



**LICENCE INFORMATIQUE**

# **ALGORITHMES & STRUCTURES DE DONNÉES**

## **Listes chaînées**

**Présenté par BABACAR DIOP**

**Qu'est-ce  
qu'une liste  
chaînée ?**

Une liste chaînée est une collection de «nœuds» connectés en mémoire, chacun contenant «l'adresse» du nœud suivant.

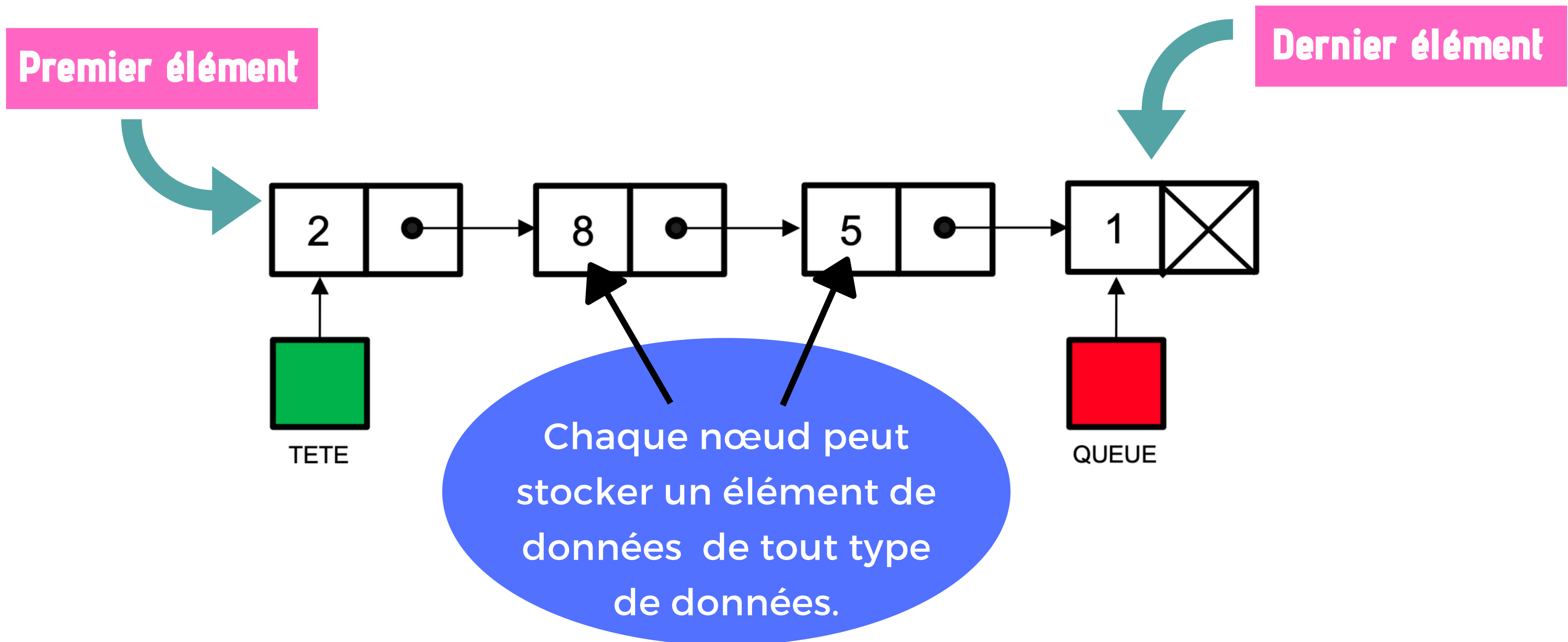
---

# Qu'est-ce qu'une liste chaînée ?

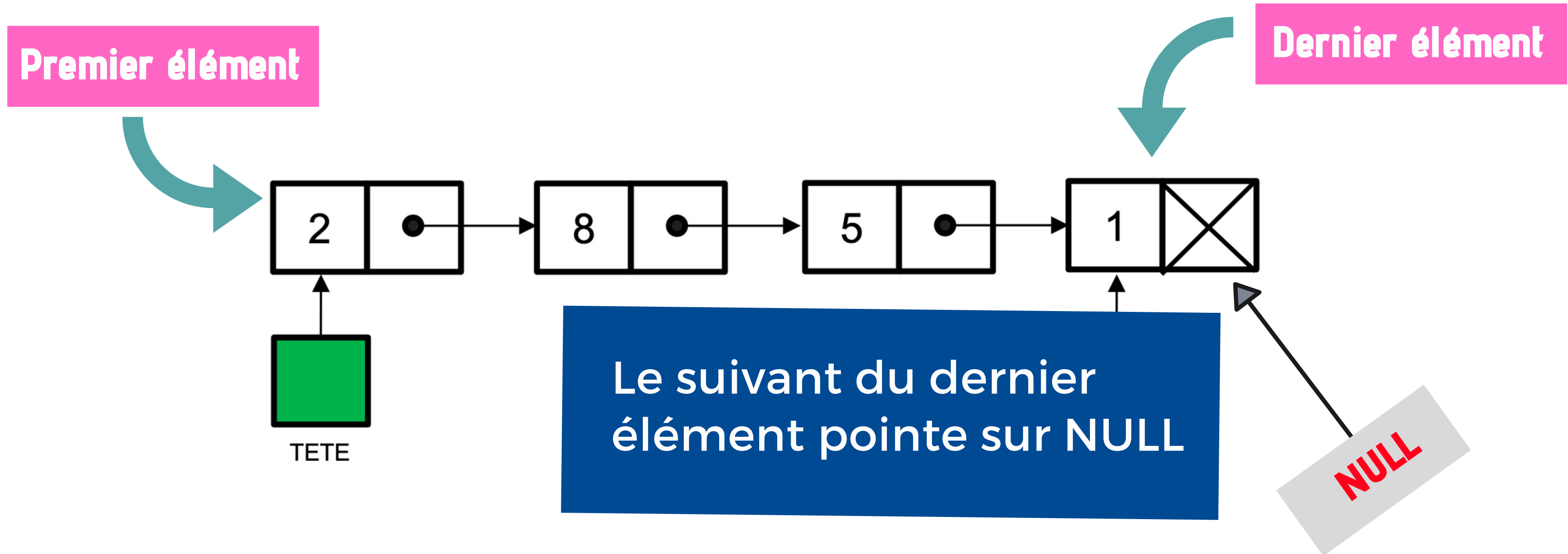
Une liste chaînée comporte  
une **tête** et une **queue**,  
représentant respectivement  
le **premier** et le **dernier**  
élément de la liste.

---

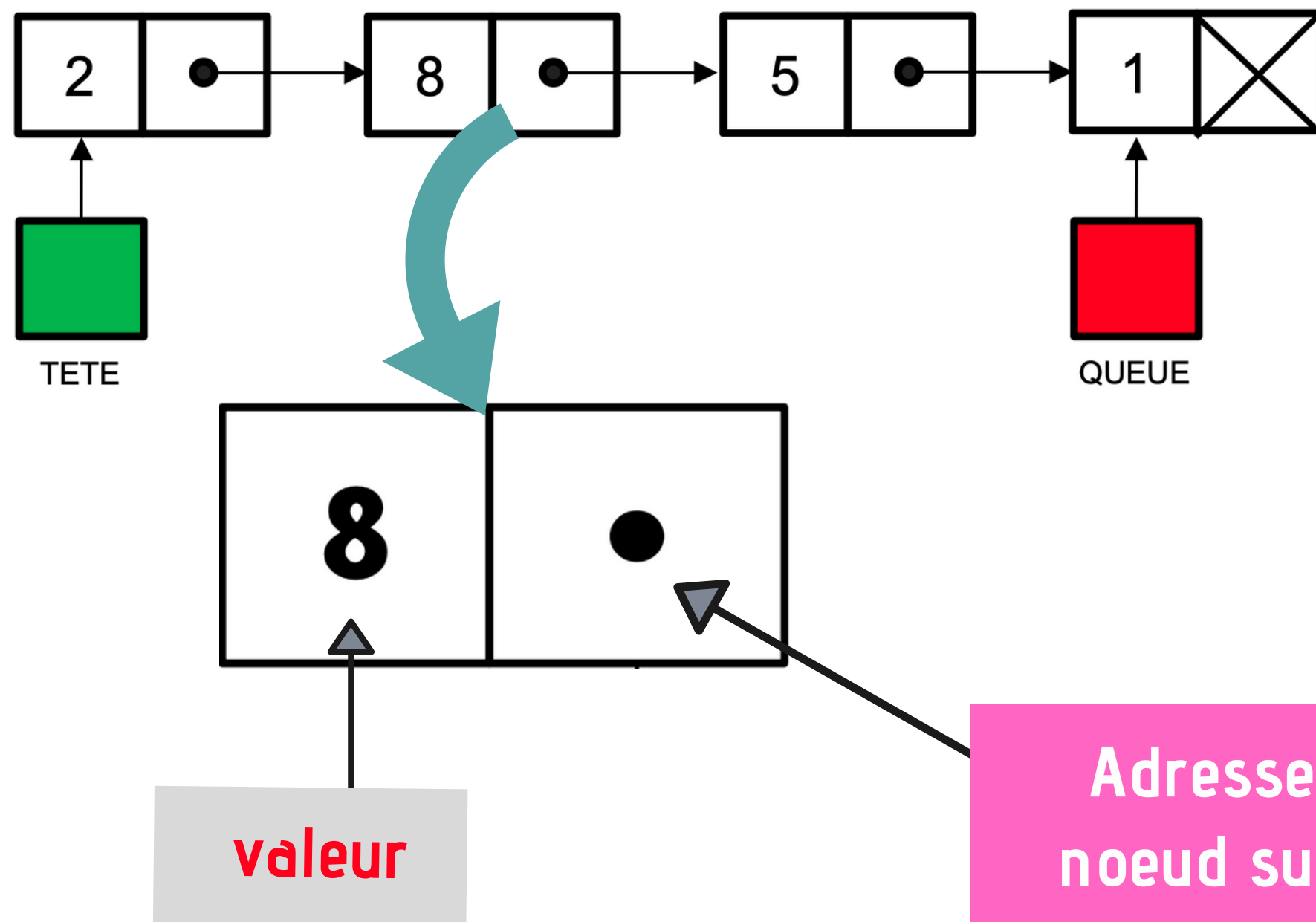
# Exemple



# Exemple



# Représentation



Chaque élément d'une liste chaînée est stocké sous la forme d'un nœud

# Comment définir une liste chaînée en C?

Chaque nœud de la liste  
contient deux champs:

- un champ de données,
- et l'adresse d'un autre  
nœud qui est le suivant.



# Comment définir une liste chaînée en C?

2 étapes:

- D'abord définir la structure des noeuds
  - Ensuite définir la structure de la liste
-



# Comment définir une liste chaînée en C?

## 1. Structure des noeuds

```
struct noeud {  
    int valeur;  
    struct noeud* suivant;  
}  
Typedef struct noeud Noeud;
```

---

**Comment  
définir une  
liste chaînée  
en C?**

## 2. Structure de la liste

```
struct Liste {  
    Noeud* premier;  
    int nbElements  
}
```

---

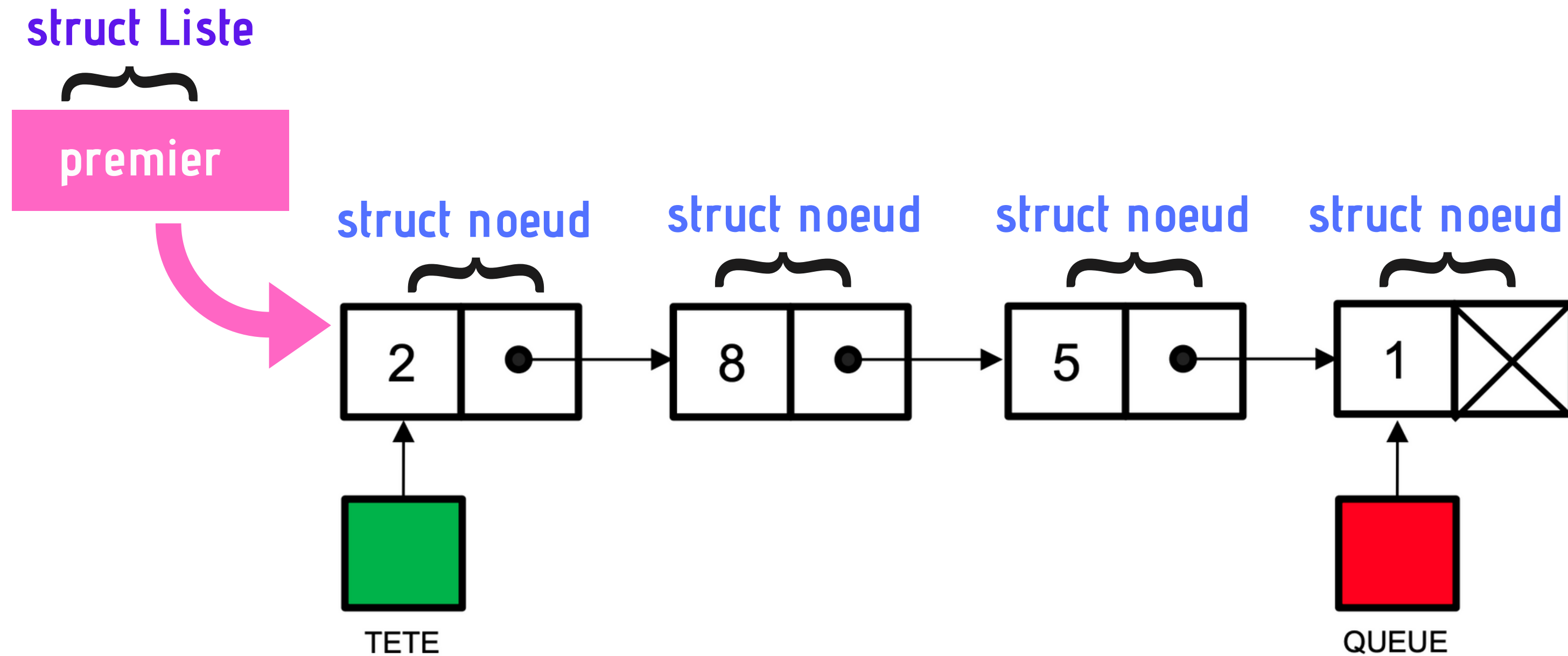
# Comment définir une liste chaînée en C?



La structure **Liste** contient un pointeur vers le premier élément.

Il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en «sautant» d'élément en élément à l'aide des pointeurs suivant.

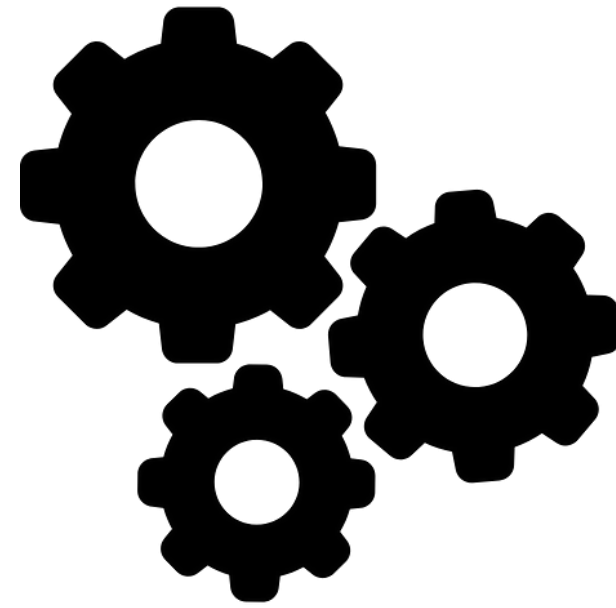
# Illustration



# Sommaire des éléments de contrôle?

En résumé, un schéma complet de structure de liste chaînée en C comprendra:

- la définition des **noeuds**,
- la définition de la **liste** (pour gérer la tête de liste),
- l'identification du **dernier élément** (dont le suivant pointe sur NULL)



**OPÉRATIONS DE BASE**

**& Fonctions de manipulation**

# Initialiser une liste

# Initialiser une liste

```
Liste *initialisation(){  
    Liste *liste = malloc(sizeof(*Liste));  
    Noeud *noeud = malloc(sizeof(*Noeud));  
    if (liste == NULL || noeud == NULL)  
    {  
        exit(EXIT_FAILURE);  
    }  
    noeud->nombre = 0;  
    noeud->suivant = NULL;  
    liste->premier = noeud;  
    return liste;  
}
```



# Initialiser une liste

Deux instructions essentielles...

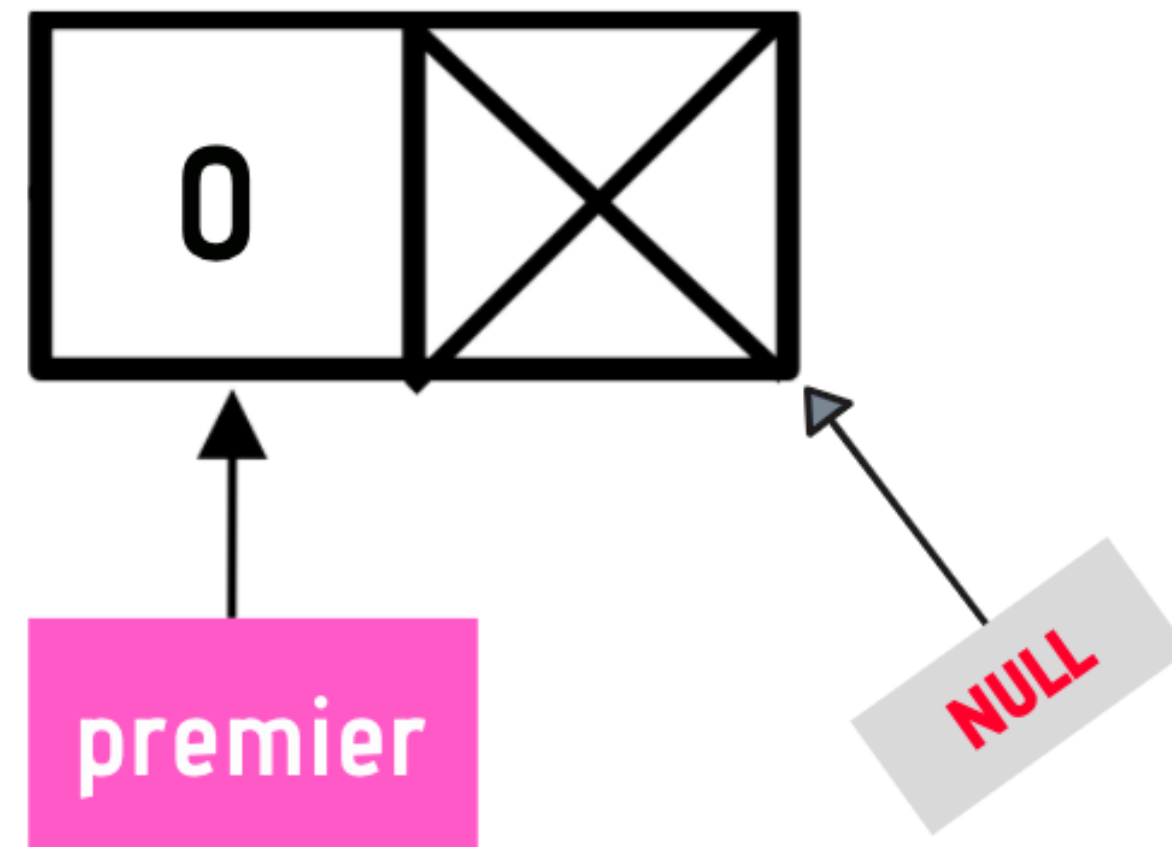
- `Liste *liste = malloc(sizeof(*Liste));`
- `Noeud *noeud = malloc(sizeof(*Noeud));`



On alloue dynamiquement  
les deux structures avec un  
malloc et sizeof

# Initialiser une liste

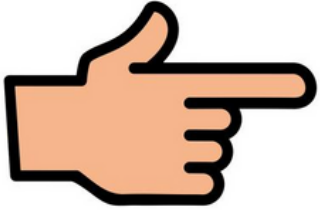
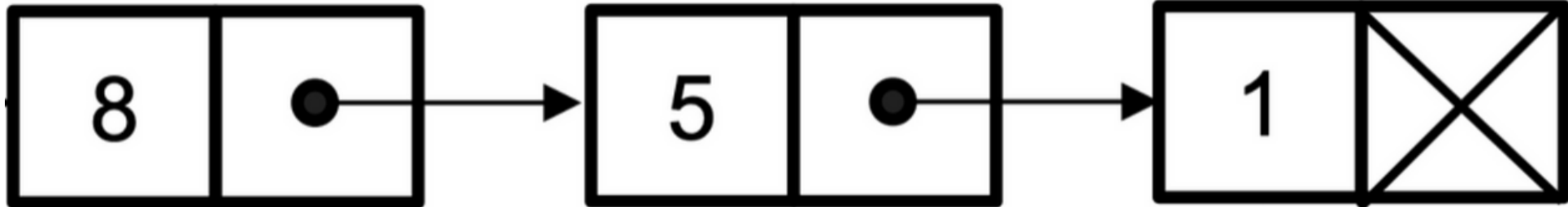
On obtient une liste composée d'un seul élément et ayant une forme semblable à la fig. ci-dessous.



**Insérer en tête de liste**

# Problème

Élément à insérer  2

Liste actuelle  

 **Problème:** comment insérer 2 en tête de liste ?

# Insérer en tête de liste

La fonction `insertion()` prend en paramètre la liste (qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer.

```
void insertion (Liste *liste, int nouveauNbre)
```

# Insérer en tête de liste

```
void insertion(Liste *liste, int nouveauNombre)
{
    Noeud *nouveau = malloc(sizeof(*Noeud));
    if (liste == NULL || nouveau == NULL)
    {exit(EXIT_FAILURE);}

    nouveau->nombre = nouveauNombre;
    /* Insertion de l'élément au début de la liste */
    nouveau->suivant = liste->premier;
    liste->premier = nouveau;
}
```

# Insérer en tête de liste

## Trois étapes

1. allouer l'espace nécessaire au stockage du nouvel élément et y placer le nouveau nombre **nouveauNbre**
2. faire pointer le suivant du nouvel élément vers l'actuel premier élément de la liste
3. faire pointer le pointeur **premier** de **Liste** vers notre nouvel élément

# Insérer en tête de liste

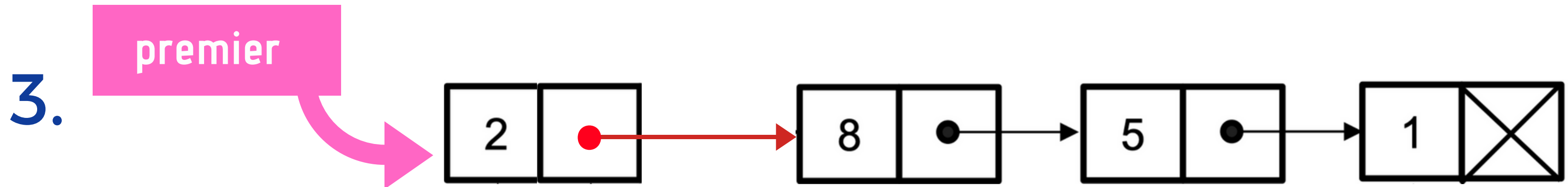
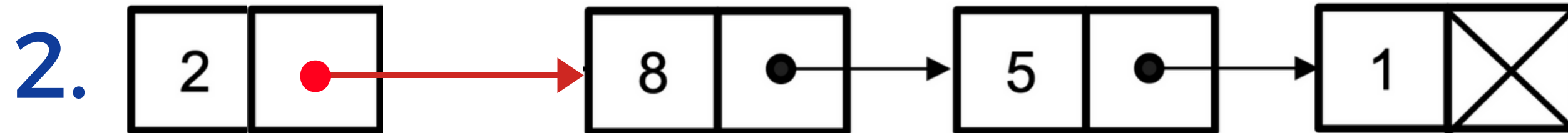
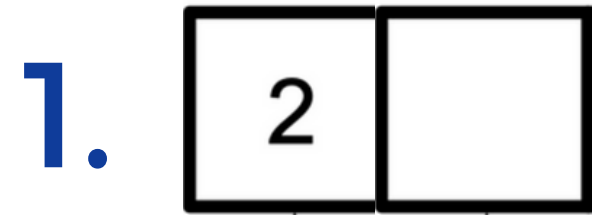


**Attention à l'ordre des étapes !**

Si vous inversez les étapes, vous perdez l'adresse du premier élément de la liste !



# Exemple



**Supprimer en tête de liste**

# Supprimer en tête de liste

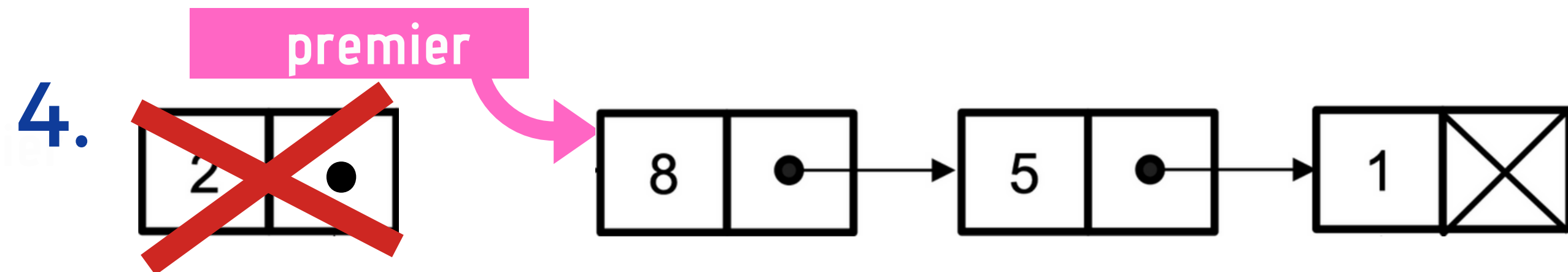
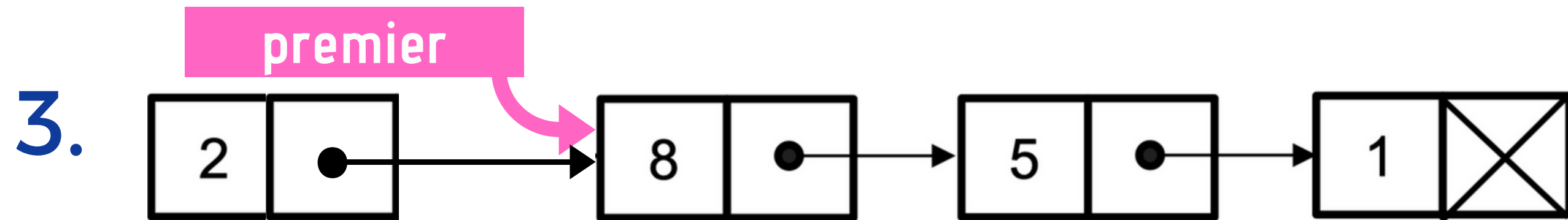
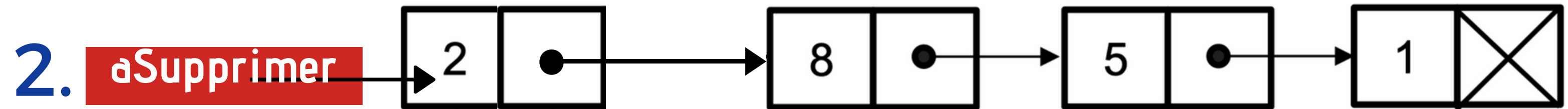
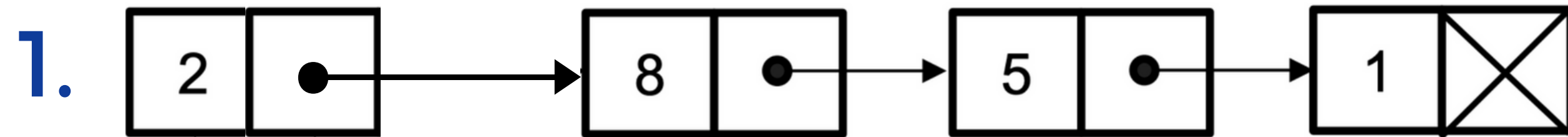
La fonction `suppression()` prend en paramètre la liste (qui contient l'adresse du premier élément).

```
void suppression (Liste *liste)
```

# Supprimer en tête de liste

```
void suppression(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }
    if (liste->premier != NULL)
    {
        Element *aSupprimer = liste->premier;
        liste->premier = liste->premier->suivant;
        free(aSupprimer);
    }
}
```

# Exemple



**Afficher une liste**

# Afficher une liste

La fonction `affichage()` prend en paramètre la liste (qui contient l'adresse du premier élément

```
void afficherListe (Liste *liste)
```

# Afficher une liste

```
void afficherListe(Liste *liste)
{
    if (liste == NULL)
    {exit(EXIT_FAILURE);}

    Element *actuel = liste->premier;
    while (actuel != NULL)
    {
        printf("%d -> ", actuel->nombre);
        actuel = actuel->suivant;
    }
    printf("NULL\n");
}
```



# Exercices d'application

Ecrire un programme avec les fonctions suivantes :

- **listeVide( )** pour tester si une liste est vide
- **listTri( )** qui trie une liste par ordre croissant
- **renverserListe( )** qui renverse une liste