Cours du Module M22 : Structures de données en C

# Chapitre 2 : Les listes chaînées

# Chapitre 2 : Les listes chaînées

## Plan :

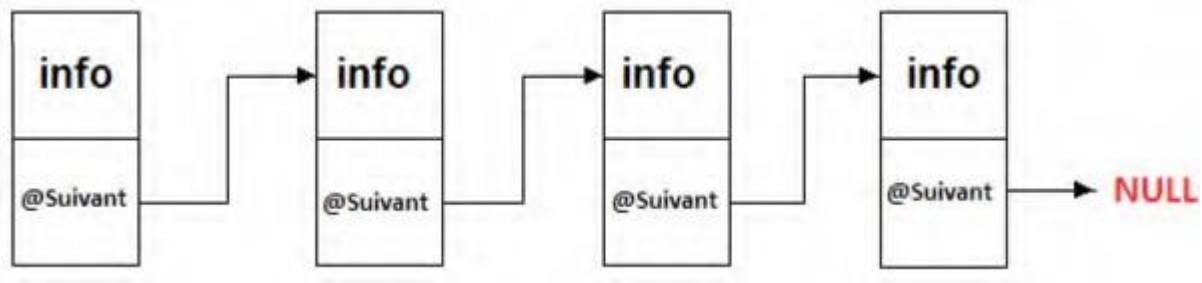
- Les listes chainées
- Les listes simplement chaînées
- Les listes doublement chaînées

# Listes chaînées

# Listes chaînées

## Notion de liste

- Une **liste chainée** est une **structure de données** qui permet de stocker une séquence de **structures** d'un même type.
- Une **liste chainée** est simplement une liste de **structures** de **même type** dans laquelle chaque élément contient :
  - des **données** : nom, prénom, âge, …
  - un **pointeur (adresse)** vers l'**élément suivant** ou une marque de fin s'il n'y a pas d'élément suivant.
- C'est ce **pointeur** vers l'élément **suivant** qui fait la "**chaine**" et permet de retrouver chaque élément de la liste.

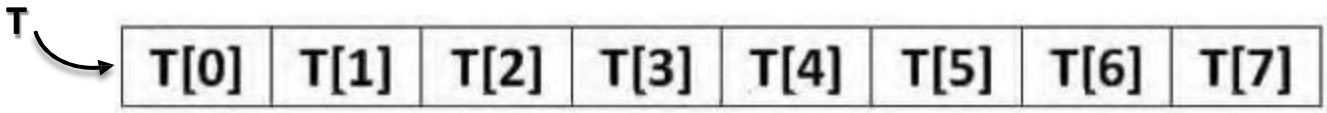


# Listes chaînées

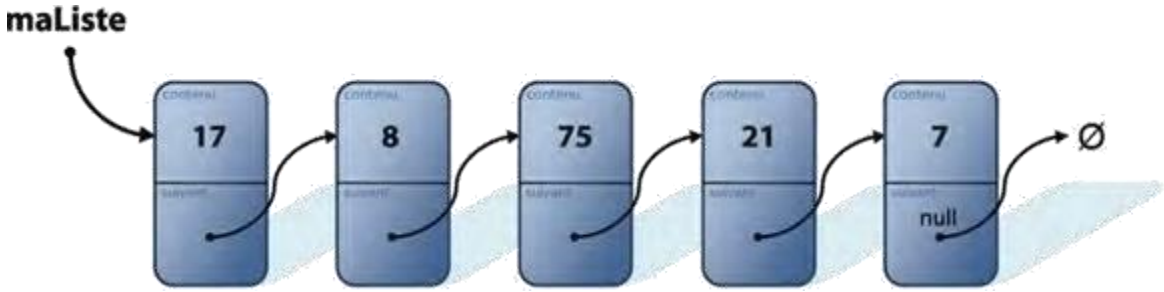
## Notion de liste

- Les **listes chainées** servent à gérer des **ensembles de données**, un peu comme les **tableaux**.
- C'est un moyen d'**organiser** une **série de données** en mémoire.
- Les **listes chainées** sont cependant plus **efficaces** pour réaliser des opérations comme l'**insertion** et la **suppression** d'éléments.

**Tableau T :**



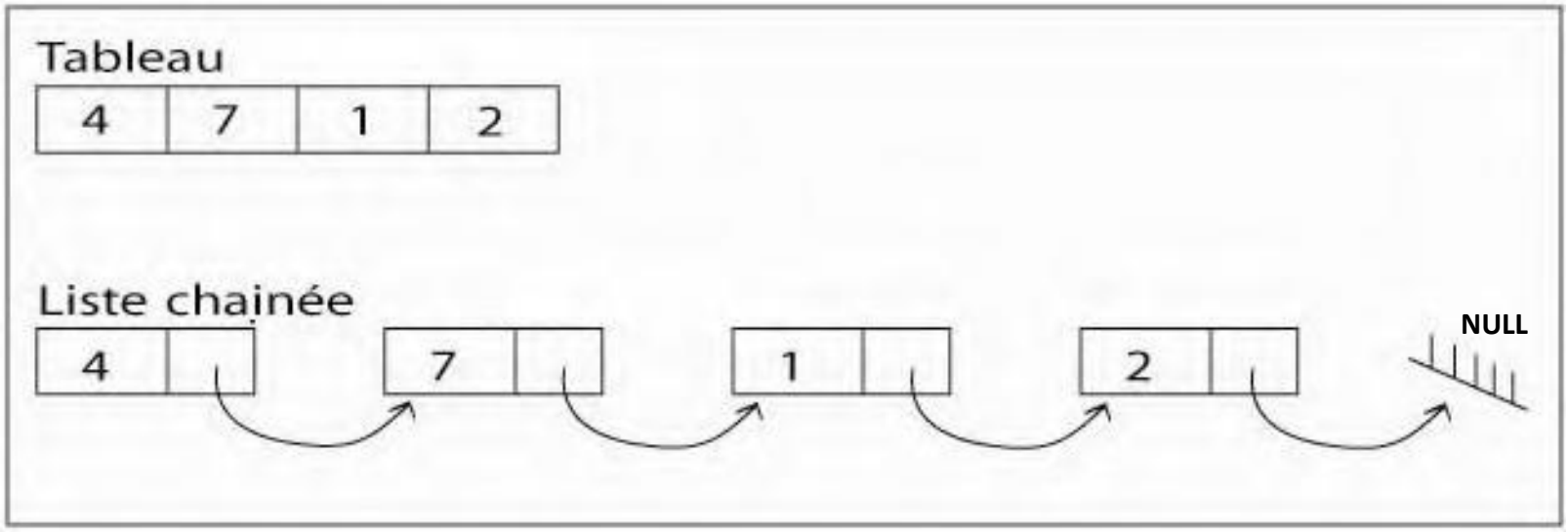

**Liste chainée maListe:**



# Listes chaînées

## Listes chaînées Vs. Tableaux

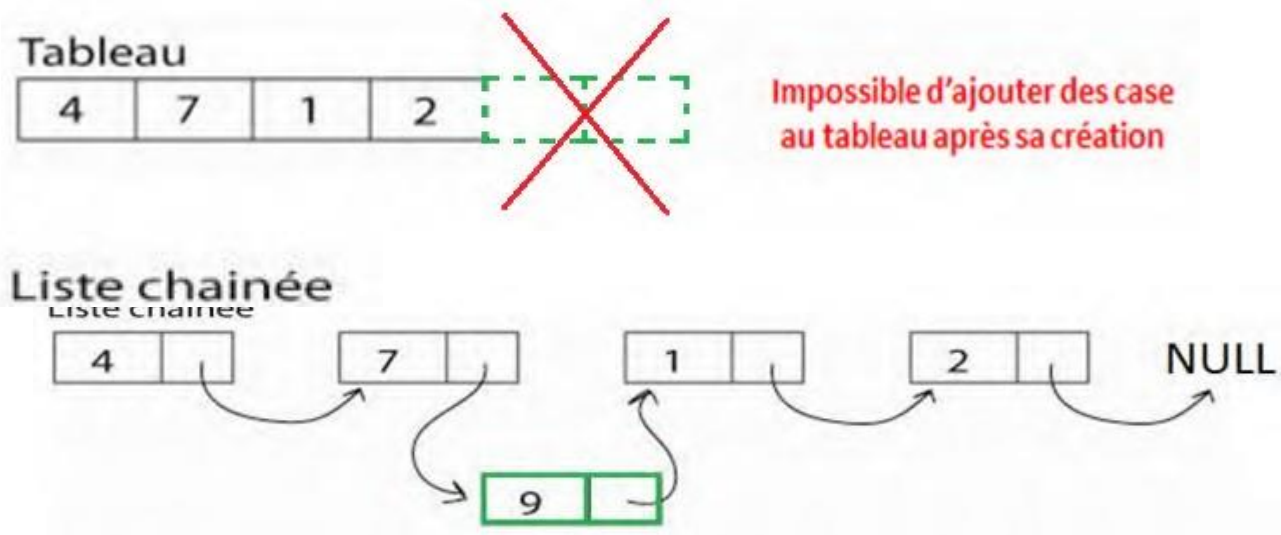
- **Différence 1 : Accès aux éléments :**
  - Dans un tableau on accède à n'importe quel élément par son **indice** : → **accès immédiat**.
  - Dans une liste chainée, on a accès aux éléments **un après l'autre**, à partir du premier élément : → **accès séquentiel**.



# Listes chaînées

## Listes chaînées Vs. Tableaux

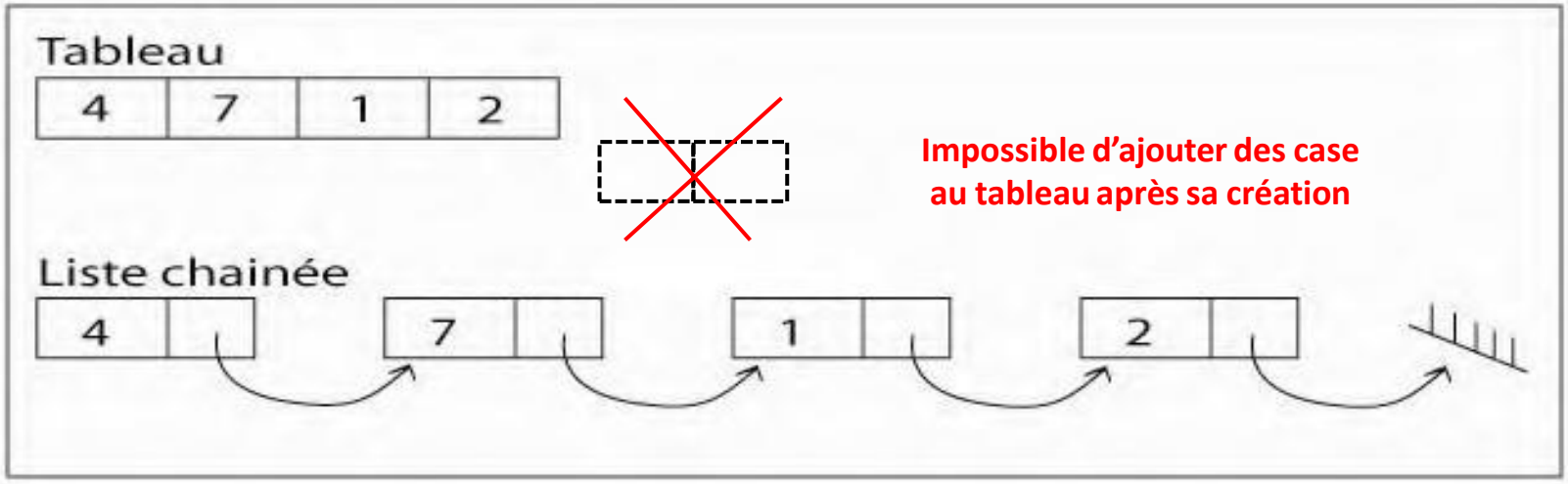
- **Différence 2 : Taille :**
  - Un **tableau** a une **taille fixe** (impossible d'agrandir un tableau). Il n'est pas possible d'**insérer une case** au milieu du tableau.
  - Les **listes** permettent d'organiser les données en mémoire de manière beaucoup **plus flexible** : on peut toujours insérer un élément à une liste.



# Listes chaînées

## Listes chaînées Vs. Tableaux

- **Différence 3 : Stockage :**
  - Les éléments d'un **tableau** sont stockés dans la mémoire de manière **contiguë**.
  - Les éléments d'une **liste chaînée** sont **dispersés** dans la mémoire et reliés entre eux par des **pointeurs**.



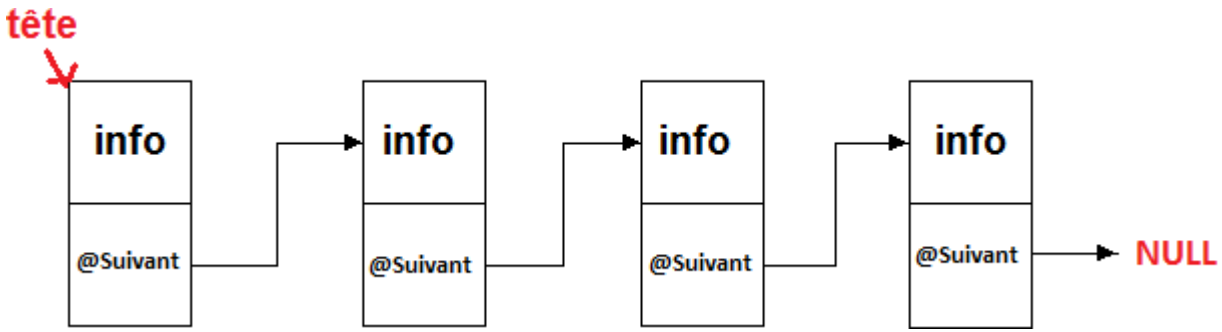
# Listes chaînées

## Caractéristiques

- Une **liste chaînée** est une **structure linéaire** qui n'a pas de dimension fixée à sa création. La **dimension** peut être modifiée selon la place disponible en mémoire.

- Les **listes chaînées** utilisent l'**allocation dynamique** de mémoire et peuvent avoir une taille qui varie pendant l'exécution.

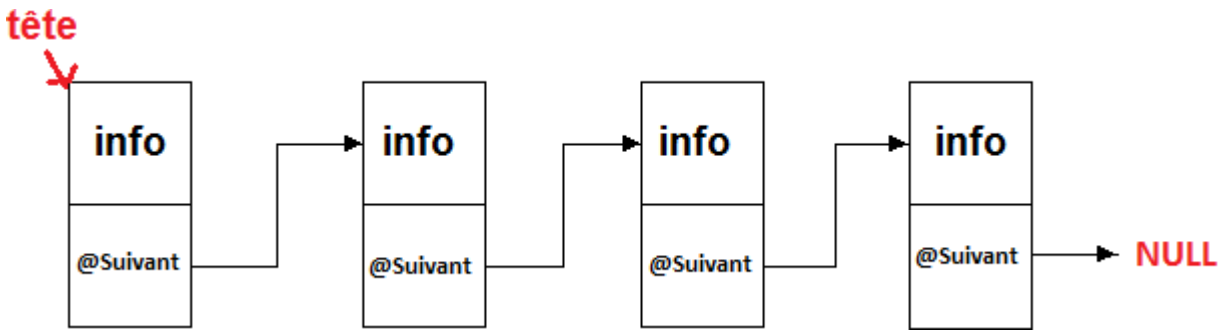
- L'espace est réservé au fur et à mesure des créations des éléments de la liste. La seule **limite** étant la **taille** de la mémoire centrale.



# Listes chaînées

## Caractéristiques

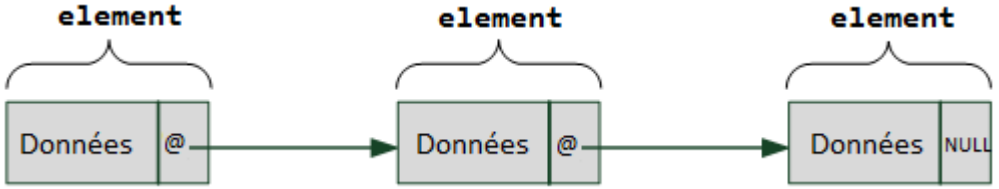
- La **liste chainée** est définie (connue) par l'adresse de son premier élément : sa **tête.**
- Les **éléments** de la **liste chainée** sont **accessibles** uniquement à partir de la **tête** de la liste.
- On accède à un **élément** de la liste chainée en **parcourant** les éléments grâce à leurs pointeurs.
- Le **dernier** élément de la **liste chainée** ne pointe sur rien : pointe sur **NULL**.



# Listes chaînées

## L'élément d'une liste

- L'élément de base d'une **liste chaînée** est défini par une **structure.**
- La construction d'une **liste chaînée** consiste à assembler des **structures** en les liant entre elles à l'aide de **pointeurs**.



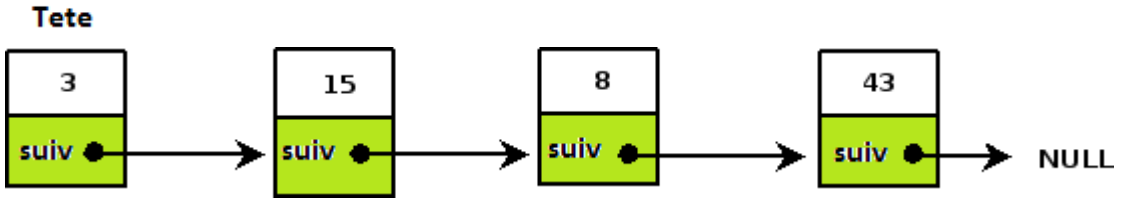

- Cette structure contient des **données** ainsi qu'un **pointeur** sur une structure de même type.
- ***Exemple*** : une **liste chaînée** peut être créée en dupliquant plusieurs fois la **structure** suivante :

```
typedef struct element element;
struct element {
                //1 : les datas pour l'application courantes
                char nom[80];
                // 2 : le pointeur pour l'élément suivant
                element  *suiv;
              };
```

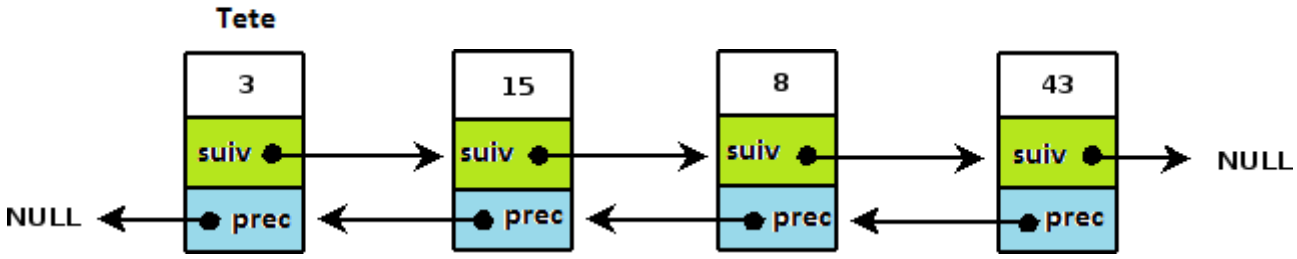
# Listes chaînées

## Types de listes chaînées

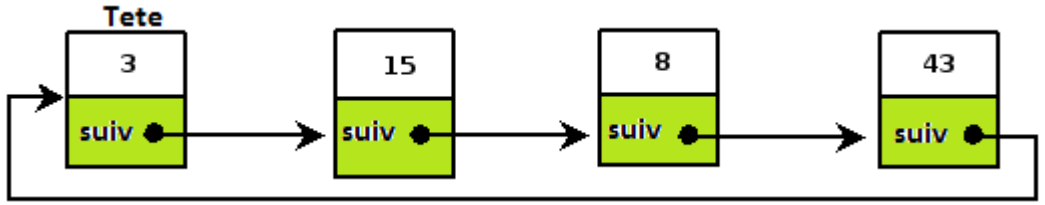
- On distingue plusieurs types de listes chaînées :
  - listes simplement chaînées :




  - listes doublement chaînées :



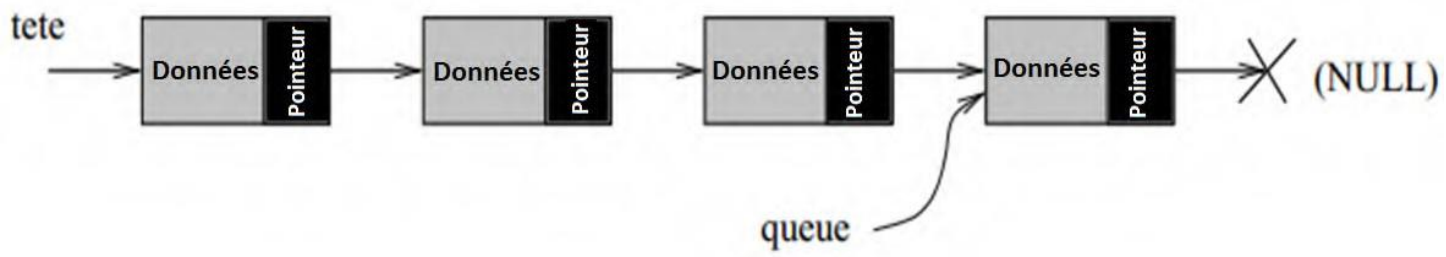

  - listes circulaires :



# Listes Simplement Chaînées (LSC)

# Listes simplement chaînées
## Définition

- Une **liste simplement chaînée (LSC)** est composée d'éléments distincts liés par un simple **pointeur**.

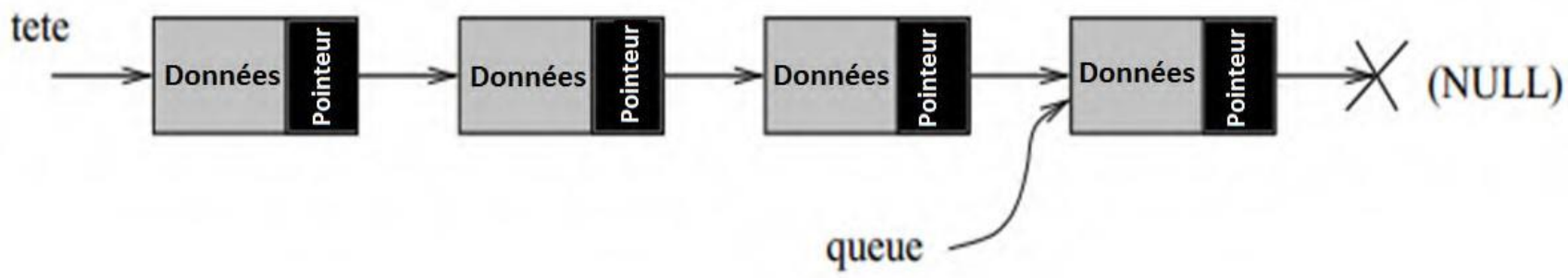



- Chaque **élément** d'une LSC est formée de 2 parties :
  - un **champ** : il contient les **données**.
  - un **pointeur** (généralement appelé ***suivant*** ou ***next***) : c'est un pointeur vers l'élément suivant de la liste.

# Listes simplement chaînées

## Caractéristiques

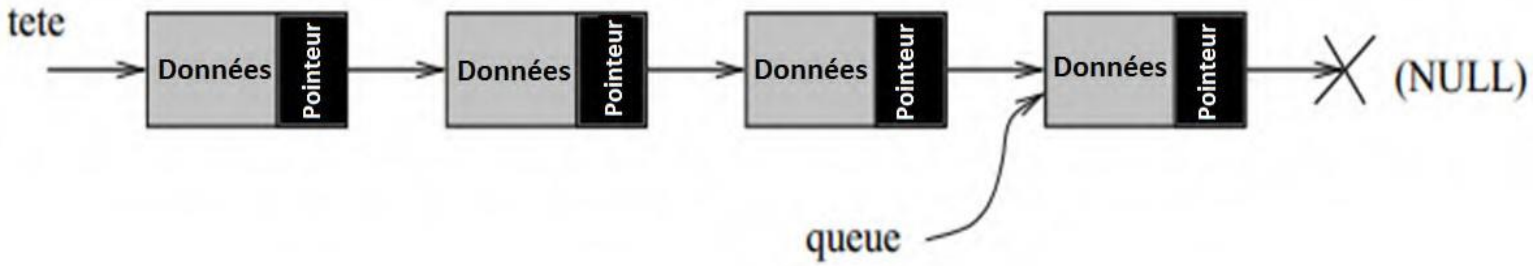
- Le **premier élément** d'une LSC est sa **tête** (appelé **tete** ou **premier**).
- Le **dernier élément** d'une LSC est sa **queue**.
- Le **pointeur** du **dernier élément** est initialisée la valeur **NULL**.
- Dans le cas d'une **LSC vide** le pointeur de la **tête** contient **NULL**.
- Pour accéder à un élément d'une **liste simplement chaînée**, on part de la **tête** et on passe d'un **élément** à l'autre à l'aide du pointeur **suivant** associé à chaque élément .



# Listes simplement chaînées

## Caractéristiques

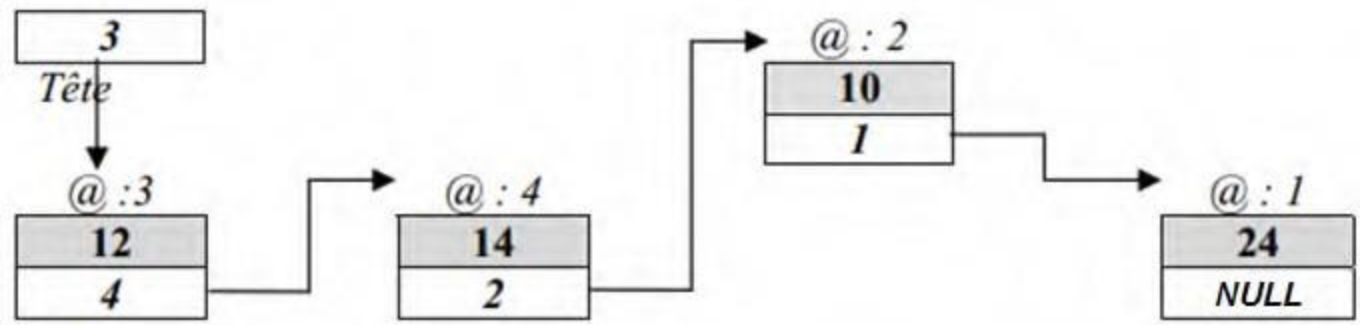
- Une **LSC** est définie par l'**adresse** de son **premier élément.**
- La **perte de l'adresse du premier élément** d'une liste conduit à la **perte de la liste toute entière** : **LSC ⇔ @tete**
- Les éléments **ne sont pas contigus** en mémoire => la suppression d'un élément **sans précaution** ne permet plus d'accéder aux éléments suivants.
- Une **LSC** ne peut être parcourue que dans un sens (tête →queue).



# Listes simplement chaînées

## Caractéristiques

- **Exemple** : Soit la liste simplement chainées suivante :

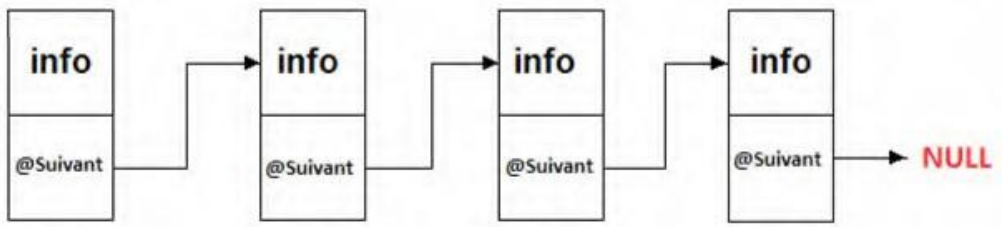



- Le **1er élément** de la liste **vaut 12** à l'adresse 3 (début de la liste chaînée).
- Le **2ème élément** de la liste **vaut 14** à l'adresse 4 (car le pointeur de la cellule d'adresse 3 est égal à 4)
- Le **3ème élément** de la liste **vaut 10** à l'adresse 2
- Le **4ème élément** de la liste **vaut 24** à l'adresse 1.

# Listes simplement chaînées

## Manipulation des LSC

- Les opérations sur les **listes simplement chaînées** peuvent être :
  - Créer une liste
  - Ajouter un élément (en début, au milieu ou en fin)
  - Supprimer un élément(en début, au milieu ou en fin)
  - Modifier un élément
  - Permuter deux éléments
  - Parcourir une liste
  - Rechercher une valeur dans une liste.
  - Supprimer une liste
  - .....



# Listes simplement chaînées

## Construction d'une LSC

- En langage C, une **liste simplement chaînée** peut être implémentée sous forme d'une **structure** : chaque **élément** de la liste est une **structure**.
- **Un élément de la liste :**
  - Pour les exemples, nous allons créer une liste chaînée de nombres **entiers**.
  - Chaque **élément** de la liste aura la forme de la **structure** suivante :

```
typedef struct element element;
struct element
    {
      int val;
      element *nxt;
    };
```

- **NB**: Dans cet exemple, la structure de base ***element*** est utilisée pour créer une liste chaînée de nombres entiers. On pourrait aussi bien créer une liste chaînée contenant des décimaux, tableaux, ..

# Listes simplement chaînées

## Construction d'une LSC

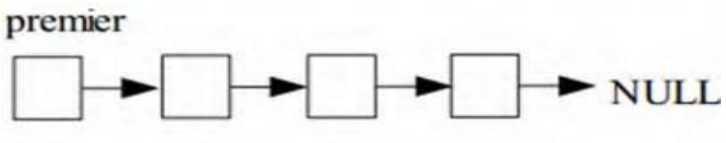
- **Créer une LSC (vide) :**
  - Il est important de toujours **initialiser** la liste chaînée à **NULL**, sinon, elle sera considérée comme contenant au moins un élément.
  - La création d'une liste vide aura la forme suivante :

```
typedef struct element element;
 struct element
      {
          int val;
          element *nxt;
      };

int main() {
/* Déclarons 4 listes chaînées de façons différentes mais équivalentes */
    element *ma_liste = NULL;
    element *tete = NULL;
    element *premier = NULL;
    element *ma_liste2 = NULL;
    return 0;
}
```

# Listes simplement chaînées

## Construction d'une LSC

- Deux positions sont très importantes dans une LSC : le **début** et la **fin**, souvent désignées par "**premier** et **dernier**" ou "**tête** et **queue** ".
  - Sans le **premier** impossible de savoir où **commence** la chaine.
  - Sans le **dernier** impossible de savoir où s'**arrête** la chaine.



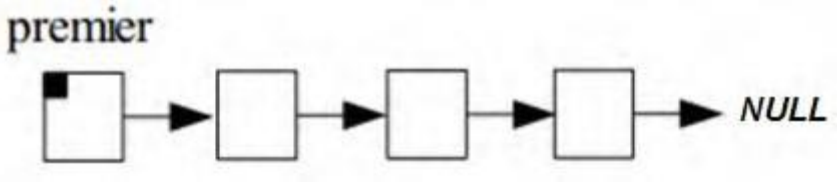

- Le **début** d'une LSC est donné par l'**adresse** du **premier élément** de la liste (la **tete**).
- La **fin** d'une LSC est marquée lorsque l'adresse de l'**élément suivant** vaut **NULL**.
- ***Remarque*** :
  - En général une **LSC** prend le **nom** du **premier élément.** Ce premier élément est un **pointeur** déclaré dans le **main()** et **initialisé** à **NULL** :

```
int main() {
    element *premier = NULL;
    return 0;
}
```

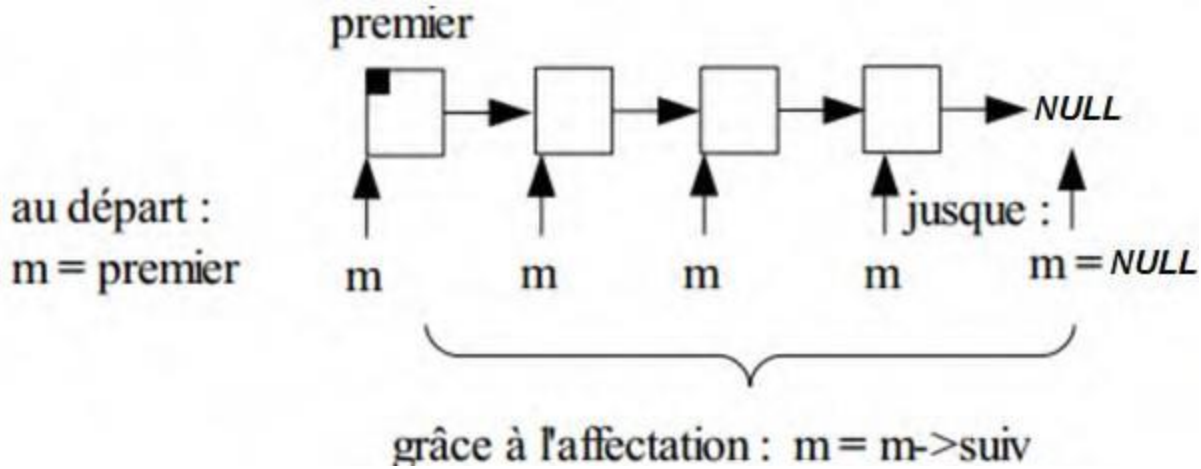
# Listes simplement chaînées

## Parcourir une LSC

- Soit la liste simplement chainée suivante :



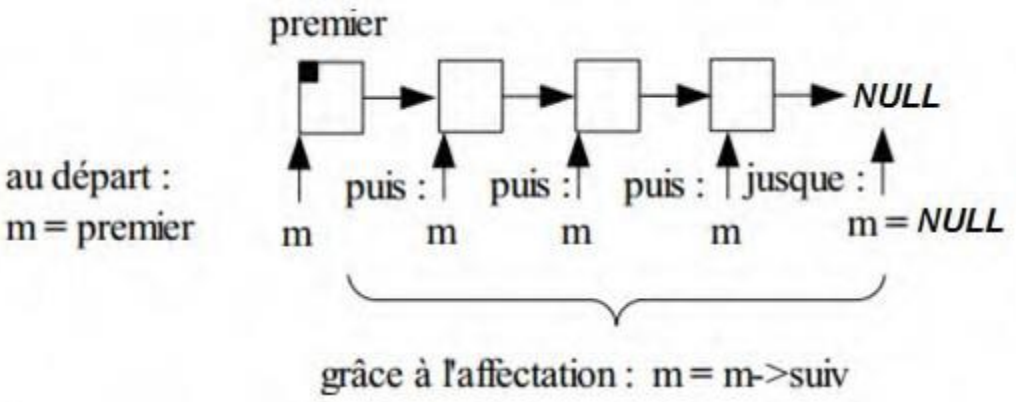

- Pour parcourir la liste, on doit passer d'un élément à l'autre grâce à un **pointeur intermédiaire m**, ce qui donne par exemple :



# Listes simplement chaînées

## Parcourir une LSC

- En langage C, on peut parcourir la liste simplement chainée comme suit :

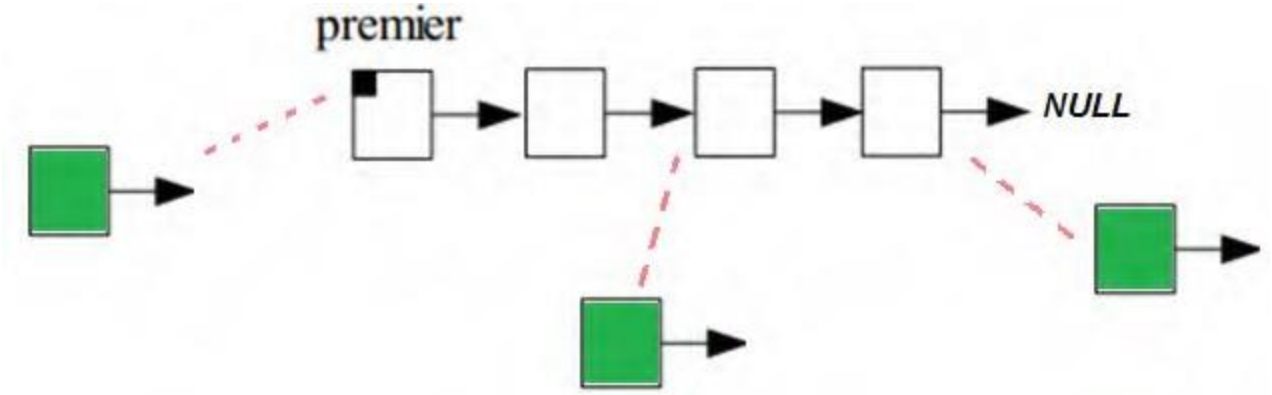



```
element *m ; // au départ prendre l'adresse du premier élément
m=premier;
// tant que m ne contient pas le marquage de fin de chaine
while (m!=NULL){
// affichage de la valeur
printf("Valeur : %d\n",m->val);
// prendre ensuite l'adresse de l'élément suivant
m=m->nxt;
```

# Listes simplement chaînées

## Ajouter un élément

- Lorsque nous voulons ajouter un élément dans une liste chaînée, il faut savoir où l'insérer.
- Les ajouts génériques des listes chaînées sont les ajouts :
  - en tête de liste.
  - au milieu de la liste.
  - en fin de liste.

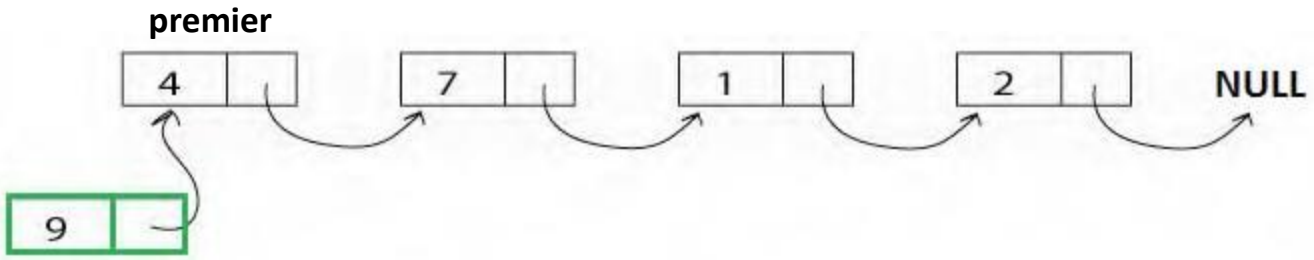


# Listes simplement chaînées

## Ajouter un élément

### Ajouter en tête de liste

- La fonction d'ajout d'un élément en tête de la liste doit permettre de :
  - **créer un élément** (lui allouer de la mémoire).
  - **lui assigner la valeur** que l'on veut ajouter.
  - **raccorder cet élément** à la liste passée en paramètre.
- Lors d'un ajout en tête, on devra donc assigner à **suivant** du nouveau élément l'adresse du **premier** élément de la liste passé en paramètre.



# Listes simplement chaînées

## Ajouter un élément

- **Ajouter en tête de liste**
  - La fonction d'ajout d'un élément en tête d'une LSC est la suivante :

```
typedef struct element element;
struct element
    {
      int val;
      element *nxt;
    };
```

**Structure de base**

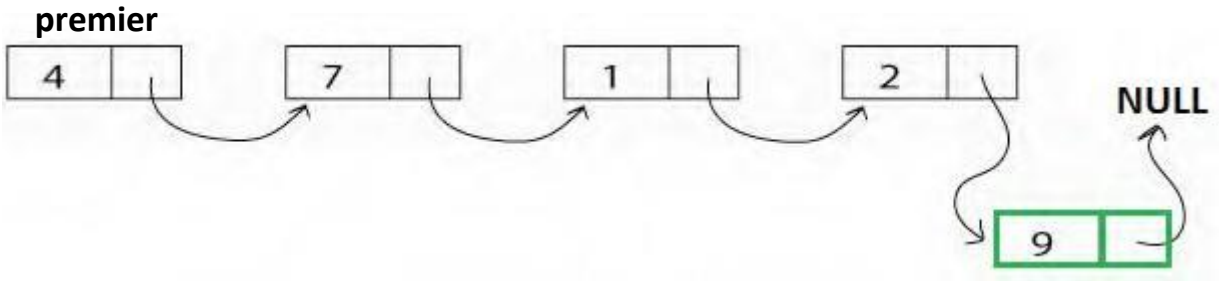
```
element *ajouterEnTete(element *premier, int valeur) {
    /* On crée un nouvel élément */
    element* nouvelElement = malloc(sizeof(element));

    /* On assigne la valeur au nouvel élément */
    nouvelElement->val = valeur;

    /* On assigne l'adresse de l'élément suivant au nouvel élément */
    nouvelElement->nxt = premier;

    /* On retourne la nouvelle liste, i.e. le pointeur sur le premier élément */
    return nouvelElement;
}
```

La **liste** est définie par le pointeur **premier**.

# Listes simplement chaînées

## Ajouter un élément

- **Ajouter en fin de liste**
  - Pour ajouter un élément en fin de liste, il faut :
    - créer un nouvel élément (lui allouer de la mémoire)
    - lui assigner sa valeur.
    - mettre le **suivant** du nouvel élément à **NULL** (il va terminer la liste).
    - faire pointer le **dernier** élément de liste originale sur le **nouvel élément**.




- Pour atteindre le dernier élément de liste, il faut parcourir la liste jusqu'à la fin grâce pointeur temporaire.

# Listes simplement chaînées

## Ajouter un élément

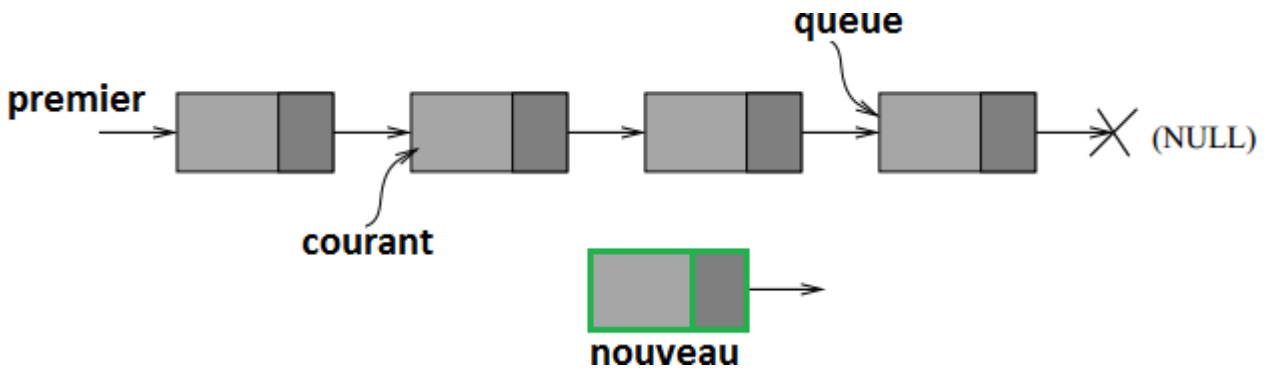
- **Ajouter en fin de liste**
  - La fonction d'ajout d'un élément en fin de la liste est la suivante :

```
element *ajouterEnFin(element *premier, int valeur){

element* nouvelElement = malloc(sizeof(element));
nouvelElement->val = valeur;

/* On ajoute en fin, donc aucun élément ne va suivre */
nouvelElement->nxt = NULL;

if(premier == NULL) /* Si la liste est vidé */
    {     return nouvelElement;     }
else  /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire */
    {
    element* temp=premier;
    while(temp->nxt != NULL)
        {  temp = temp->nxt;  }
    temp->nxt = nouvelElement;
    return premier;
    }
}
```

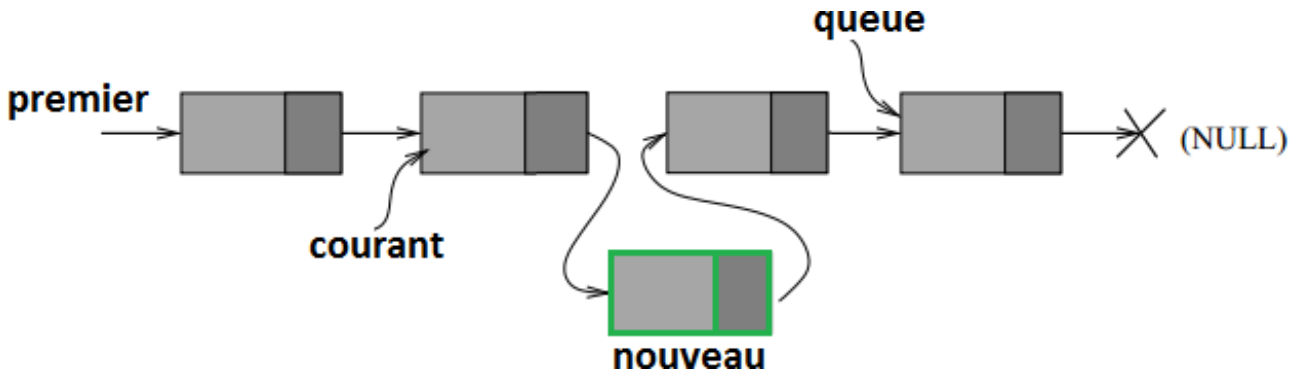
# Listes simplement chaînées

## Ajouter un élément

- **Ajouter au milieu de liste**
  - Insertion d'un élément **après** l'élément référencé par le pointeur **courant** :
    - **Avant insertion :**



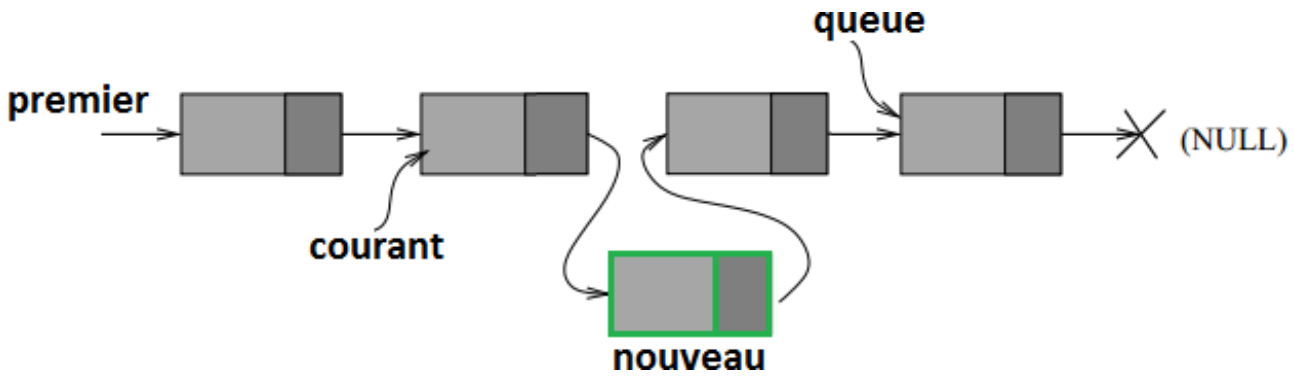

    - **Après insertion :**



# Listes simplement chaînées

## Ajouter un élément

- **Ajouter au milieu de liste**
  - Les opérations à effectuer sont (dans l'ordre) :
    - créer un **nouvel élément** (lui allouer de la mémoire).
    - lui assigner la valeur que l'on veut ajouter.
    - faire pointer le **nouvel élément** vers l'élément le **suivant de courant**.
    - faire pointer l'élément **courant** vers le **nouvel élément**.




  - ***NB*** : Le cas ou la liste est vide (premier == NULL) doit être traité à part.

# Listes simplement chaînées

## Ajouter un élément

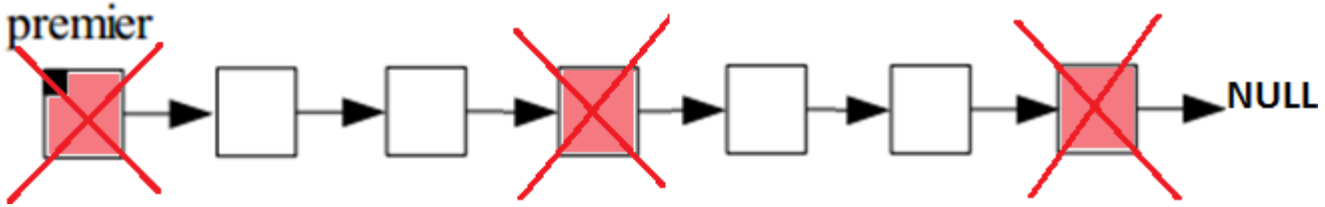
- **Ajouter au milieu de liste**
  - La fonction d'ajout d'un élément (après l'élément référencé par le pointeur **courant** :

```
element *insertion_nimporte_ou(element *premier, element *courant, int valeur){
    element* nouveau = malloc(sizeof(element));
    nouveau->val = valeur;
    /* ajout du nouveau */
    if (courant !=NULL && premier !=NULL)
        {
            nouveau->nxt=courant->nxt ;
            courant->nxt=nouveau ;
            return premier ;
        }
    else /* cas d'une liste vide ou d'une insertion en tete */
        {
            nouveau->nxt=premier ;
            return nouveau ;
        }
}
```

# Listes simplement chaînées

## Supprimer un élément

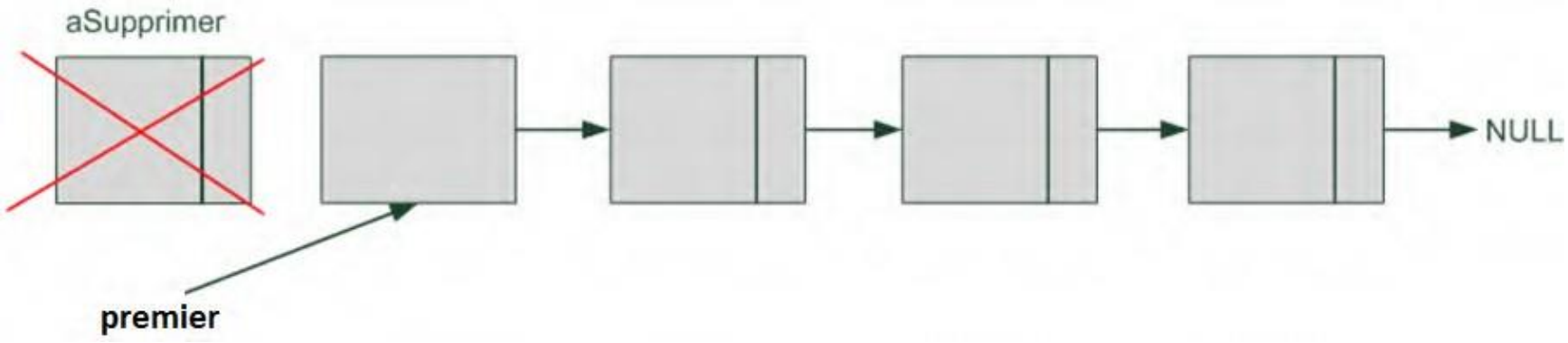
- Lorsque nous voulons supprimer un élément de la liste chaînée, il faut savoir quel élément supprimer.
- Les suppressions génériques des listes chaînées sont les suppressions de l'élément :
  - en tête de liste
  - du milieu de la liste
  - en fin de liste.



# Listes simplement chaînées

## Supprimer un élément

- **Supprimer l'élément en tête de liste**
  - Pour supprimer l'élément en tête de liste, il faut :
    - faire pointer **premier** vers le **second** élément
    - **supprimer** le premier élément avec un free

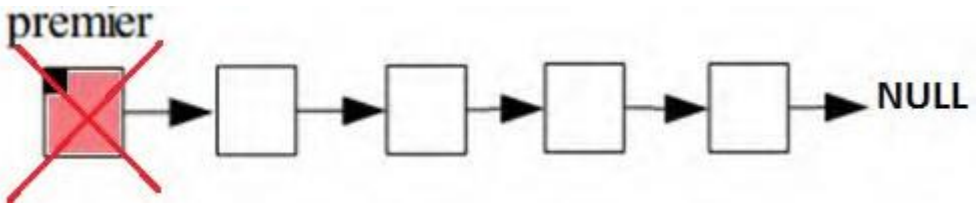


# Listes simplement chaînées

## Supprimer un élément

- **Supprimer l'élément en tête de liste**
  - La fonction de suppression de l'élément en tête de liste est la suivante :

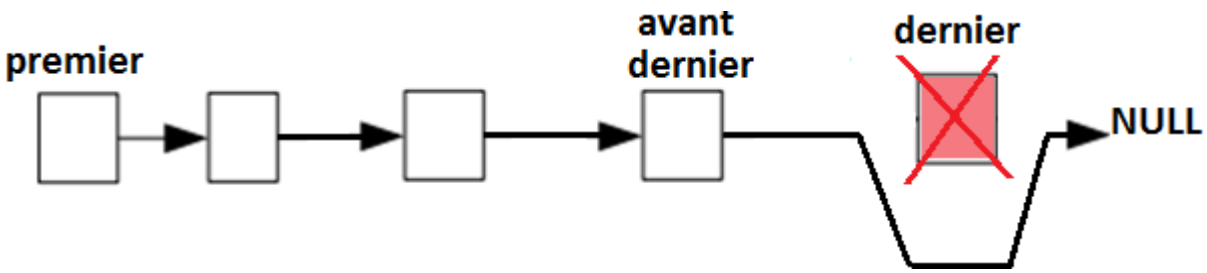
```
element *supprimerElementEnTete(element *premier)  {
    if(premier != NULL) /* renvoyer l'adresse de l'élément en 2ème position */
    {
        element* aRenvoyer = premier->nxt;
        free(premier);/* On libère le premier élément */
        /* On retourne le nouveau début de la liste */
        return aRenvoyer;
    }
    else
    { return NULL;}
}
```



# Listes simplement chaînées

## Supprimer un élément

- **Supprimer l'élément en fin de la liste**
  - Pour supprimer l'élément en fin de liste, il faut :
    - se positionner sur l'**avant dernier** élément de la liste
    - supprimer le **suivant** de l'**avant dernier**
    - mettre **suivant** de l'**avant dernier** à **NULL**.




  - ***NB*** :
    - Il faut que la liste ne soit pas vide
    - le cas ou il n'y a qu'un seul élément dans la liste doit être traité à part .

# Listes simplement chaînées

## Supprimer un élément

- **Supprimer l'élément en fin de la liste**
  - La fonction de suppression de l'élément en fin de liste est la suivante :

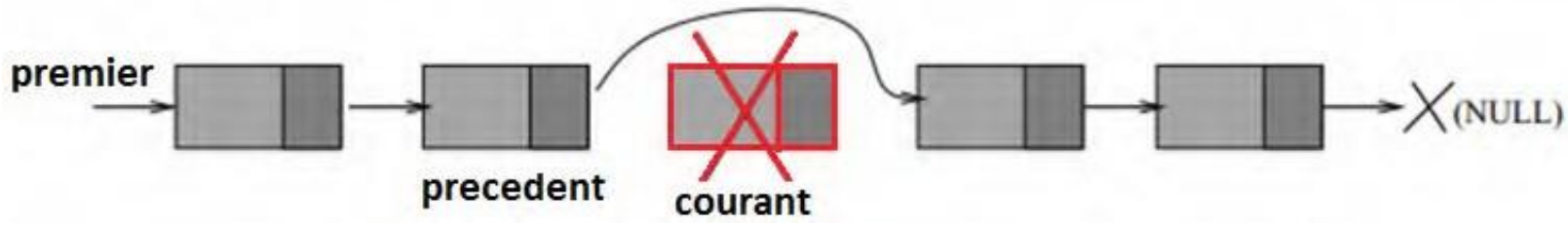
```
element *supprimerElementEnFin(element *premier) {
    if(premier == NULL) return NULL; // si la liste est vide
    if(premier->nxt == NULL) /* Si la liste contient un seul élément */
        { free(premier); return NULL;  }
    /* Si la liste contient au moins deux éléments */
    element* tmp = premier;
    element* ptmp = premier; /* ptmp va stocker l'adresse de tmp */
    /* Tant qu'on n'est pas au dernier élément */
    while(tmp->nxt != NULL)
    {   ptmp = tmp;
        tmp = tmp->nxt;
    }
    /* A la sortie de la boucle, tmp pointe sur le dernier élément, et ptmp sur
    l'avant-dernier (qui devient la fin de la liste */
    ptmp->nxt = NULL;
    free(tmp);
    return premier;
}
```

# Listes simplement chaînées

## Supprimer un élément

- **Supprimer un élément du milieu de la liste**
  - Suppression d'un élément pointé par **courant** :
    - **Avant suppression :**


    - **Après suppression :**

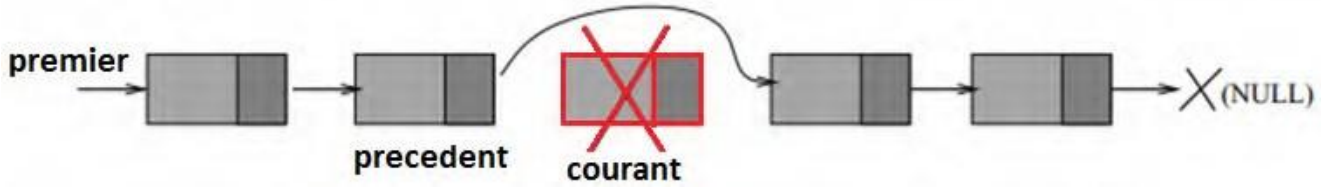


# Listes simplement chaînées

## Supprimer un élément

- **Supprimer un élément du milieu de la liste**
  - Suppression d'un élément pointé par courant :

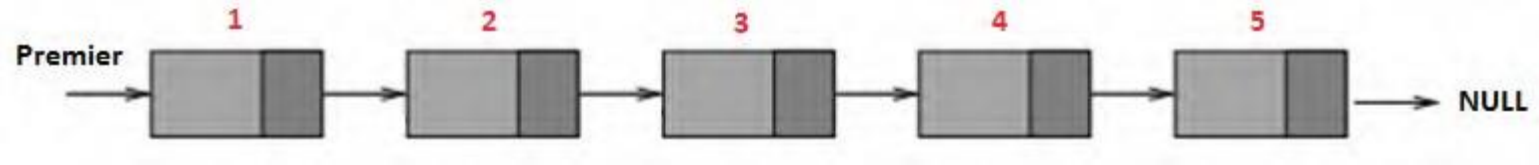
```
element *suppression(element *premier, element *courant){
      if (courant!=premier) /* pas le premier element */
            {
             element *precedent=premier;
             while (precedent->nxt!=courant)
             precedent=precedent->nxt;
             precedent->nxt=courant->nxt;
            }
      else /* suppression du 1er element */
      premier=courant->nxt;
      free(courant);
      return premier;
}
```



# Listes simplement chaînées

## Rechercher le i-ème élément

- Pour rechercher l'**adresse de l'élément** d'indice **i**, il suffit de se déplacer **i** fois le long de la liste et de renvoyer l'élément qui à l'indice **i**.
- Si la liste contient moins de i élément(s), la fonction renvoie NULL.




```
element *element_i(element *premier, int indice) {
    int i;
    element* tmp = premier;
    /* On se déplace de i cases, tant que c'est possible */
    for(i=1; i<= indice && tmp != NULL; i++)
        {  tmp = tmp->nxt;          }

    /* Si l'élément est NULL, c'est que la liste contient moins de i éléments */
     if(tmp == NULL)
        return NULL;
    else
        return tmp     ; /* Sinon on renvoie l'adresse de l'élément i */
}
```

# Listes simplement chaînées

## Compter le nombre d'éléments

- Pour compter le nombre d'éléments d'une liste, on doit :
  - parcourir la liste de bout en bout.
  - incrémenter d'un pour chaque nouvel élément trouvé.


```
int nombreElements(element *premier)
{
    /* Si la liste est vide, il y a 0 élément */
    if(premier == NULL)
        return 0;

    /* Sinon, il y a un élément (celui que l'on est en train de traiter)
    plus le nombre d'éléments contenus dans le reste de la liste */
    return nombreElements(premier->nxt)+1;
}
```

# Listes simplement chaînées

## Afficher la liste

- Pour visualiser ce que contient une liste simplement chaînée, une fonction d'affichage peut être définie comme suit :

```
void afficherListe(element *premier){
element *tmp = premier;
/* Tant que l'on n'est pas au bout de la liste */
while(tmp != NULL)
    {
        printf("%d ", tmp->val); /* On affiche */
        tmp = tmp->nxt; /* On avance d'une case */
    }
}
```

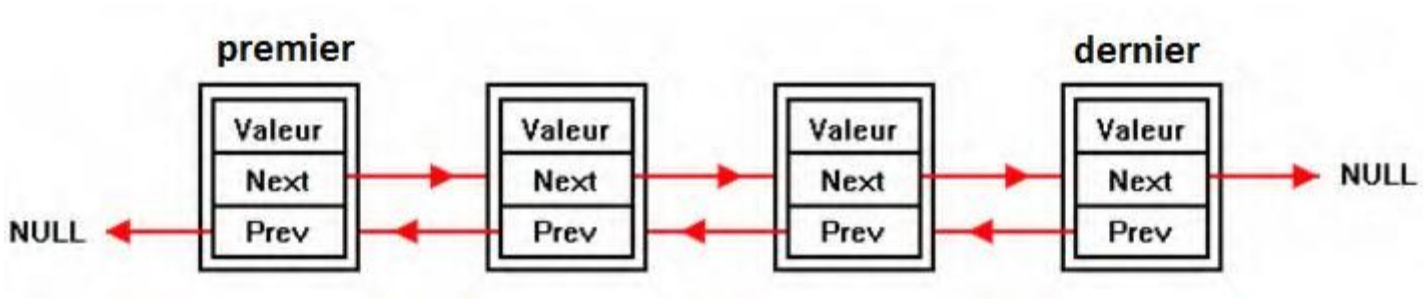
  - Il suffit de partir du premier élément et d'afficher chaque élément un à un en « sautant » d'un élément à un autre.

# Listes Doublement Chaînées (LDC)

# Listes doublement chaînées

## Définition

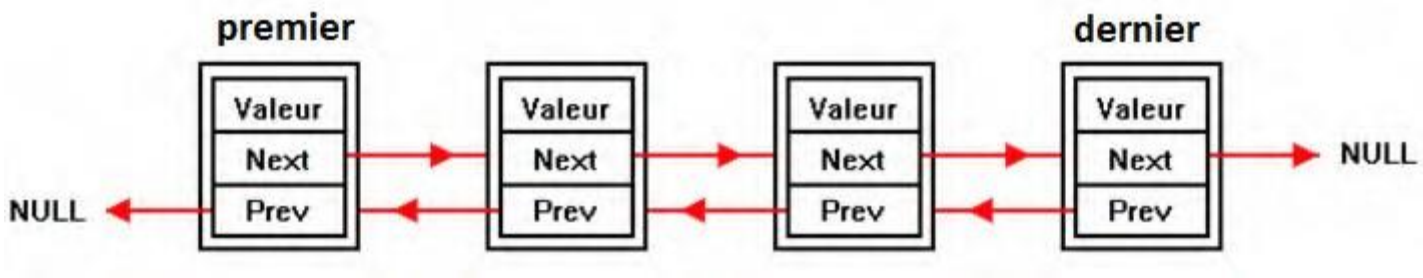
- Une **liste doublement chaînée** (**LDC**) est basée sur le même principe que la **liste simplement chaînée**.
- La différence est que la **LDC** contient non seulement un **pointeur** vers l'**élément suivant** mais aussi un pointeur vers l'**élément précédent**.
- Une **LDC** est une liste dont chaque élément peut accéder à l'aide de pointeurs aux éléments positionnés **avant** et **après** lui dans la liste.



# Listes doublement chaînées

## Définition

- Les **listes doublement chaînées** sont constituées d'éléments comportant trois composants :
  - un **champ** contenant les données
  - un **pointeur** vers **l'élément suivant** de la liste.
  - un **pointeur** vers **l'élément précédent** de la liste.




- Les listes doublement chaînées peuvent donc être parcourues dans les deux sens.

# Listes doublement chaînées

## Construction d'une LDC

- En langage C, une **liste doublement chaînée** peut être implémentée sous forme d'une **structure** : chaque **élément** de la liste est une **structure**.
- **Un élément de la liste :**
  - Pour les exemples, nous allons créer une liste chaînée de nombres entiers.
  - Chaque **élément** de la liste aura la forme de la **structure** suivante :

```
typedef struct element element;
 struct element
     {
         int val;
         element *nxt;
         element *prev;
     };
```

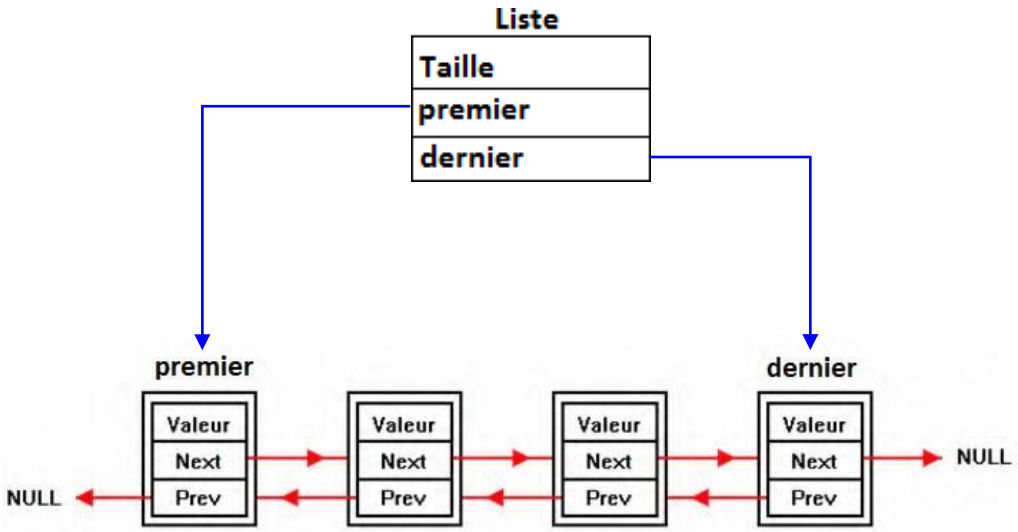
- **next** : pointe vers l'élément **suivant** (ou **NULL** s'il s'agit du **dernier élément**)
- **prev** : pointe vers l'élément **précédent** (ou **NULL** s'il s'agit du **premier élément**)

# Listes doublement chaînées

## Construction d'une LDC

- Pour faciliter la représentation d'une liste doublement chaînée, on utilise une deuxième **structure** qu'on appelle **Liste**.
- L'objectif de cette deuxième **structure** est de **contrôler** le **premier** et **dernier** élément de la liste.

```
typedef struct element element;
 struct element
      {
           int val;
           element *nxt;
           element *prev;
      };

typedef struct Liste Liste;
struct Liste
     {
          int taille;
          element *premier;
          element *dernier;
     };
```

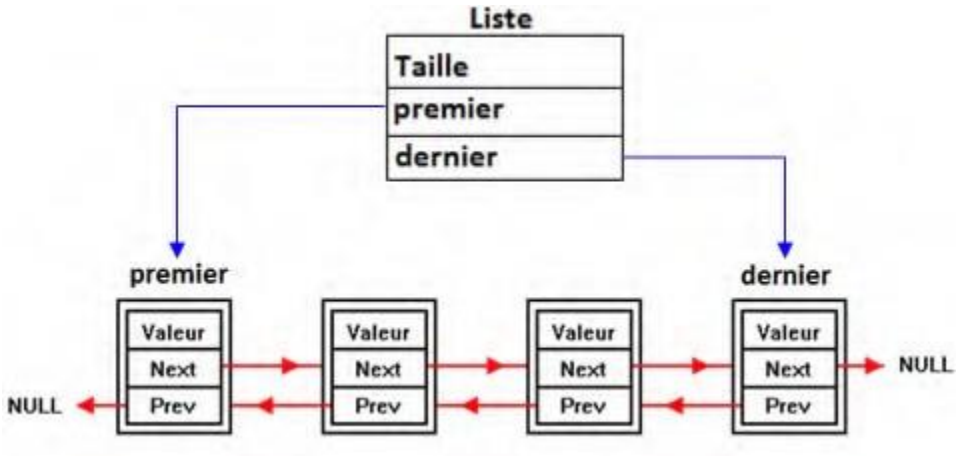


# Listes doublement chaînées

## Construction d'une LDC

- **Créer une liste chaînée (vide) :**
  - Il est important de toujours initialiser la liste chaînée à NULL.
  - La fonction de création d'une LDC vide aura la forme suivante :

```
Liste *initialise() {
 Liste *dliste = malloc(sizeof *dliste);

 dliste->taille = 0;
 dliste->premier = NULL;
 dliste->dernier = NULL;

return dliste;
}
```

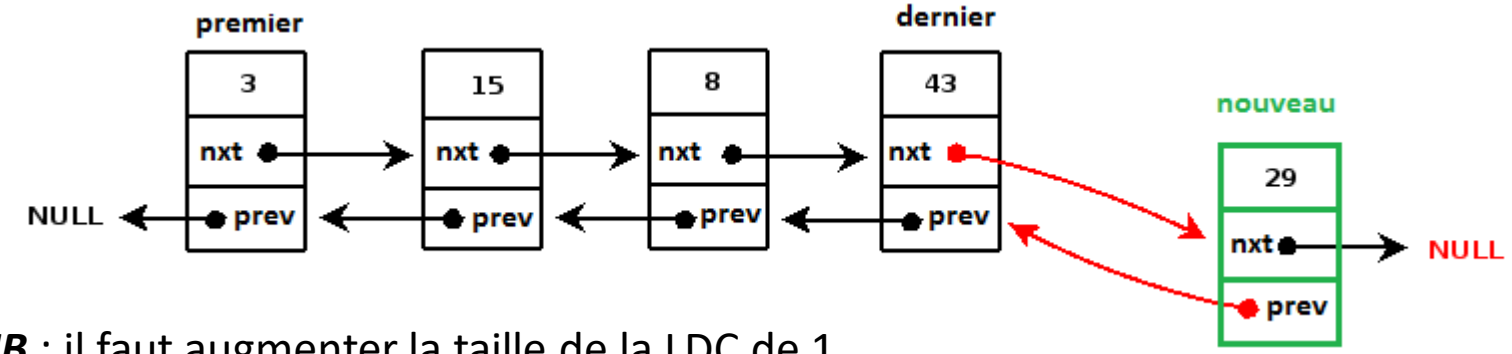



  - Dans cette fonction, la liste doublement chaînée est appelée **dliste** (double liste).

# Listes doublement chaînées

## Ajouter un élément

- **Ajouter un élément en fin de liste**
  - Si la liste n'est pas vide :
    - créer un **nouvel** élément
    - lui attribuer sa valeur
    - rattacher le **dernier** élément de la liste au **nouvel** élément
    - pointer **nouveau->prev** vers le **dernier** élément de la liste
    - faire pointer **nouveau->next** vers **NULL**
  - Sinon, pointer **premier** et **dernier** de la liste vers le **nouvel élément**


  - ***NB*** : il faut augmenter la taille de la LDC de 1.

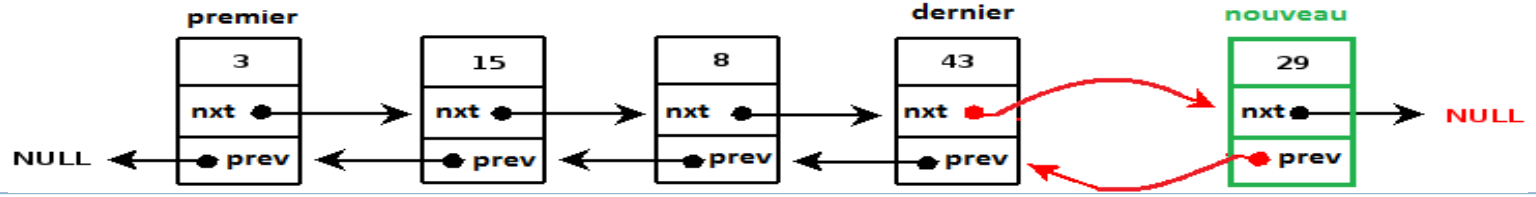
# Listes doublement chaînées

## Ajouter un élément

- La fonction d'ajout d'un élément en fin de la liste est la suivante :

```
Liste *ajouterEnFin(Liste *dliste, int valeur){
    if (dliste != NULL)
    {   element *nouveau = malloc(sizeof *nouveau);
        nouveau->val = valeur;
        nouveau->nxt = NULL;
        if (dliste->dernier == NULL) /*  pointer la tête et la fin vers le nouvel élément */
            {   nouveau->prev = NULL;
                dliste->premier = nouveau;
                dliste->dernier = nouveau;
            }
            else /* Cas où des éléments sont déjà présents dans notre liste */
            {   dliste->dernier->nxt = nouveau;
                nouveau->prev = dliste->dernier;
                dliste->dernier = nouveau;
            }
            dliste->taille++; /* Incrémentation de la taille de la liste */
        }
    return dliste; /* on retourne notre nouvelle liste */
}
```

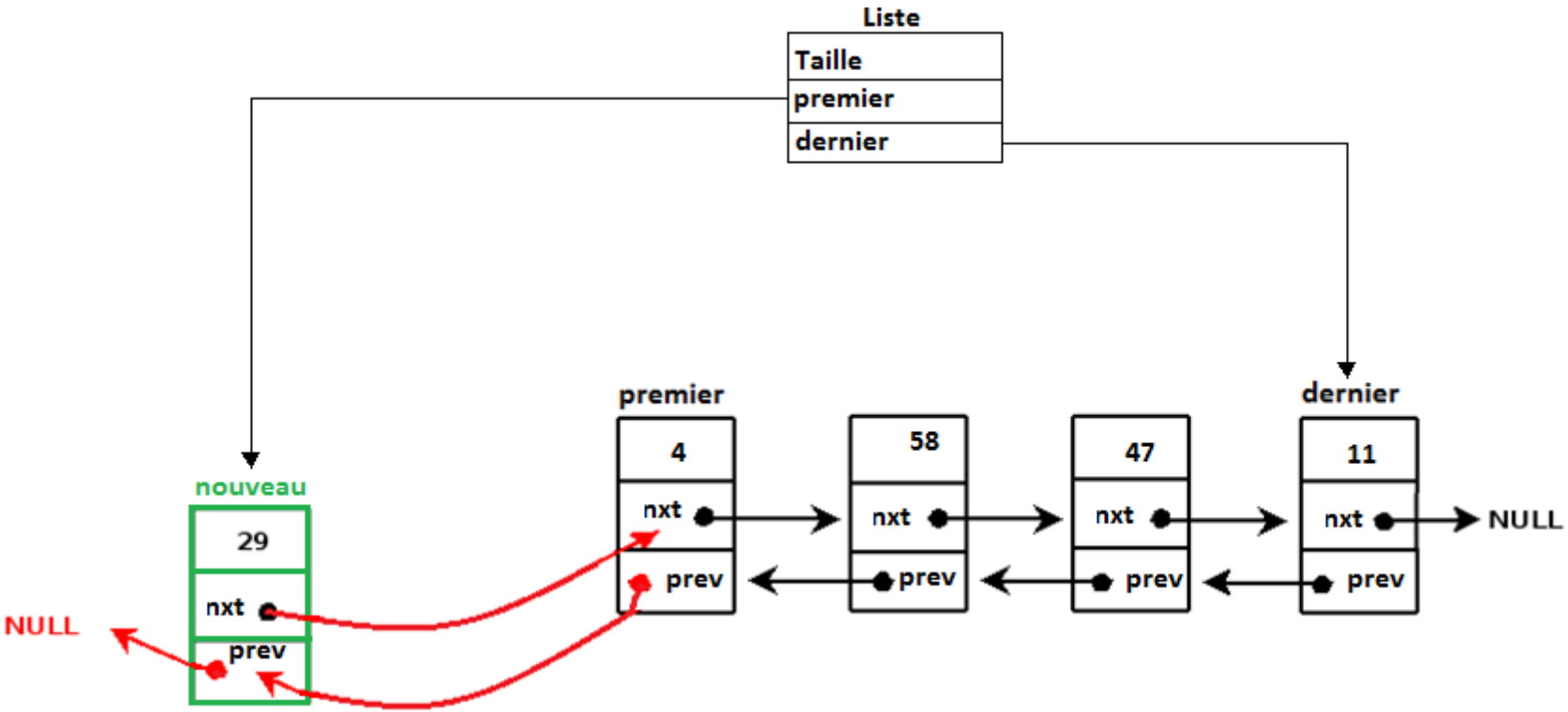
| Liste |
|---|
| Taille |
| premier |
| dernier |



# Listes doublement chaînées

## Ajouter un élément

- **Ajouter un élément en tête de liste**
  - L'ajouter d'un élément en début de liste est similaire à l'ajout en fin de liste.

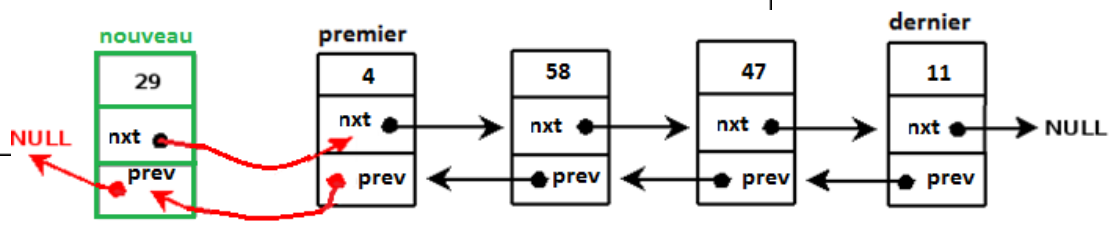


# Listes doublement chaînées

## Ajouter un élément

- La fonction d'ajout d'un élément en tête de la liste est la suivante :

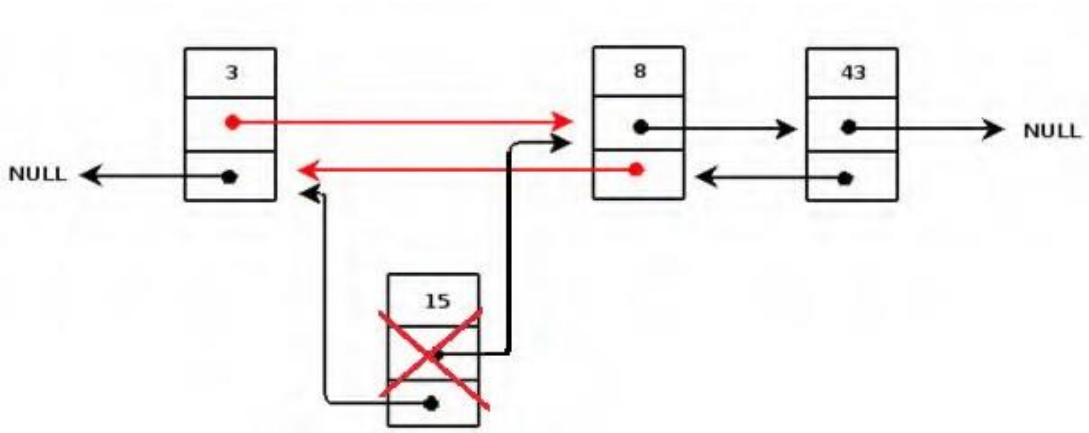
```
Liste *ajouterEnTete(Liste *dliste, int valeur) {
    if (dliste != NULL)
        {   element *nouveau = malloc(sizeof *nouveau);
            nouveau->val = valeur;
            nouveau->prev = NULL;
            if (dliste->dernier == NULL)
                {   nouveau->nxt = NULL;
                    dliste->premier = nouveau;
                    dliste->dernier = nouveau;
                }
            else
                {   dliste->premier->prev = nouveau;
                    nouveau->nxt = dliste->premier;
                    dliste->premier = nouveau;
                }
        dliste->taille++;
    }
return dliste;
}
```



# Listes doublement chaînées

## Supprimer un élément en fonction de sa valeur

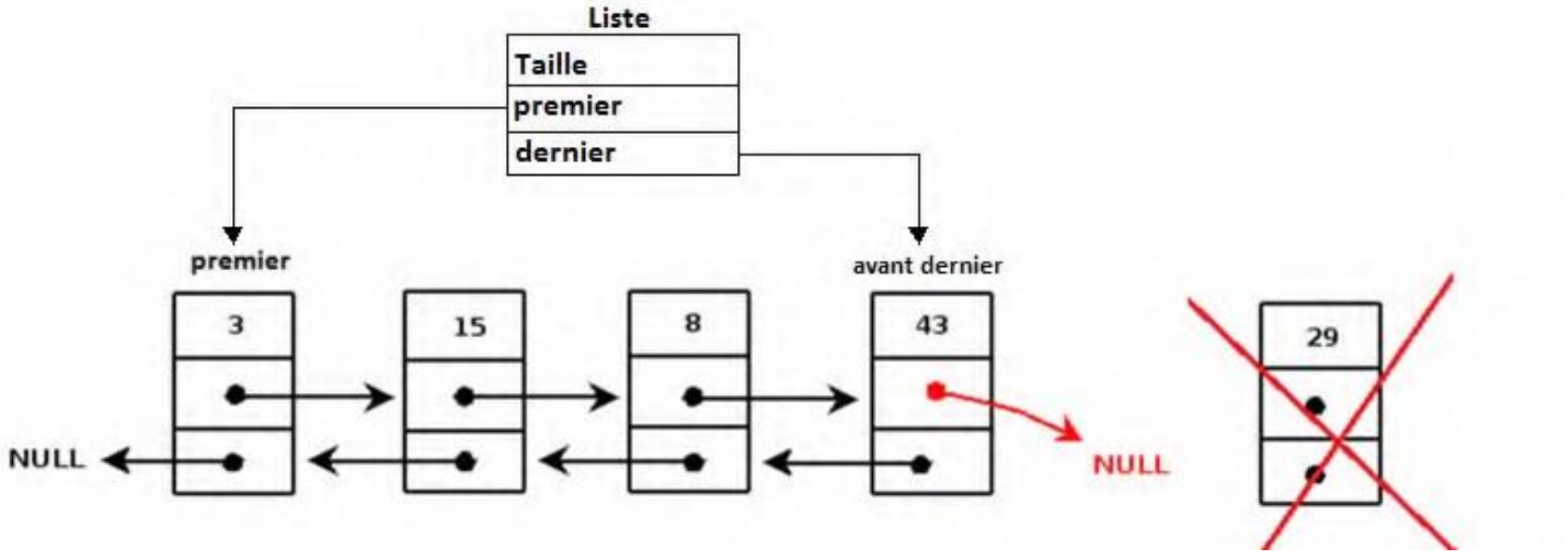
- Pour supprimer un élément en fonction de sa valeur, il faut parcourir la liste et rechercher l'élément à supprimer.
- Dès que l'on aura trouver la valeur correspondante, trois possibilités sont à traiter :
  - l'élément se trouve en **fin** de liste.
  - l'élément se trouve en **début** de liste.
  - l'élément se trouve en **milieu** de liste.



# Listes doublement chaînées

## Supprimer un élément en fonction de sa valeur

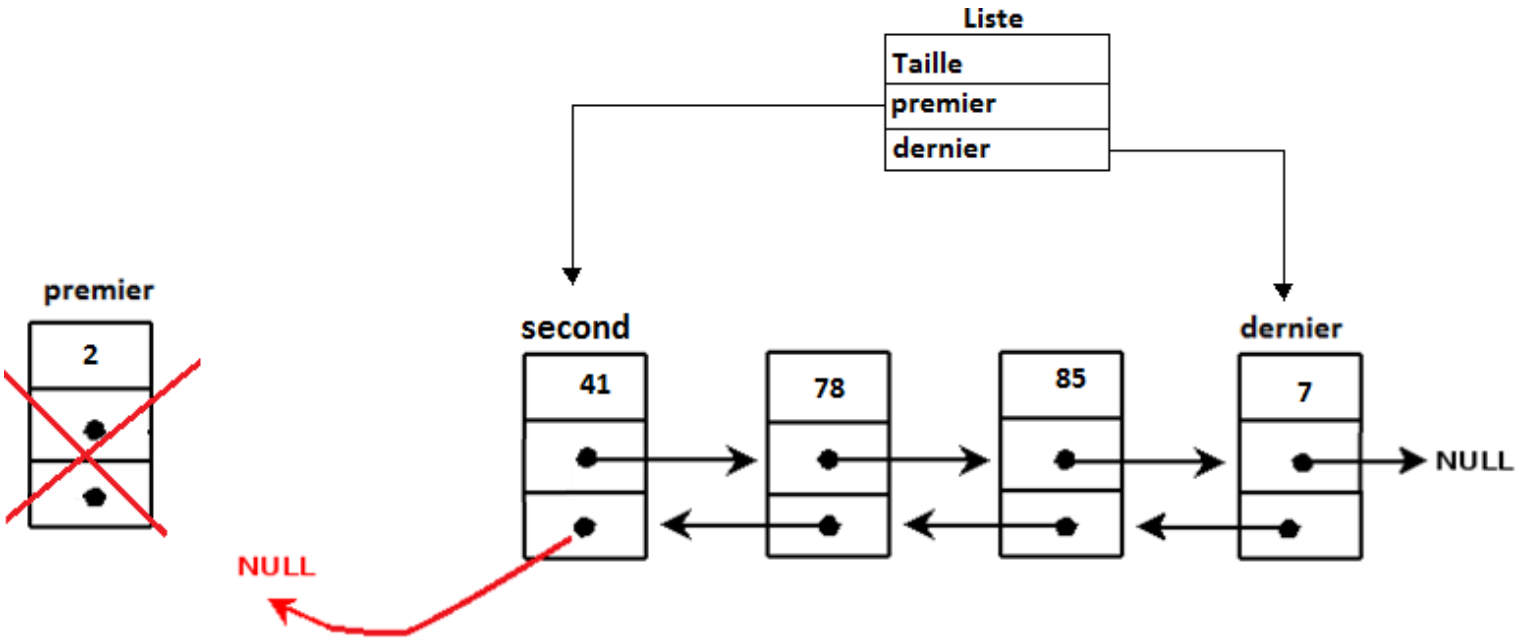
- **L'élément se trouve en fin de liste.**
  - faire pointer **dernier** de la **Liste** vers l'**avant dernier** élément.
  - faire pointer le pointeur **nxt** de l'**avant dernier** élément vers **NULL**.
  - supprimer le **dernier** élément.



# Listes doublement chaînées

## Supprimer un élément en fonction de sa valeur

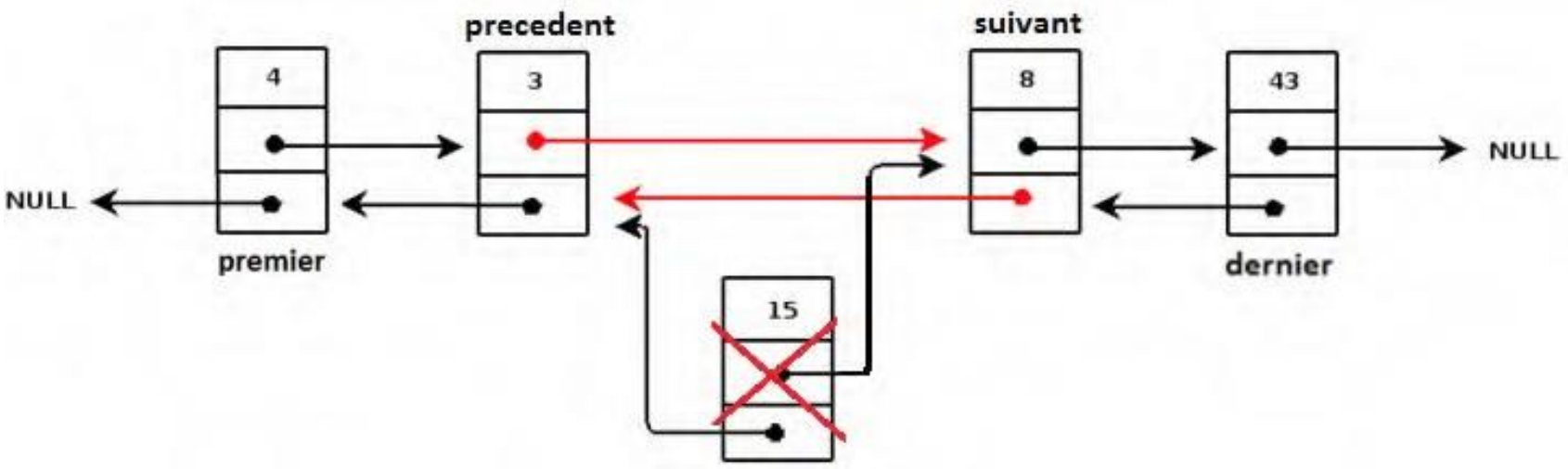
- **L'élément se trouve en début de liste :**
  - faire pointer **premier** de la **Liste** vers le **second** élément.
  - faire pointer **prev** du **second** élément vers **NULL**.
  - supprimer le **premier** élément.



# Listes doublement chaînées

## Supprimer un élément en fonction de sa valeur

- **L'élément se trouve en milieu de liste :**
  - relier l'élément **précédent** à l'élément que l'on veut supprimer vers l'élément **suivant** à l'élément que l'on veut supprimer
  - relier l'élément **suivant** à l'élément que l'on veut supprimer vers l'élément **précédent** à l'élément que l'on veut supprimer.
  - supprimer l'élément que l'on veut supprimer .

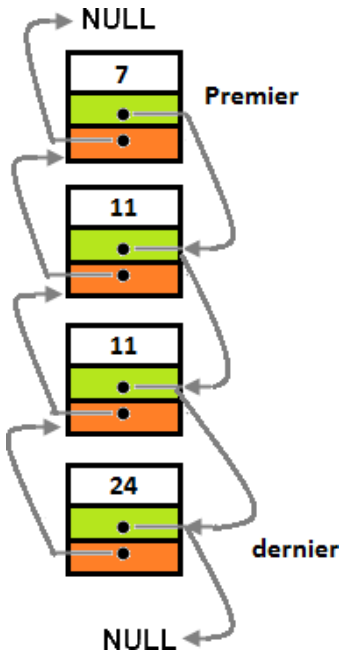


# Listes doublement chaînées

## Supprimer un élément en fonction de sa valeur

- **La fonction permettant de supprimer un élément en fonction de sa valeur :**

```
Liste *supprimer_element(Liste *dliste, int valeur){
   if (dliste != NULL)
      {   element *temp = dliste->premier; int found = 0;
          while (temp != NULL && !found)
              { if (temp->val == valeur)
                      { if (temp->nxt == NULL) // On est à la fin de la liste
                          {   dliste->dernier = temp->prev;
                              dliste->dernier->nxt = NULL;
                          }
                        else if (temp->prev == NULL) //On est au début de la liste
                          {   dliste->premier = temp->nxt;
                              dliste->premier->prev = NULL;
                          }
                        else  // On est au milieu de la liste
                          {   temp->nxt->prev = temp->prev;
                              temp->prev->nxt = temp->nxt;
                          }
                        free(temp);
                        dliste->taille--;
                        found = 1; //pour s'arrêter au premier élément trouvé
                      }
                else  { temp = temp->nxt; }
              }
      }
   return dliste;
```




- ***NB*** : La fonction ne supprimera que le premier élément trouvé.