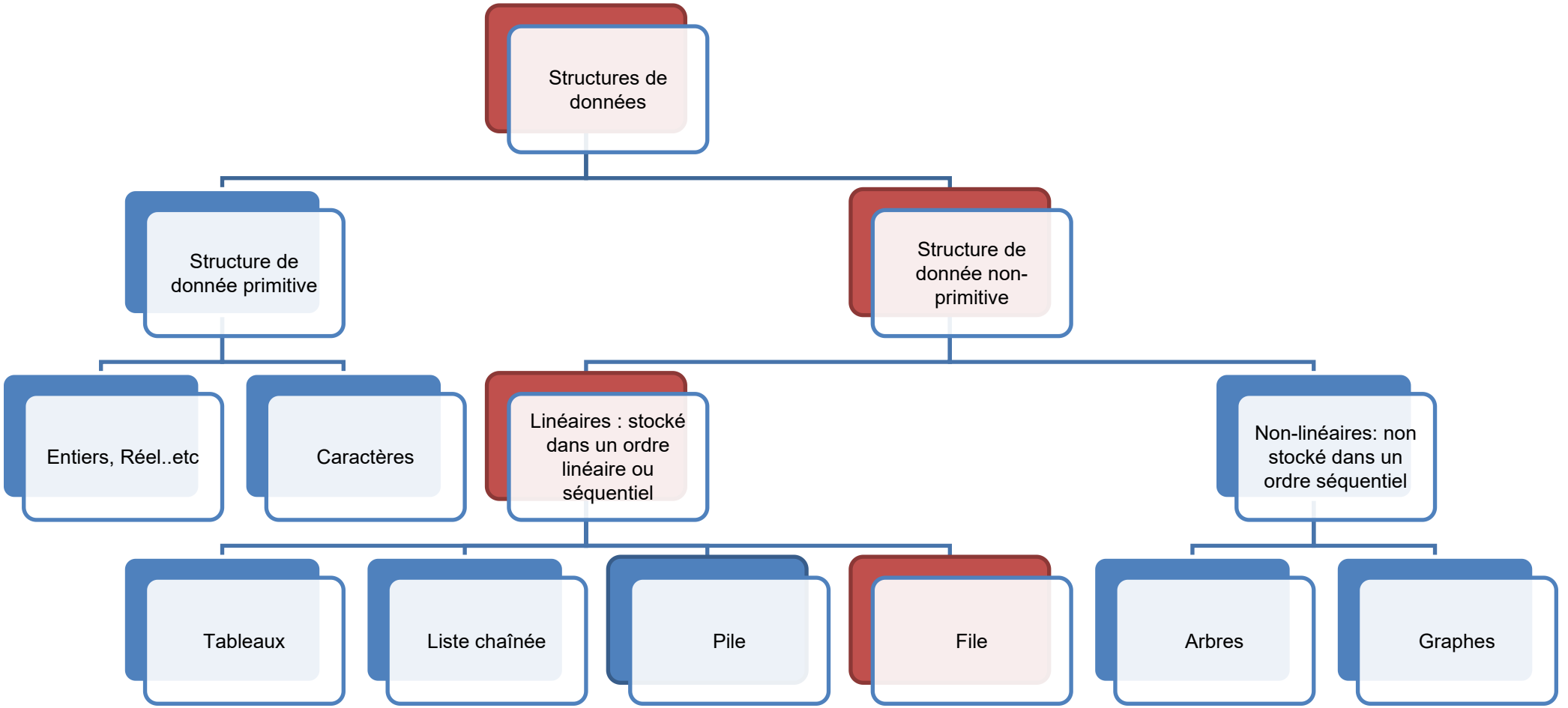
Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

CM 11 : Les Files (3)

CLASSIFICATION DES STRUCTURES DE DONNÉES

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



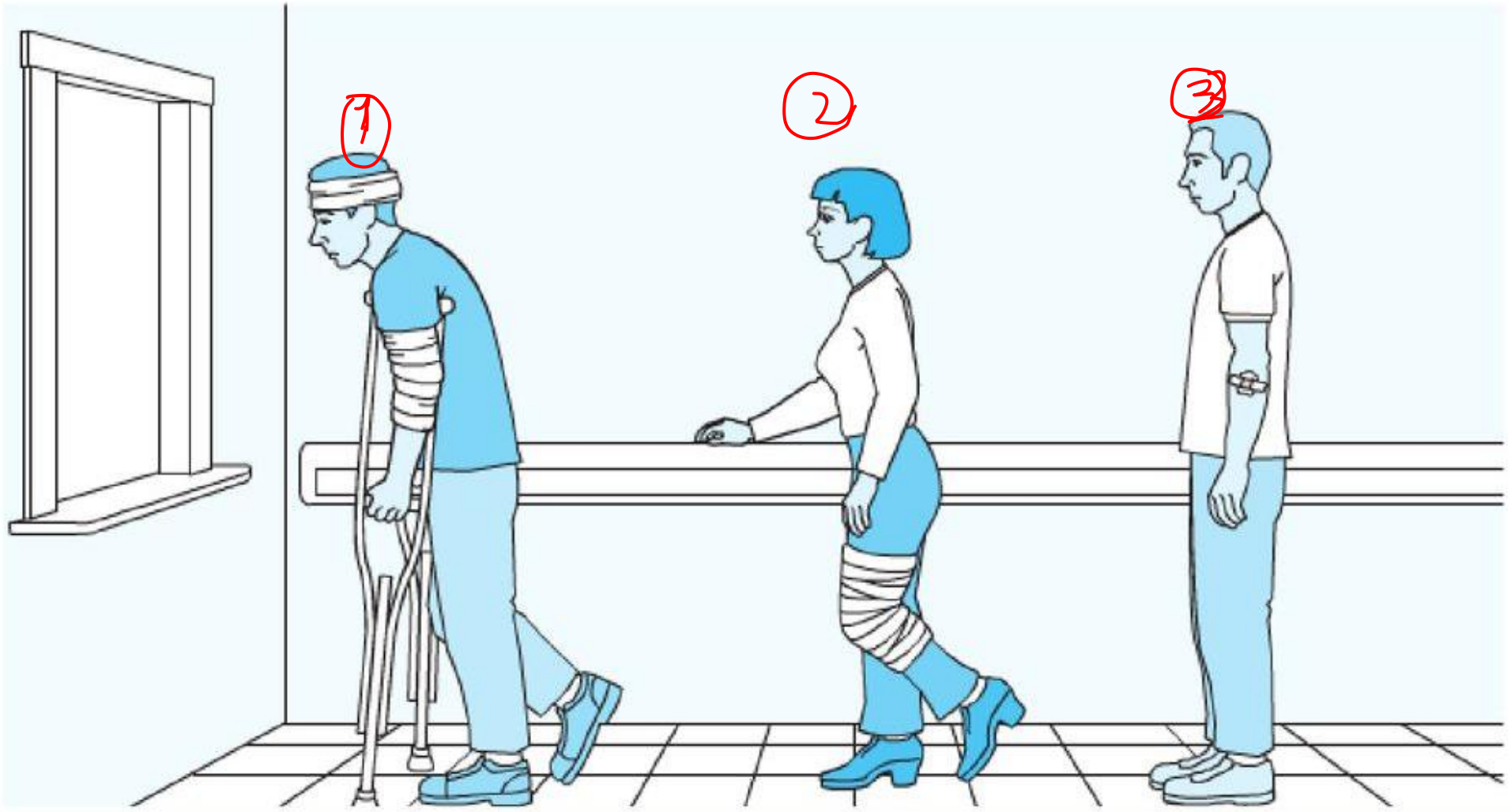
PLAN

- Les Types de Files
  - File de priorité
    - Définition et Exemples d'utilisation
    - Représentation par listes chainées
    - Représentation contigüe
  - Files multiples
    - Définition et Exemple
    - Primitives

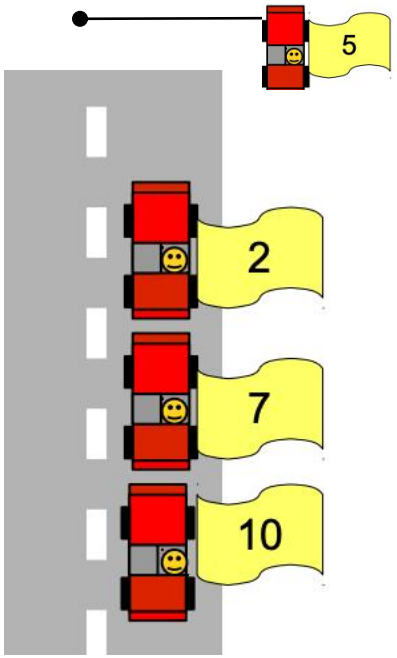
PLAN

- Les Types de Files
  - File de priorité
    - Définition et Exemples d'utilisation
    - Représentation par listes chainées
    - Représentation contigüe
  - Files multiples
    - Définition et Exemple
    - Primitives

Context



FILE DE PRIORITÉ (PIORITY QUEUE) : DÉFINITION



- Une file de priorité est une file particulière contenant des éléments ayant chacun une **priorité représentée** par un nombre entier positif.
- La priorité de l'élément sera utilisée pour déterminer l'ordre dans lequel les éléments seront traités.
- Les règles générales de traitement des éléments d'une file de priorité sont:
  - Un élément avec une priorité plus élevée est traité avant un élément avec une priorité inférieure.
  - Deux éléments avec la même priorité sont traités sur la base du premier arrivé, premier servi (FIFO).

FILE DE PRIORITÉ : EXEMPLES D'UTILISATION

- Ordonnancement des tâches d'un système d'exploitation.
- Serveur d'impression
- Contrôle aérien
- Tri par tas (à introduire S5)
- Etc…

Une file de priorité peut être considérée comme une file d'attente modifiée dans laquelle lorsqu'un élément doit être supprimé de la file d'attente, celui qui a la priorité la plus élevée est récupéré en premier.

Représentation d'une file de priorité:

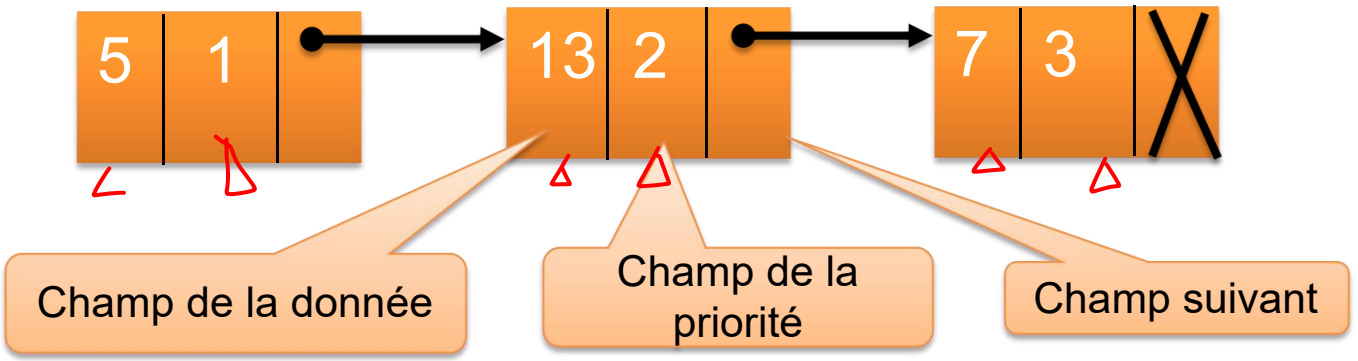
- Représentation par liste chainée
- Représentation contiguë (tableaux)
- Représentation par pile

REPRÉSENTATION PAR LISTE CHAINÉE

Lorsqu'une file de priorité est implémentée à l'aide d'une liste chaînée, chaque nœud de la liste comprendra trois parties:

- la partie informations ou données,
- le numéro de priorité de l'élément, et
- l'adresse du suivant élément.

```
struct node {
   int data;
   int priority;
   struct node * next;
}
```



REPRÉSENTATION PAR LISTE CHAINÉE

- Si nous utilisons une liste chaînée triée, l'élément avec la priorité la plus élevée précédera l'élément avec la priorité la plus basse.

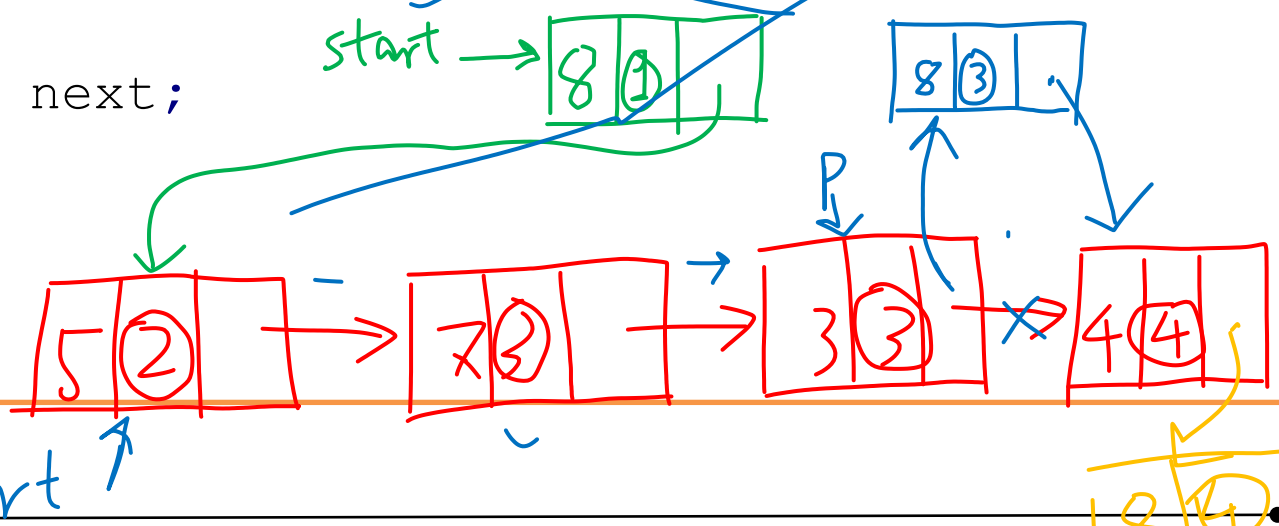
Un numéro de priorité plus faible signifie une priorité plus élevée. Par exemple, s'il y a deux éléments A et B, où A a un numéro de priorité 1 et B a un numéro de priorité 5, alors A sera traité avant B car il a une priorité plus élevée que B.

REPRÉSENTATION PAR LISTE CHAINÉE : PRIMITIVES

- **Ajout d'un élément : *insert* (push)**
  - Lorsqu'un nouvel élément doit être inséré dans une file de priorité, nous devons parcourir toute la liste jusqu'à ce que nous trouvions un nœud qui a une priorité inférieure à celle du nouvel élément. Le nouveau nœud est inséré avant le nœud avec la priorité la plus basse. Cependant, s'il existe un élément qui a la même priorité que le nouvel élément, le nouvel élément est inséré après cet élément.
- **Suppression d'un élément : *delete (pop)***
  - La suppression est un processus très simple dans ce cas. Le premier nœud de la liste sera supprimé et les données de ce nœud seront traitées en premier.

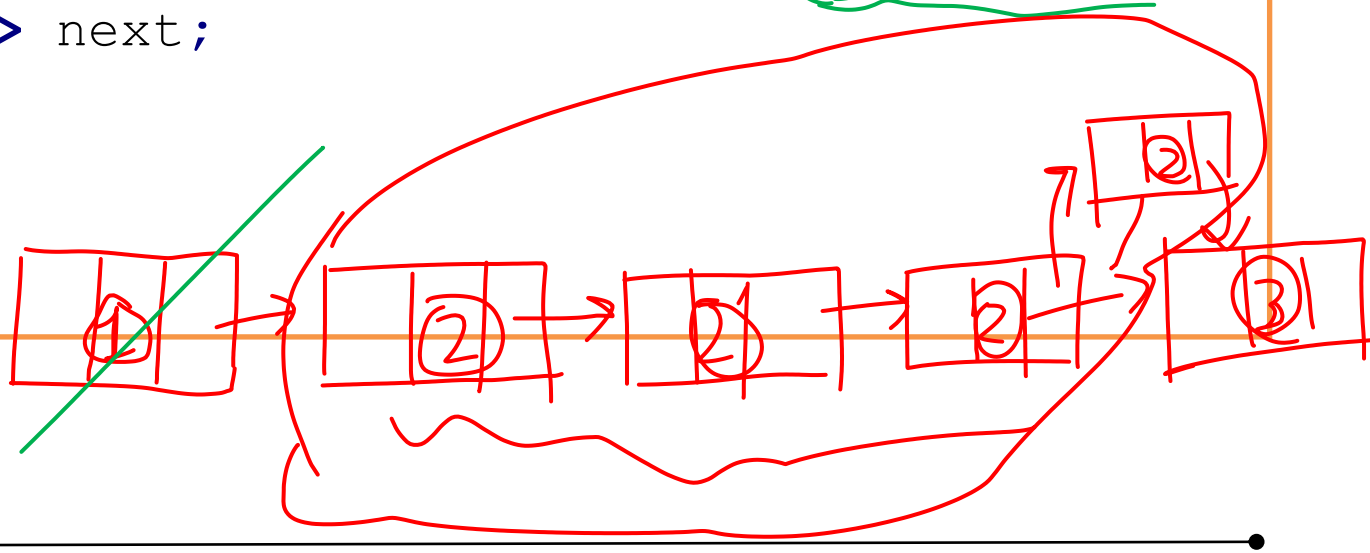
REPRÉSENTATION PAR LISTE CHAINÉE : AJOUT

```
struct node * insert(struct node * start, int val, int pri) {
  struct node * ptr, * p;
  ptr = (struct node * ) malloc(sizeof(struct node));
  ptr -> data = val;
  ptr -> priority = pri;
  if (start == NULL || pri < start -> priority) {
    ptr -> next = start;
    start = ptr;
  } else {
    p = start;
    while (p -> next != NULL && p -> next -> priority <= pri)
      p = p -> next;
    ptr -> next = p -> next;
    p -> next = ptr;
  }
  return start;
}
```



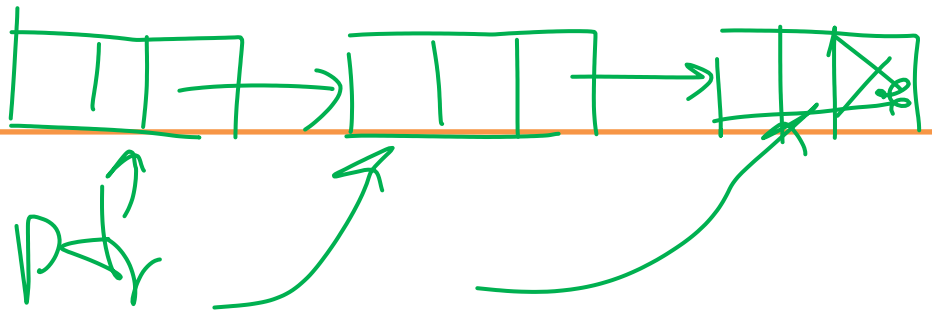
REPRÉSENTATION PAR LISTE CHAINÉE : SUPPRESSION

```
struct node * delete(struct node * start) {
  struct node * ptr;
  if (start == NULL) {
    printf("\n File de priorité vide");
    return;
  } else {
    ptr = start;
    printf("\n L'élément supprimé est: %d", ptr -> data);
    start = start -> next;
    free(ptr);
  }
  return start;
}
```



REPRÉSENTATION PAR LISTE CHAINÉE : AFFICHAGE

```
void display(struct node * start) {
  struct node * ptr;
  ptr = start;
  if (start == NULL)
    printf("\n File de priorité vide");
  else {
    printf("\n File de priorité est : ");
    while (ptr != NULL) {
      printf("\t%d[priorité=%d]", ptr -> data, ptr -> priority);
      ptr = ptr -> next;
    }
  }
}
```



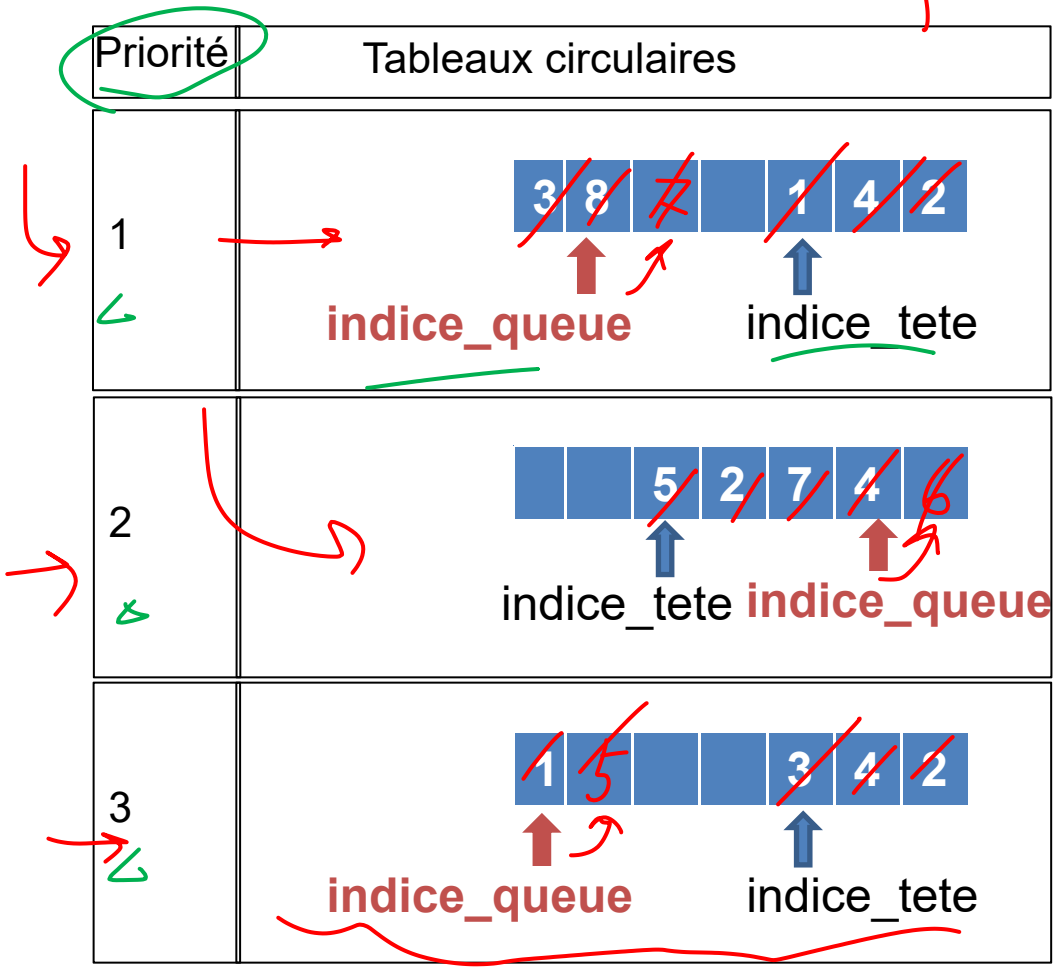
REPRÉSENTATION CONTIGUË (TABLEAUX)

Lorsque des tableaux sont utilisés pour implémenter une file de priorité, une file d'attente distincte pour chaque numéro de priorité est utilisée. Chacune de ces files d'attente sera implémentée à l'aide de tableaux circulaires ou de files d'attente circulaires. Chaque file d'attente individuelle aura ses propres pointeurs TETE et QUEUE.

Nous utilisons un tableau bidimensionnel (circulaire) à cette fin où chaque file d'attente se verra allouer la même quantité d'espace.

REPRÉSENTATION CONTIGUË (TABLEAUX)

Example de file de priorité avec représentation contigüe contenant 3 niveaux de priorité



REPRÉSENTATION CONTIGUË (TABLEAUX) : PRIMITIVES

- **Ajout d'un élément :**
  - Pour insérer un nouvel élément avec la priorité K dans la file de priorité, ajoutez l'élément à l'extrémité arrière de la ligne K, où K est le numéro de ligne ainsi que le numéro de priorité de cet élément.
- **Suppression d'un élément :**
  - Pour supprimer un élément, nous trouvons la première file d'attente non vide, puis traitons l'élément à la tête de la première file d'attente non vide.

QUIZ

Dans une file de priorité, si deux éléments ont la même priorité alors ils soront traités sur la base du premier arrivé, premier servi (FIFO).

A- Vrai
B- Faux

2 minute

QUIZ

Le nombre maximum d'éléments qu'une file circulaire peut contenir est :

A- La taille du tableau utilisé pour l'implémenter
B- Le nombre double de la taille du tableau utilisé pour l'implémenter
C- La moitié de la taille du tableau utilisé pour l'implémenter

5 minute

PLAN

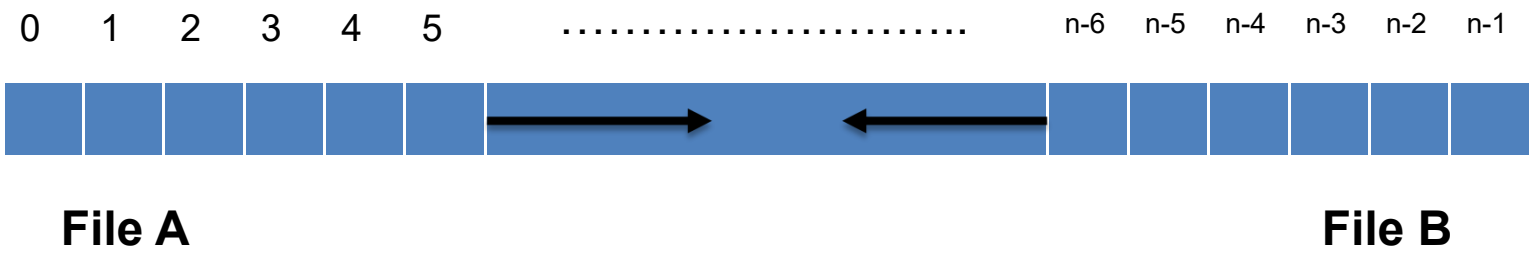
- Les Types de Files
  - File de priorité
    - Définition et Exemples d'utilisation
    - Représentation par listes chainées
    - Représentation contigüe
  - Files multiples
    - Définition et Exemple
    - Primitives

FILES MULTIPLES : DÉFINITION

- Lorsque nous implémentons une file d'attente **à l'aide d'un tableau**, la taille du tableau doit être connue à l'avance. Si moins d'espace est alloué à la file d'attente, des conditions de dépassement de capacité fréquentes seront rencontrées.
- Dans le cas où nous allouons une grande quantité d'espace pour la file d'attente, cela entraînera un gaspillage de mémoire.
- Ainsi, il existe un compromis entre la fréquence des débordements et l'espace alloué. Une meilleure solution pour résoudre ce problème est donc d'avoir **plusieurs files d'attente dans un même grand tableau**.

FILES MULTIPLES : EXAMPLE

- Dans la figure, un tableau *QUEUE[n]* est utilisé pour représenter deux files d'attente, la file A et la file B.
- La valeur de n est telle que la taille combinée des deux files ne dépassera jamais n.
- Lors de l'utilisation de ces files d'attente, il est important de noter une chose: la file d'attente A augmentera de gauche à droite, tandis que la file d'attente B augmentera de droite à gauche en même temps.



FILES MULTIPLES : EXAMPLE

- En étendant le concept à plusieurs files d'attente, une file d'attente peut également être utilisée pour représenter M nombre de files d'attente dans le même tableau.
- Alors chaque file d'attente se verra attribuer une quantité égale d'espace délimitée par des indices.

Files multiples contenant deux files A et B

- **Initialisation**

```
int QUEUE[MAX], rearA = -1, frontA = -1,
rearB = MAX, frontB = MAX;
```

- **Primitives :**
  - insertA
  - deleteA
  - insertB
  - deleteB

FILES MULTIPLES : PRIMITIVES

- **insertA**

```
void insertA(int val)
{
  if (rearA == rearB- 1)
    printf("\n File pleine");
  else
  {
    if (rearA == -1 && frontA == -1)
    {
      rearA = frontA = 0;
      QUEUE[rearA] = val;
    }
    else
      QUEUE[++rearA] = val;
  }
}
```

- **deleteA**

```
int deleteA()
{
  int val;
  if (frontA == -1)
  {
    printf("\n File A vide");
    return- 1;
  }
  else
  {
    val = QUEUE[frontA];
    frontA++;
    if (frontA > rearA)
      frontA = rearA = -1
    return val;
  }
}
```

- **display_queueA**

```
void display_queueA()
{
  int i;
  if (frontA == -1)
    printf("\n File est vide");
  else
  {
    for (i = frontA; i <= rearA; i++)
      printf("\t %d", QUEUE[i]);
  }
}
```

FILES MULTIPLES : PRIMITIVES

- **insertB**

```
void insertB(int val)
{
  if (rearA == rearB- 1)
    printf("\n File plein");
  else
  {
    if (rearB == MAX && frontB == MAX)
    {
      rearB = frontB = MAX- 1;
      QUEUE[rearB] = val;
    }
    else
      QUEUE[--rearB] = val;
  }
}
```

FILES MULTIPLES : PRIMITIVES

- **deleteB**

```
int deleteB()
{
  int val;
  if (frontB == MAX)
  {
    printf("\n File B est vide");
    return- 1;
  }
  else
  {
    val = QUEUE[frontB];
    frontB--;
    if (frontB < rearB)
      frontB = rearB = MAX;
    return val;
  }
}
```

- **display_queueB**

```
void display_queueB()
{
  int i;
  if (frontB == MAX)
    printf("\n File vide");
  else
  {
    for (i = frontB; i >= rearB; i--)
      printf("\t %d", QUEUE[i]);
  }
}
```