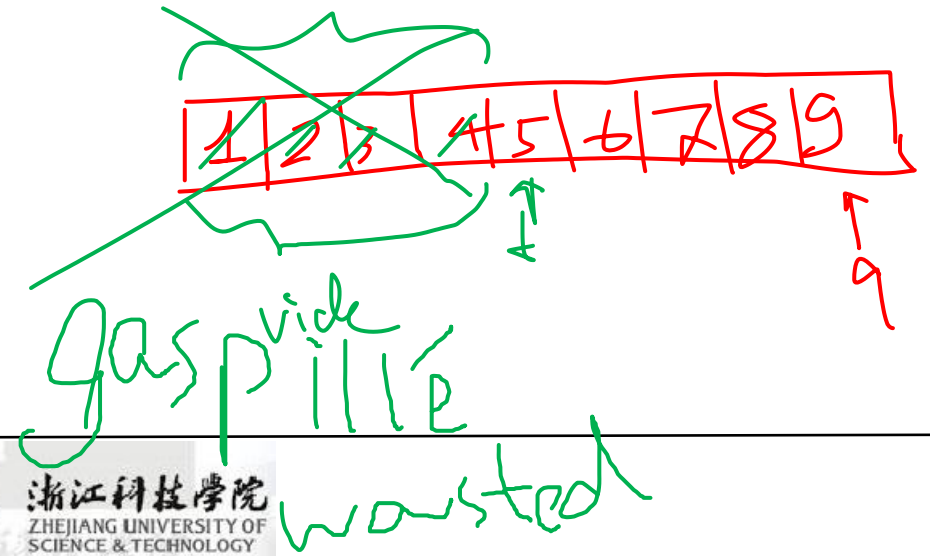
q=2 ,

t=3

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **File pleine**
  - La fonction permettant de savoir si la file est pleine est la suivante :

```
int EstPleine(File F)
{
  /* la file est pleine quand la queue est au bout du tableau */
  return (F.indice_queue == F.nb_elem_max-1) ? 1 : 0;
}
```



IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Accéder à la tête de la file**
  - La fonction effectue un passage par adresse pour ressortir le tête de la file
    - La tête de la file est le premier élément entré, qui est l'élément du tableau avec l'indice ***indice_tete***
  - La fonction permet d'accéder à la tête de la file et renvoie le code d'erreur 1 en cas de liste vide et 0 sinon

```
int AccederTete(File F, TypeDonnee *pelem)
{
  if (EstVide(F))
    return 1; /* on retourne un code d'erreur */
  *pelem = F.tab[F.indice_tete]; /* on renvoie l'élément */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

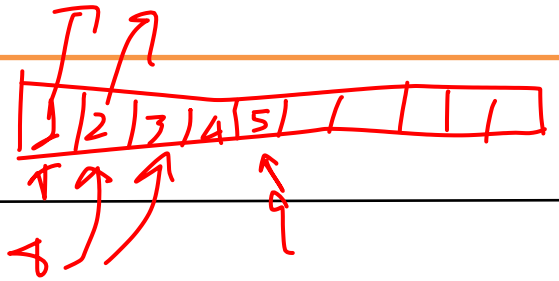
- **Ajouter un élément**
  - Pour modifier le nombre d'éléments de la file, il faut passer la file par adresse. La fonction Enfiler, qui renvoie 1 en cas d'erreur et 0 dans le cas contraire, est la suivante :

```
int Enfiler(File *pF, TypeDonnee elem)
{
  if (pF->indice_queue >= pF->nb_elem_max-1)
    return 1; /* erreur : file pleine */
  pF->indice_queue++; /* on insère un élément en queue */
  pF->tab[pF->indice_queue] = elem; /* nouvel élément */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

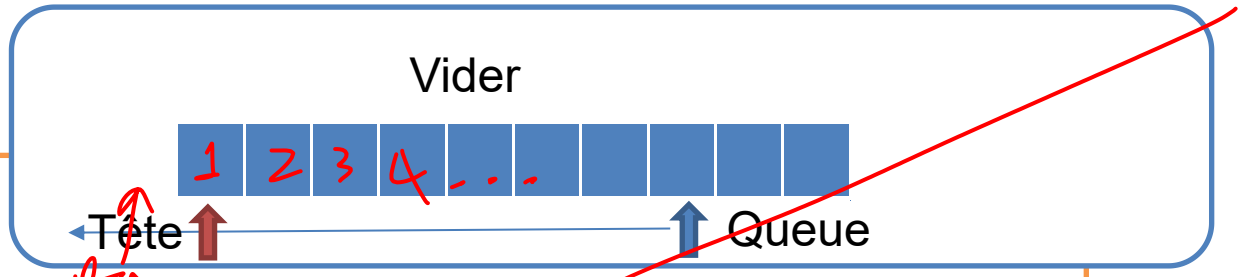
- **Supprimer un élément**
  - La fonction Defiler supprime la tête de la file en cas de file non vide. La fonction renvoie 1 en cas d'erreur (file vide), et 0 en cas de succès.

```
int Defiler(File *pF, TypeDonnee *pelem)
{
  if (EstVide(*pF))
    return 1; /* erreur : file vide */
  *pelem = pF->tab[pF->indice_tete]; /* on renvoie la tête */
  pF->indice_tete++; /* supprime l'élément de tête */
  return 0;
}
```

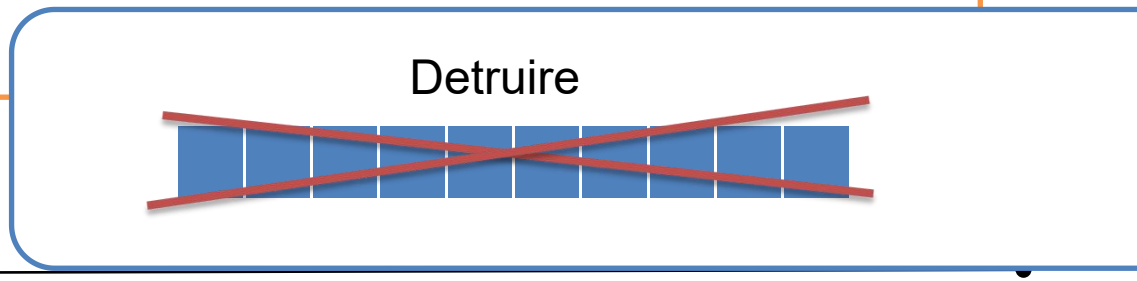


IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Vider et détruire**

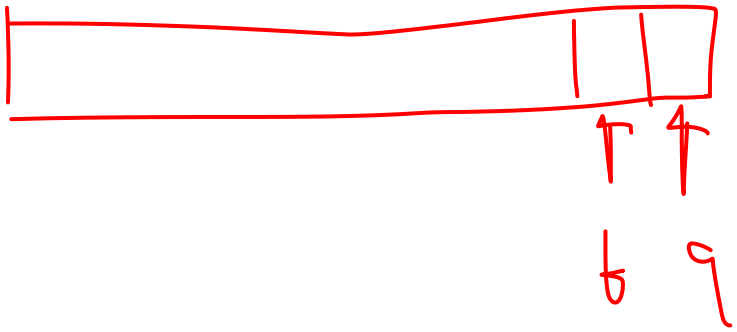



```
void Vider(File *pF)
{
  pF->indice_tete = 0; /* réinitialisation des indices */
  pF->indice_queue = -1;
}
void Detruire(File *pF)
{
  if (pF->nb_elem_max != 0)
    free(pF->tab); /* libération de mémoire */
  pF->nb_elem_max = 0; /* file de taille 0 */
}
```



IMPLÉMENTATION SOUS FORME DE TABLEAU

- Le problème dans cette gestion des files par tableaux est qu'à mesure qu'on insère et qu'on supprime des éléments, les indice de tête et de queue ne font qu'augmenter.
- L'utilisation d'une telle file est donc limitée dans le temps, et nb_elem_max est le nombre total d'éléments qui pourront jamais être insérés dans la file.



QUIZ

Une pile *Pile* représentée par un liste chaînée est déclarée comme la suit :

```
typedef float TypeDonnee;
typedef struct Cell
{
   TypeDonnee donnee;
   struct Cell *suivant; /* pointeur sur la cellule suivante */
} TypeCellule;
typedef TypeCellule* Pile;
```

5 minute

Est-il possible de retrouver le nombre d'éléments de la pile?

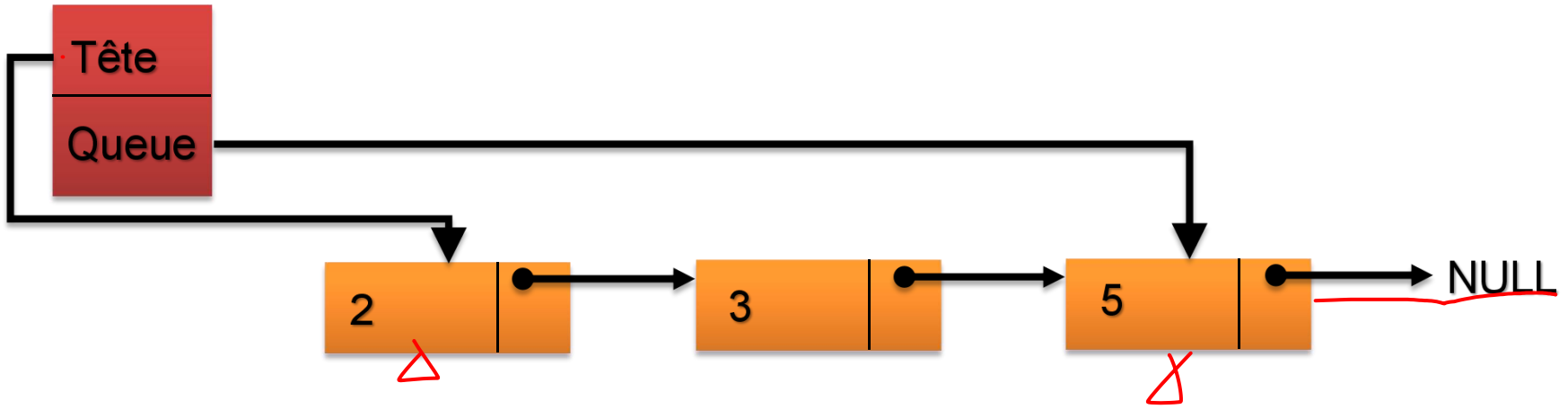
Choix:

A- Oui

B- Non

Représentation d'une File

- **Représentation chaînée (par pointeurs → liste chaînée)**
  - Les éléments de la file sont chaînés entre eux
  - Un pointeur vers le premier maillon de la liste
  - Un pointeur vers le *dernier* maillon
  - La liste chaînée sera donc orientée de la tête vers la queue



IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Types**
  - Pour implémenter une file sous forme de liste chaînée, on crée la structure de données suivante.

```
typedef float TypeDonnee;
typedef struct Cell /* déclaration de la liste chaînée */
{
  TypeDonnee donnee;
  struct Cell *suivant;
} TypeCellule;
typedef struct
{
  TypeCellule *tete, *queue;      /* pointeurs sur la première et dernière
cellule */
} File;
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

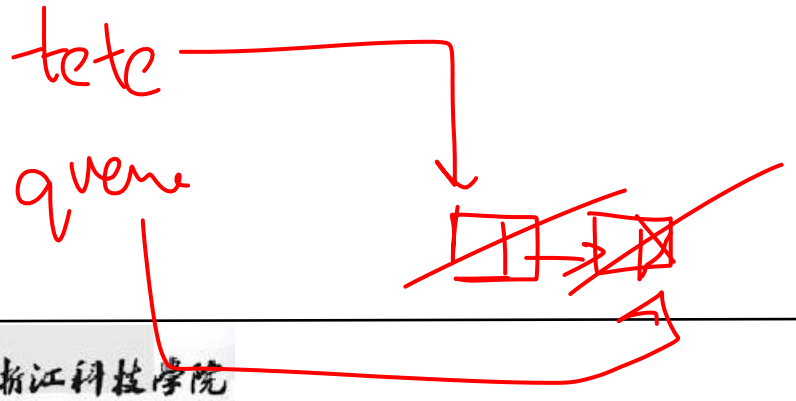
- **Créer une file vide**
  - La fonction permettant de créer une file vide est la suivante :

```
File Initialiser()
{
  File filevide;
  filevide.tete=NULL; /* liste vide : NULL */
  return filevide;
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **File vide**
  - La fonction permettant de savoir si la file est vide est la suivante. La fonction renvoie 1 si la file est vide, et renvoie 0 dans le cas contraire.

```
int EstVide(File F)
{
  return (F.tete == NULL) ? 1 : 0;
}
```



IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

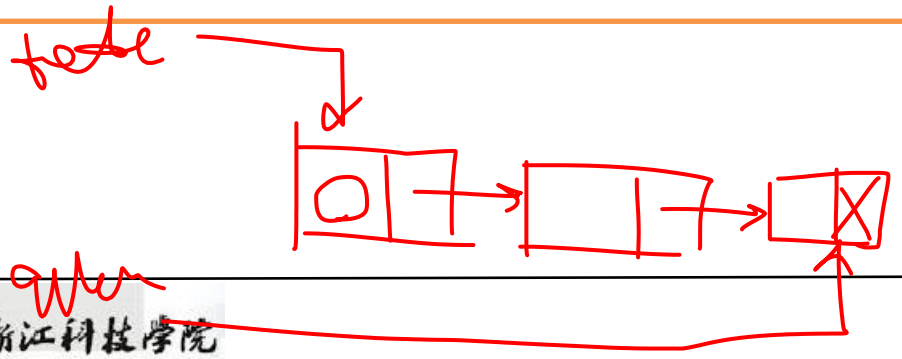
- **File pleine**
  - La fonction permettant de savoir si la file est pleine est la suivante :

```
int EstPleine(File F)
{
  return 0; /* une liste chaînée n'est jamais pleine */
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Accéder à la tête de la file**
  - La tête de la file est le premier élément entré qui est la tête de liste. La fonction renvoie 1 en cas de file vide, 0 sinon:

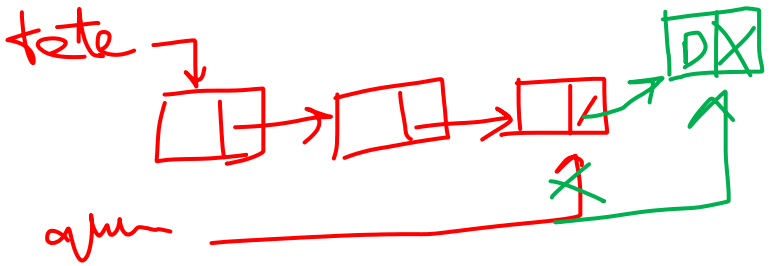
```
int AccederTete(File F, TypeDonnee *pelem)
{
  if (EstVide(F))
    return 1; /* on retourne un code d'erreur */
  *pelem = F.tete->donnee; /* on renvoie la donnée de tête */
  return 0;
}
```



IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Ajouter un élément**
  - La fonction d'ajout d'un élément **Enfiler** est une fonction d'insertion en queue de liste

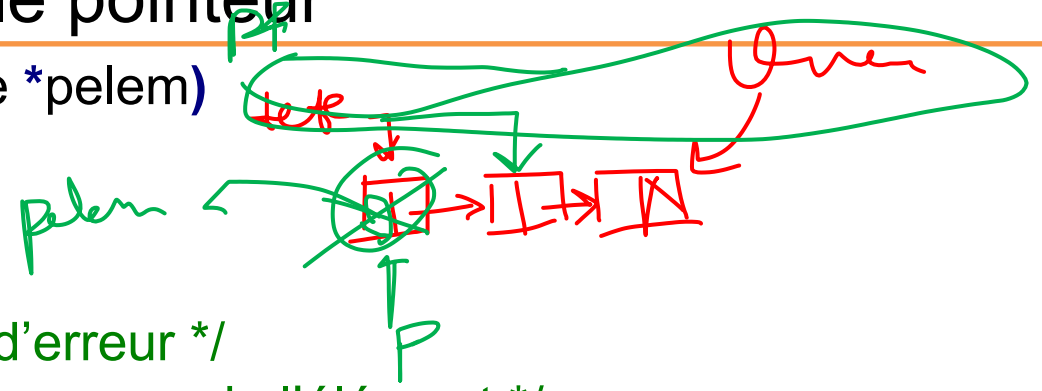
```
void Enfiler(File *pF, TypeDonnee elem)
{
  TypeCellule *q;
  q = (TypeCellule*)malloc(sizeof(TypeCellule)); /* allocation */
  q->donnee = elem;
  q->suivant = NULL; /* suivant de la dernière cellule NULL */
  if (pF->tete == NULL) /* si file vide */
    {
      pF->queue = pF->tete = q;
    }
  else
  {
    pF->queue->suivant = q; /* insertion en queue de file */
    pF->queue = q;
  }
}
```



IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Supprimer un élément**
  - La fonction **Defiler** supprime la tête de liste en cas de file non vide. La fonction renvoie 1 en cas d'erreur, et 0 en cas de succès. La file est passée par adresse, on a donc un double pointeur

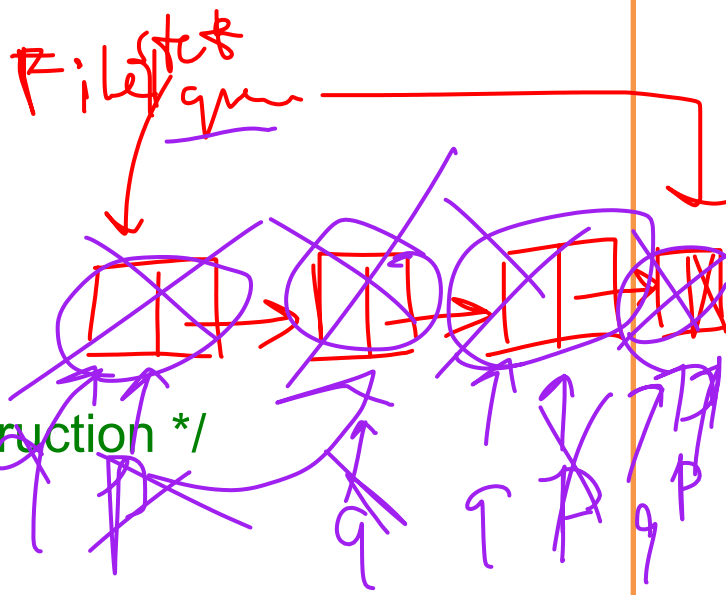
```
int Defiler(File *pF, TypeDonnee *pelem)
{
  TypeCellule *p;
  if (EstVide(*pF))
    return 1; /* retour d'un code d'erreur */
  *pelem = pF->tete->donnee; /* on renvoie l'élément */
  p = pF->tete; /* mémorisation de la tête de liste */
  pF->tete = pF->tete->suivant; /* passage au suivant */
  free(p); /* destruction de l'ancienne tête de liste */
  return 0;
}
```



IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Vider et détruire**
  - La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Detruire(File *pF)
{
  TypeCellule *p, *q;
  p=pF->tete; /* initialisation pour parcours de liste */
  while (p!= NULL)
    {
      q = p; /* mémorisation de l'adresse */
      p = p->suivant; /* passage au suivant avant destruction */
      free(q); /* destruction de la cellule */
    }
  pF->tete = NULL; /* on met la liste à vide */
}
```



pF->tete = pF->queue = NULL

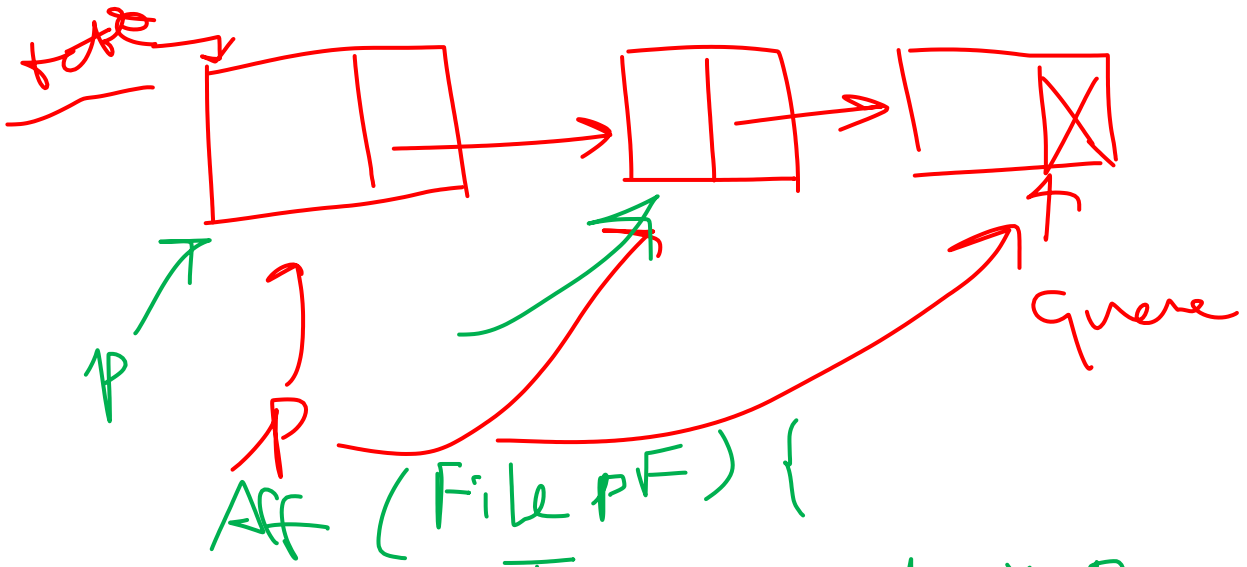
IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Vider et détruire**
  - La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Vider(File *pF)
{
  Detruire(pF); /* destruction de la liste */
  pF->tete = NULL; /* liste vide */
}
```

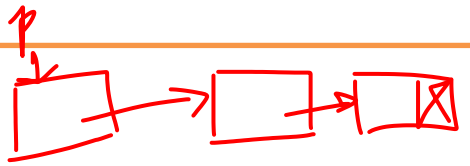
EXERCICES

- Écrire la fonction qui affiche les éléments d'une file d'attente représentée avec une liste chaînée



EXERCICES

- Écrire la fonction qui affiche les éléments d'une file d'attente représentée avec une liste chaînée



```
void Affichage(File * pF)
{
  TypeCellule *p;
  p=pF->tete; /* initialisation pour parcours de liste */
  while (p!= NULL)
    {
      printf("%f ", p->donnee); /* affichage de l'élément */
      p = p->suivant; /* passage au suivant */
    }
}
```

COMPARAISON ENTRE TABLEAUX ET LISTES CHAÎNÉES

- Dans les deux types de gestion des files, chaque primitive ne prend que quelques opérations (complexité en temps constant).

- Par contre, la gestion par listes chaînées, présente l'énorme avantage que la file a une capacité virtuellement illimitée (limitée seulement par la capacité de la mémoire centrale), la mémoire étant allouée à mesure des besoins.

- Au contraire, dans la gestion par tableaux, la mémoire est allouée au départ avec une capacité fixée.