CHAPITRE 1 : LES LISTES CHAÎNÉES

Une structure de données est un format spécial pour l'organisation et le stockage des données. Les types de structure de données générales comprennent les tableaux, les fichiers, les listes, piles, files d'attente, arbres, graphiques, etc.

Une liste chaînée est une structure de données utilisée pour stocker des collections de données. Une liste chaînée possède les caractéristiques suivantes.

- Les éléments successifs sont reliés par des pointeurs ;
- Le dernier élément indique NULL ;
- Peut augmenter ou diminuer en taille pendant l'exécution d'un programme ;
- Peut être rallongée aussi longtemps que nécessaire (jusqu'à épuisement de la mémoire des systèmes)
- Ne gaspille pas d'espace mémoire (mais prend un peu de mémoire supplémentaire pour les pointeurs). La mémoire est allouée au fur et à mesure que la liste s'allonge.

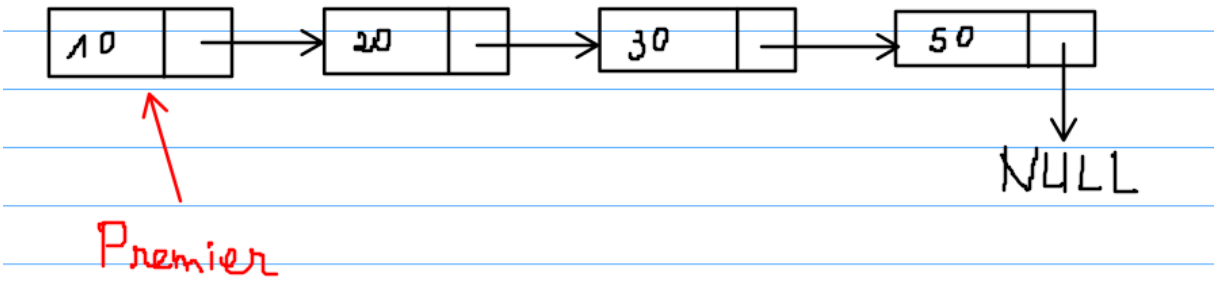

Figure 1 : Exemple d'une liste chaînée

Les opérations suivantes sont possibles sur les listes chaînées :

- Principales opérations sur les listes chaînées :
  - Insérer : insérer un élément dans la liste ;
  - Supprimer : supprime et renvoie l'élément à position spécifiée dans la liste.
- Opérations auxiliaires sur les listes chaînées
  - Supprimer la liste : supprime tous les éléments de la liste (élimine la liste) ;
  - Compter : renvoie le nombre d'éléments de la liste ;
  - Trouver le n-ième nœud à partir de la fin de la liste.

Il existe de nombreuses autres structures de données qui font la même chose que les listes chaînées. Avant de parler plus des listes chaînées, il est important de comprendre la différence entre les listes chaînées et les tableaux. Les deux sont utilisés pour stocker des collections de données, et comme les deux sont utilisés dans le même but, nous devons différencier leur utilisation. Cela signifie trouver dans quels cas les tableaux sont adaptés et dans quels cas les listes chaînées sont appropriées.

APERÇU DES TABLEAUX

Un bloc de mémoire est alloué pour l'ensemble du tableau afin de contenir tous les éléments du tableau. Les éléments du tableau peuvent être accessibles à tout moment en utilisant l'index de l'élément particulier à atteindre.

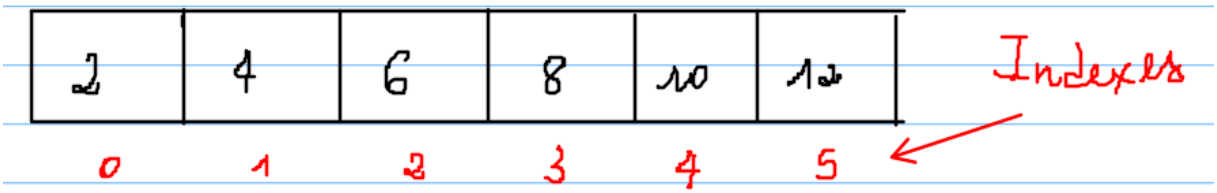

Figure 2 : Exemple d'un tableau avec des index

Pourquoi un accès constant aux éléments du tableau ?

Pour accéder à un élément de tableau, l'adresse d'un élément est calculée comme un décalage par rapport à l'adresse de base du tableau et une multiplication est nécessaire pour calculer ce qui est censé être ajouté à l'adresse de base pour obtenir l'adresse mémoire de l'élément. D'abord la taille d'un élément du type de cette donnée est calculée et ensuite multipliée par l'indice de l'élément pour obtenir la valeur à ajouter à l'adresse de base.

Ce processus nécessite une multiplication et une addition. Comme ces deux opérations peuvent être exécutées à tout moment, on peut dire que l'accès au réseau peut être effectué en tout temps.

AVANTAGES DES TABLEAUX

- Simple et facile à utiliser
- Accès plus rapide aux éléments (accès permanent)

INCONVÉNIENTS DES TABLEAUX

- **Pré-allocation** de toute la mémoire nécessaire au départ et gaspillage d'espace mémoire pour les index dans le tableau qui sont vides.
- **Taille fixe** : La taille du tableau est statique (préciser la taille du tableau avant de l'utiliser).
- **Allocation d'un bloc** : Pour allouer de l'espace au tableau au début, il peut parfois être impossible d'obtenir suffisamment la mémoire pour le tableau au complet (si la taille du tableau est grande).
- **Insertion complexe basée sur la position** : Pour insérer un élément à une position donnée, on peut avoir besoin de déplacer les éléments existants. Cela nous permettra d'insérer le nouvel élément à l'endroit souhaité. Si la position à laquelle nous voulons ajouter un élément est au début (en première position), alors, l'opération de déplacement est plus coûteuse (en temps d'exécution, vu que, évidemment, il faudra déplacer tous les éléments du tableau).

AVANTAGES DES LISTES CHAÎNÉES

Les listes chaînées présentent à la fois des avantages et des inconvénients. L'avantage des listes chaînées est qu'elles peuvent être élargi en permanence. Pour créer un tableau, nous devons allouer de la mémoire pour un certain nombre d'éléments. Pour ajouter d'autres éléments au tableau lorsqu'il est plein, nous devons créer un nouveau tableau et copier l'ancien tableau dans le nouveau. Cela peut prendre beaucoup de temps. Nous pouvons éviter cela en allouant beaucoup d'espace au départ, mais le fait est que nous pourrions en allouer plus que ce dont nous aurons réellement besoin et ainsi, gaspiller la mémoire. Avec une liste chaînée, nous pouvons commencer avec un espace pour un seul élément alloué et ajouter de nouveaux éléments facilement sans avoir à faire de copie ni de réaffectation.

PROBLÈMES LIES AUX LISTES CHAÎNÉES (INCONVÉNIENTS)

Les listes chaînées posent un certain nombre de problèmes. Le principal inconvénient des listes chaînées est le temps d'accès aux éléments un à un. Le tableau est à accès aléatoire, ce qui signifie qu'il faut une complexité en O(1) pour accéder à tout élément dans le pire des cas. Les listes chaînées prennent O(n) en complexité pour l'accès à un élément de la liste dans le pire des cas.

Un autre avantage des tableaux dans le temps d'accès aux éléments, est la localisation spatiale en mémoire. Les tableaux sont définis comme des blocs de mémoire contigus, et donc tout élément de tableau sera physiquement proche de ses voisins. Ceci est un avantage indéniable dans les méthodes modernes de mise en cache des CPU.

Bien que l'allocation dynamique du stockage soit un grand avantage, les frais généraux liés au stockage et à l'extraction de données peuvent faire une grande différence. Parfois, les listes chaînées sont difficiles à manipuler. Si le dernier élément est supprimé, l'avant-dernier doit alors faire changer son pointeur pour contenir une référence sur NULL. Pour cela, il faut parcourir la liste pour trouver l'avant-dernier élément pour le faire pointer sur NULL.

Au final, les listes chaînées gaspillent de la mémoire en termes de points de référence supplémentaires.

Tableau 1 : Complexités LC & Tableaux

Paramètres	**Listes chaînées**	**Tableau**	**Tableau dynamique**
Indexage	O(n)	O(1)	O(1)
Insertion / Suppression au début	O(1)	O(n),*	O(n)
Insertion en fin	O(n)	O(1)*	O(1)* O(n)**
Suppression en fin	O(n)	O(1)	O(n)
Insertion / Suppression au milieu	O(n)	O(n)*	O(n)
Espace gaspillé	O(n)***	0	O(n)

*si le tableau n'est pas plein, **si le tableau est plein, ***à cause des pointeurs.

N.B. : En général, on entend par "liste chaînée" une liste simplement chaînée. Cette liste se compose d'un certain nombre de nœuds dans lesquels chaque nœud est suivi d'un pointeur

vers l'élément suivant. Le lien du dernier nœud de la liste est NULL, qui indique la fin de la liste.

Vous trouverez ci-dessous une déclaration de type pour une liste chaînée d'entiers :

```c
struct ListeNoeuds {
    int data;
    struct ListeNoeuds *suivant;
};
```

Figure 3 : Exemple de déclaration d'une LC

OPÉRATIONS DE BASE SUR UNE LISTE

- Parcourir la liste
- Insérer un élément dans la liste
- Suppression d'un élément de la liste

PARCOURIR LA LISTE CHAÎNÉE

Supposons que la tête pointe vers le premier nœud de la liste. Pour parcourir la liste, nous suivons les étapes suivantes :

- Faire suivre les pointeurs ;
- Afficher le contenu des nœuds (ou les compter) au fur et à mesure qu'ils sont parcourus ;
- S'arrêter lorsque le pointeur suivant indique NULL.

La fonction LongueurListe() prend en entrée une liste chaînée et compte le nombre de nœuds de la liste. La fonction donnée ci-dessous peut être utilisée pour afficher les données de la liste avec une fonction d'affichage supplémentaire.

```c
int LongueurListe(struct ListeNoeuds *tete) {
    struct ListeNoeuds *courant = tete;
    int compteur = 0;

    while(courant != NULL) {
        compteur++;
        courant = courant->suivant;
    }
    return compteur;
}
```

Figure 4 : Exemple de codes C de parcours d'une LC

Complexité du temps : O(n), pour parcourir la liste de la taille n.
Complexité spatiale : O(1), pour créer une variable temporaire.

INSERTION DANS LA LISTE CHAÎNÉE

Nous avons 3 cas possibles :

- Insertion d'un nouveau nœud en tête (au début)
- Insertion d'un nouveau nœud en queue (à la fin de la liste)
- Insertion d'un nouveau nœud au milieu de la liste (emplacement aléatoire)

N.B. : Pour insérer un élément dans la liste chaînée à une position p quelconque, supposons qu'après avoir inséré l'élément, la position de ce nouveau nœud est p.

INSERTION EN TÊTE

Dans ce cas, un nouveau nœud est inséré avant le nœud principal actuel. Un seul pointeur suivant doit être modifié (le pointeur du nouveau nœud) et cela peut se faire en deux étapes :

- Mettre à jour le pointeur suivant du nouveau nœud, pour pointer sur la tête actuelle ;

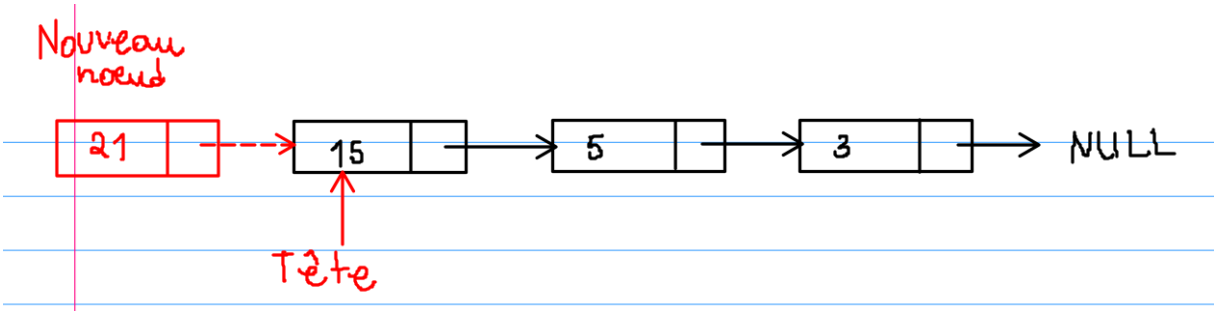

Figure 5 : Insertion en Tête 1/2

- Mettez à jour le pointeur de tête pour qu'il pointe vers le nouveau nœud.

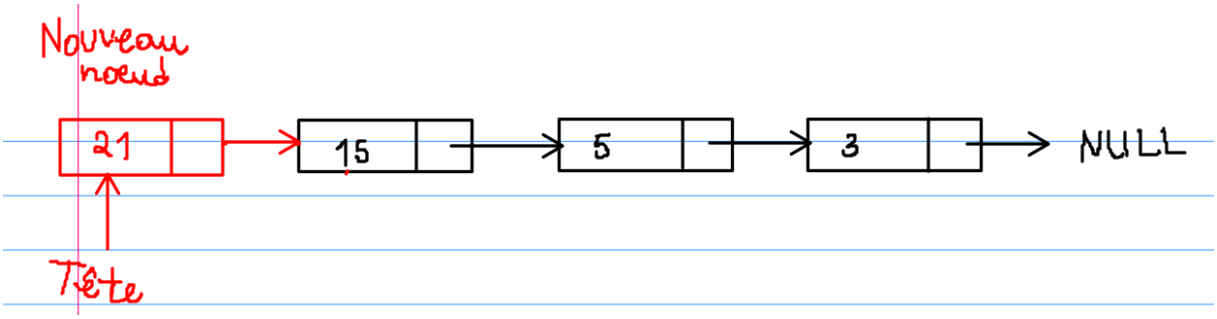

Figure 6 : Insertion en Tête 2/2

INSERTION EN QUEUE (EN FIN)

Dans ce cas, nous devons modifier deux pointeurs suivants (le pointeur du dernier nœud et le pointeur suivant du nouveau nœud).

- Le pointeur du nouveau nœud pointe vers NULL ;

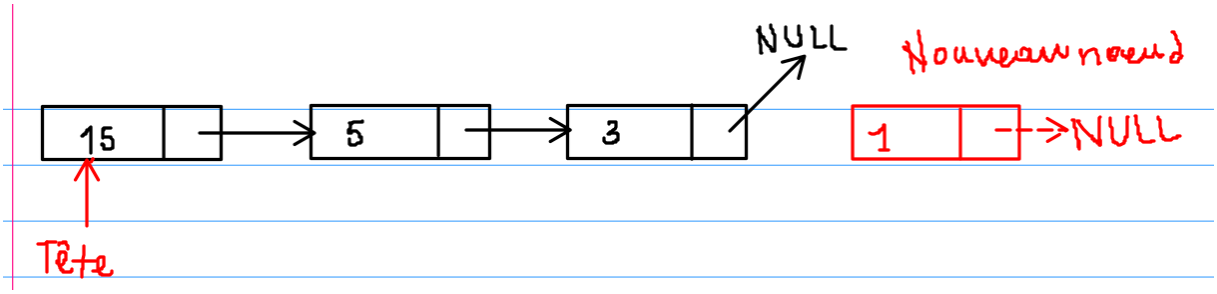

Figure 7 : Insertion en Queue 1/2

- Le pointeur du dernier pointe sur le nouveau nœud.

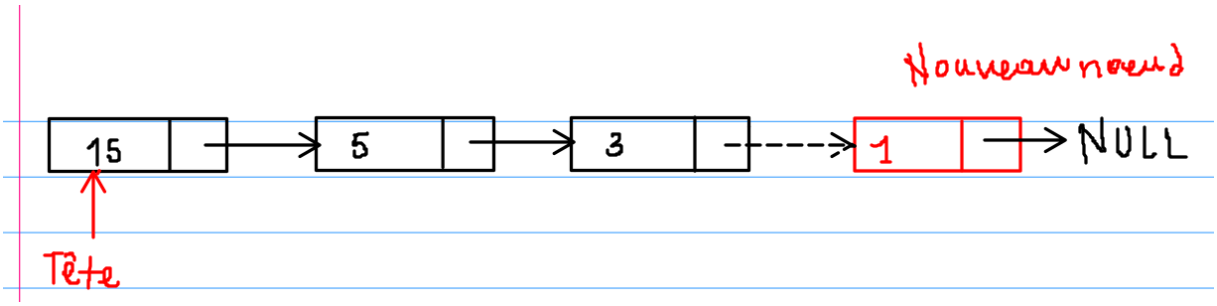

Figure 8 : Insertion en Queue 2/2

INSERTION AU MILIEU

Supposons que l'on nous donne une position où nous voulons insérer le nouveau nœud. Dans ce cas, nous devons également modifier deux points suivants :

- Si nous voulons ajouter un élément à la position 3, alors nous nous arrêtons à la position 2. Cela signifie que nous devons traverser 2 nœuds et insérer le nouveau nœud. Par souci de simplicité, supposons que le deuxième nœud est appelé **nœud de position**. Le nouveau nœud pointe vers le nœud suivant de la position où nous voulons ajouter ce nœud.

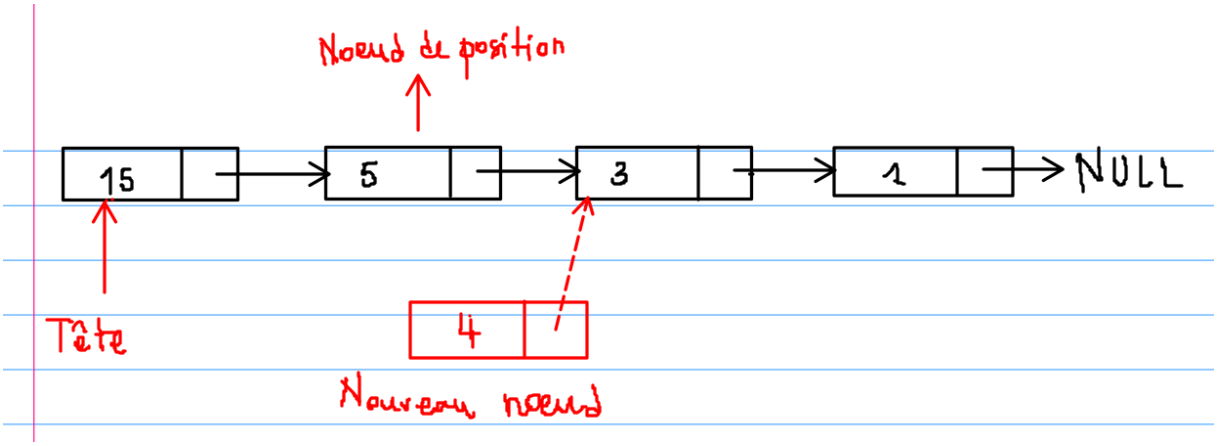

Figure 9 : Insertion en position p 1/2

- Le pointeur du nœud de position pointe maintenant sur le nouveau nœud.

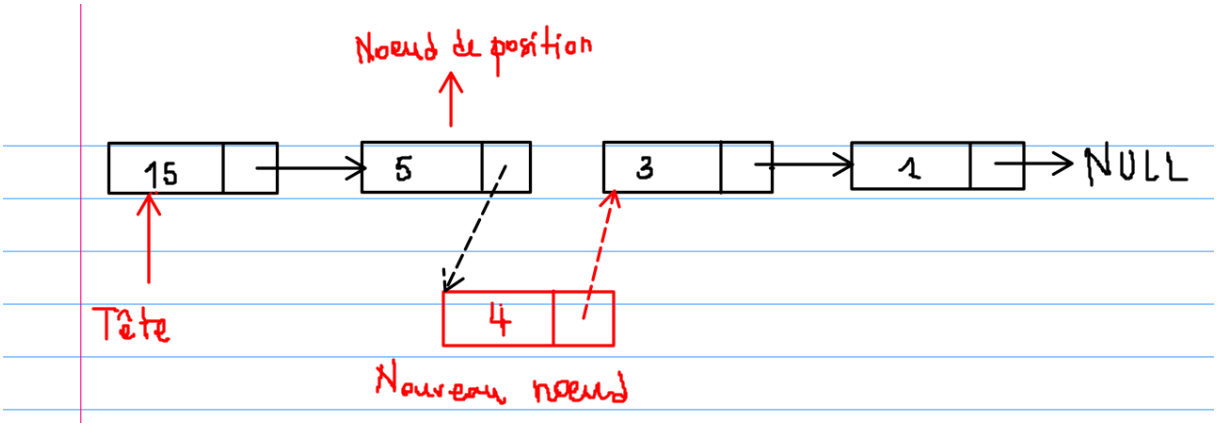

Figure 10 : Insertion en position p 2/2

Écrivons le code pour les trois cas vus plus haut. Nous devons mettre à jour le premier pointeur d'élément dans la fonction appelante, et pas seulement dans la fonction appelée. C'est pourquoi nous devons envoyer un double pointeur. Le code suivant insère un nœud dans la liste chaînée simples.

```c
void insererListe(struct ListeNoeuds *tete, int donnee, int position) {
    int k = 1;
    struct ListeNoeuds *p, *q, *nouveauNoeud;
    nouveauNoeud = (ListeNoeuds*)malloc(sizeof(struct LiisteNoeuds));
    nouveauNoeud -> data = donnee;
    p = *tete;

    //Insertion au début
    if(position == 1) {
        nouveauNoeud -> suivant = p;
        *tete = nouveauNoeud;
    }
    else {
        // On parcourt la liste jusqu'à la position désirée
        while((p != NULL) && (k < position)) {
            k++;
            q = p;
            p = p->suivant;
        }
        q->suivant = nouveauNoeud;
        nouveauNoeud -> suivant = p;
    }
}
```

Figure 11 : Exemple de codes C d'insertions dans une LC

SUPPRESSION DANS UNE LISTE CHAÎNÉE

Comme pour l'insertion, nous avons ici aussi trois cas.

- Suppression du premier nœud
- Supprimer le dernier nœud
- Suppression d'un nœud intermédiaire.

SUPPRESSION DU PREMIER NŒUD DE LA LISTE CHAÎNÉE

Le premier nœud (nœud de tête actuel) est retiré de la liste. Cela peut se faire en deux étapes :

- Créer un nœud temporaire qui pointera vers le même nœud que celui de tête ;

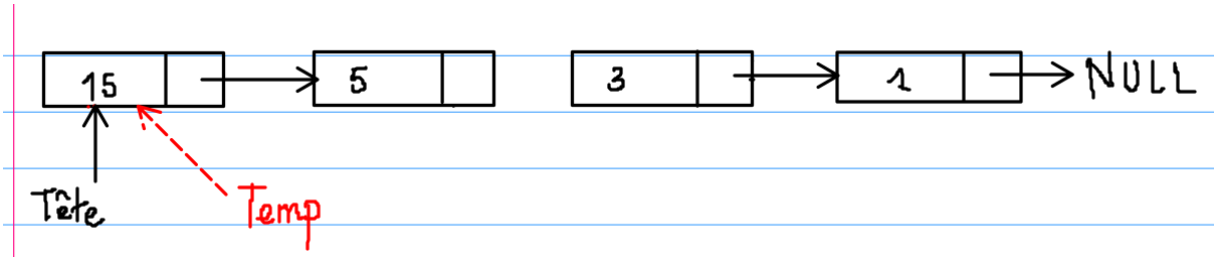

Figure 12 : Suppression en Tête 1/2

- Maintenant, déplacez le pointeur des nœuds de tête vers le nœud suivant et éliminez les nœuds.

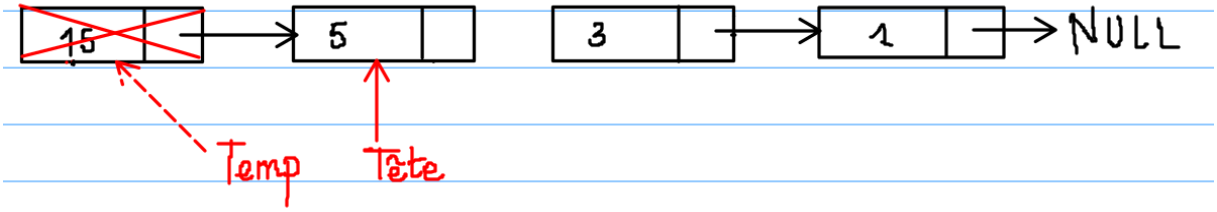

Figure 13 : Suppression en Tête 2/2

SUPPRESSION EN QUEUE

Dans ce cas, le dernier nœud est supprimé de la liste. Cette opération est un peu plus délicate que la suppression du premier nœud, parce que l'algorithme devrait trouver un nœud, qui est antérieur à la queue (l'avant dernier nœud). Cela peut se faire en trois étapes :

- Parcourir la liste et, pendant la traversée, conserver l'adresse du nœud précédent également. Quand nous arriverons à la fin de la liste, nous aurons deux pointeurs, l'un pointant sur le dernier et l'autre pointant vers le nœud situé avant le nœud de queue ;

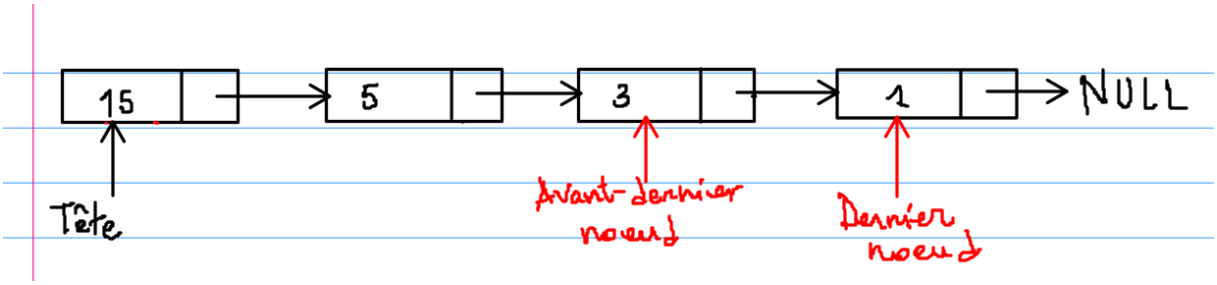

Figure 14 : Suppression en Queue 1/3

- Mettre à jour le pointeur suivant de l'avant dernier nœud avec NULL ;

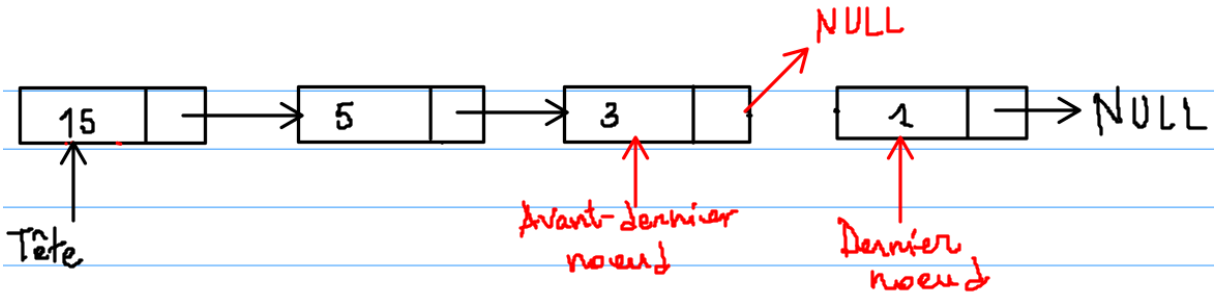

Figure 15 : Suppression en Queue 2/3

- Se débarrasser du nœud de queue.

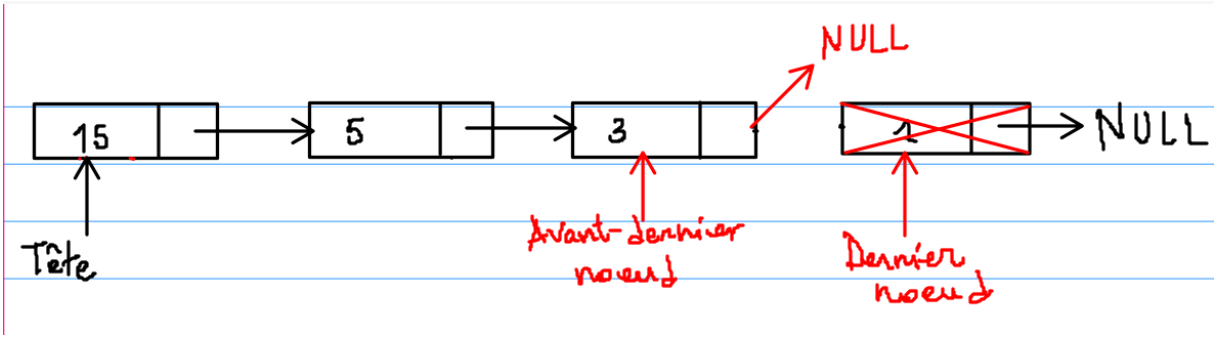

Figure 16 : Suppression en Queue 3/3

SUPPRESSION AU MILIEU

Dans ce cas, le nœud à supprimer est *toujours situé entre deux nœuds*. Les liens de tête et de queue ne sont pas mis à jour dans ce cas. Une telle suppression peut être effectuée en deux étapes :

- Comme dans le cas précédent, maintenez le nœud précédent tout en parcourant la liste. Une fois que nous avons trouvé le nœud à supprimer, changez le pointeur suivant du nœud précédent pour le pointeur suivant du nœud à supprimer ;

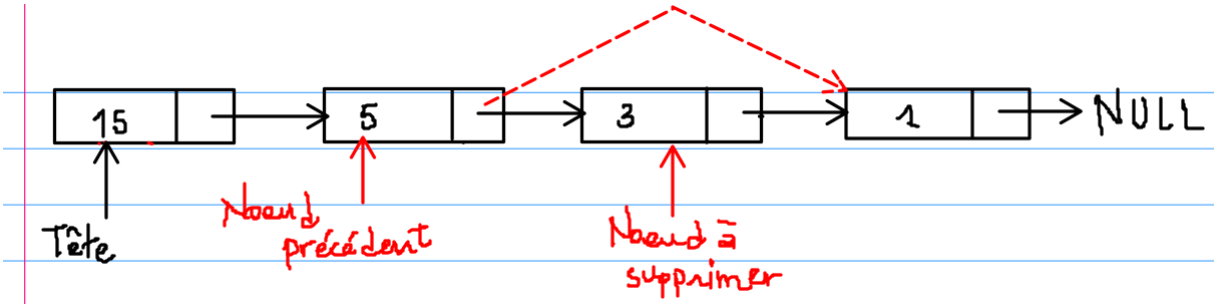

Figure 17 : Suppression en position p 1/2

- Se débarrasser du nœud courant à supprimer.

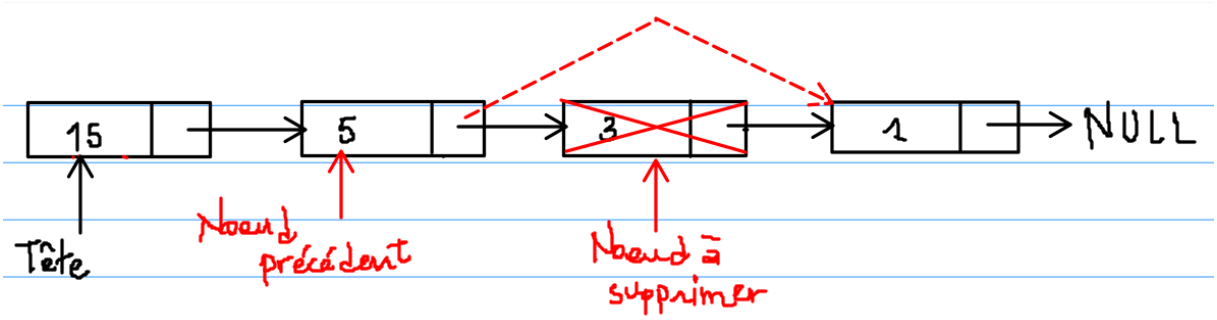

Figure 18 : Suppression en position 2/2

```c
void supprimerListe(struct ListeNoeuds *tete, int position) {
    int k = 1;
    struct ListeNoeuds *p, *q;

    if(*head == NULL) {
        printf("La liste est vide !");
        return;
    }
    p = *tete;

    //Au début
    if(position == 1) {
        *tete = (*tete) -> suivant;
        free(p);
        return;
    }
    else {
        //Parcourir la liste jusqu'au noeud à supprimer
        while((p != NULL) && (k < position)) {
            k++;
            q = p;
            p = p->suivant;
        }
        if(p == NULL) //Si on arrive à la fin de la liste sans trouver la position à atteindre
            printf("La position n'existe pas !");
        else {  //Au milieu
            q -> suivant = p -> suivant;
            free(p);
        }
    }
}
```

Figure 19 : Exemple de codes C de suppression dans une LC

SUPPRIMER LA LISTE CHAÎNÉE

Cela fonctionne en stockant le nœud actuel dans une variable temporaire et en libérant le nœud actuel. Après avoir libéré le nœud courant, passez au nœud suivant avec une variable temporaire et répétez ceci pour tous les nœuds.

```c
void supprimerToutLaListe(struct ListeNoeuds *tete) {
    struct ListeNoeuds *noeudAuxilliaire, *iterateur;
    iterateur = *tete;

    while(iterateur) {
        noeudAuxilliaire = iterateur -> suivant;
        free(iterateur);
        iterateur = noeudAuxilliaire;
    }
    *tete = NULLL;  //Pour modifier la vraie tête dans le paramètre
}
```

Figure 20 : Exemple de codes C de suppression d'une LC