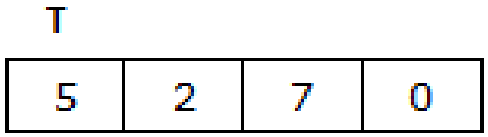
Les listes simplement chainées

Aziza EL OUAAZIZI
Cours SMIA S4
Faculté Polydisciplinaire de Taza
Univérsité Sidi Mohammed Ben Abdellah

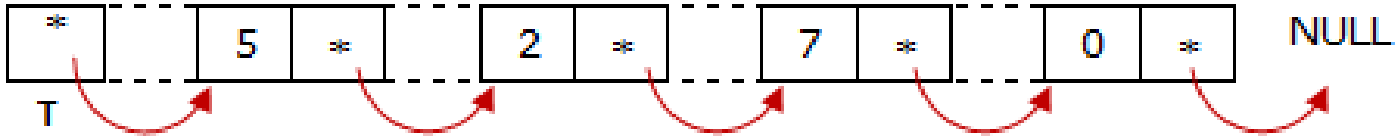
Introduction

Pour créer un conteneur (variable) de plusieurs valeurs de même types soit, on peut utiliser les structure de données:

- tableaux: les éléments de celui-ci sont placés de façon contiguë en mémoire.



- Listes chainées: les éléments sont répartis dans différentes zones mémoire et reliés entre eux par des pointeurs.



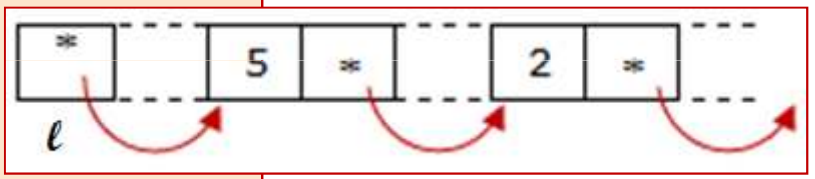
- Les éléments peuvent être de n'importe quel type d'éléments : entiers, caractères, structures, tableaux, voir même d'autres listes chaînées...

Déclaration en C d'une liste chaînée

Chaque élément d'une liste chaînée est composé de deux parties :

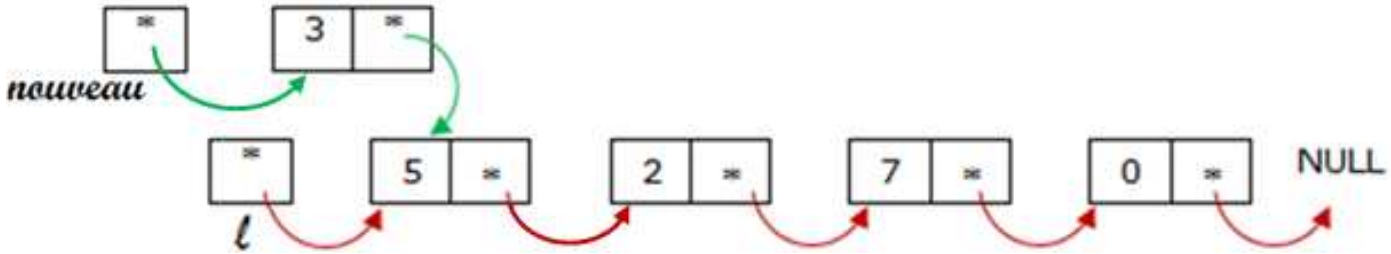
- la valeur que vous voulez stocker,
- l'adresse de l'élément suivant, s'il existe. S'il n'y a plus d'élément suivant, alors l'adresse sera NULL, et désignera le bout de la chaîne.

```
struct element
{
    int valeur;
    struct element *suivant;
};

/*-Définition du type element-*/
typedef struct element element;

/*-Définition du type liste-*/
typedef element* liste;

/*-Déclaration d'une liste-*/
liste l=NULL; //ou element* l=NULL;
```



Ajouter un élément en tête de liste

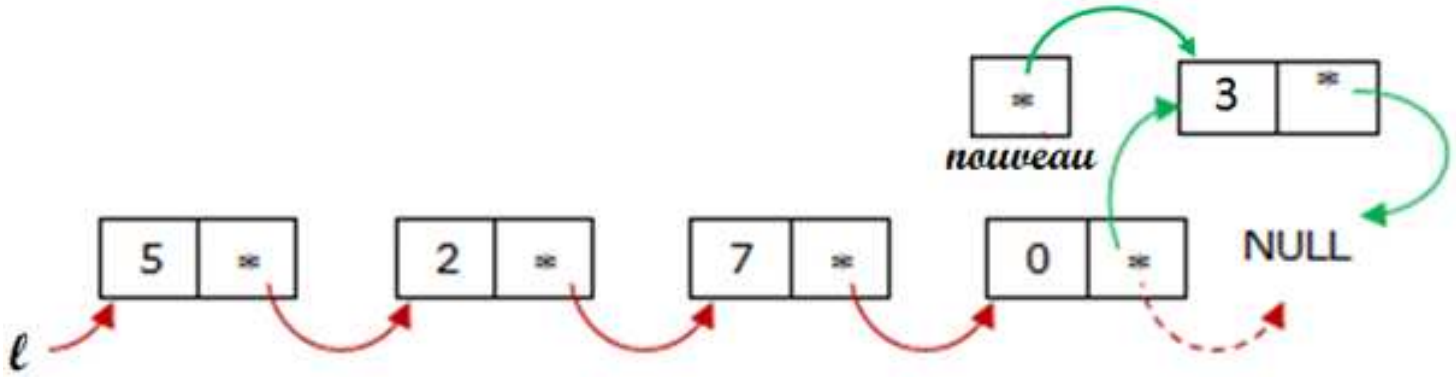
Lors d'un ajout en tête, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis pour terminer, raccorder cet élément à la liste passée en paramètre.




```
liste AjouterTete(liste l, int x)
{
    /*On crée un nouvel élément*/
    element *nouveau = (element*)malloc(sizeof(element));

   /* On assigne la valeur au nouvel élément */
    nouveau->valeur = x;

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

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

Ajouter un élément en fin de liste

- Il nous faut tout d'abord créer un nouvel élément **nouveau**, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à **NULL**.
- il faut faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire **tmp** sur la liste qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste.
- Un élément sera forcément le dernier de la liste s'il est pointé vers la constante symbolique **NULL** .



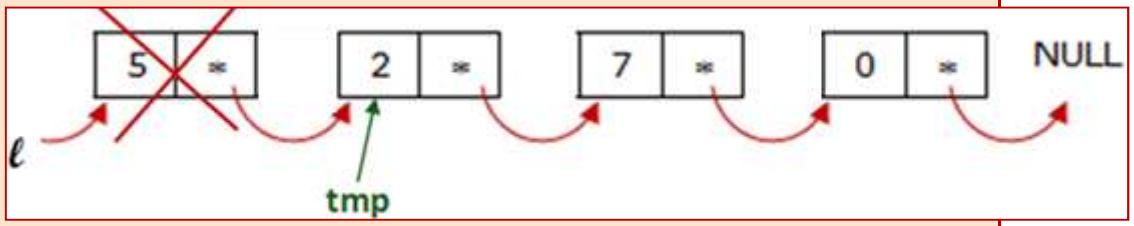
Ajouter un élément en fin de la liste

```
liste AjouterFin(liste l, int x)
{
    /*On crée un nouvel élément*/
    element* nouveau = (element*)malloc(sizeof(element));
    /*On assigne la valeur au nouvel élément*/
    nouveau->valeur = x;
    /*On ajoute en fin, donc aucun élément ne va suivre*/
    nouveau->suivant = NULL;
    if(l == NULL)
    {
        /*Si la liste est vidée il suffit de renvoyer l'élément créé*/
         return nouveau;
    }
    else
    {
        /*Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
         indique que le dernier élément de la liste est relié au nouvel élément*/
         element* tmp=l;
         while(tmp->suivant != NULL)
         {
            tmp = tmp->suivant;
         }
         tmp->suivant = nouveau;
         return l;
    }
}
```

Supprimer un élément en tête de la liste

- Si la liste n'est pas vide, on stocke l'adresse du deuxième élément, on supprime le premier élément, et on renvoie la nouvelle liste.
- Il faut utiliser la fonction free pour libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.

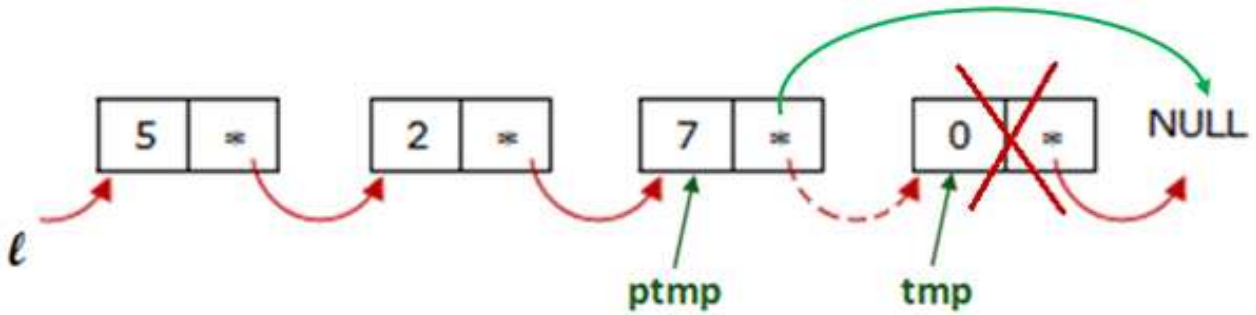
```
liste SupprimerTete(liste l)
{
    if(l != NULL)
    {
        /*Si la liste est non vide, on se prépare à renvoyer
        l'adresse de l'élément en 2ème position */
        element* tmp = l->suivant;
        /* On libère le premier élément */
        free(l);
        /* On retourne le nouveau début de la liste */
        return tmp;
    }
    else
    {
        return NULL;
    }
}
```



Supprimer un élément en fin de la liste

Il faut parcourir la liste jusqu'à son dernier élément, lier l'avant-dernier élément **NULL** et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine.

- Si la liste est vide, on retourne **NULL**
- Si la liste contient un seul élément, on le libère et on retourne **NULL**
- Si la liste contient au moins deux éléments alors on utilise deux pointeurs, un pointeur **tmp** qui pointe sur l'élément courant et un autre **ptmp** qui pointe sur l'élément précedent Tant qu'on n'est pas au dernier élément, on déplace ces pointeurs d'une position.
- A la sortie de la boucle , **tmp** pointe sur le dernier élément, et **ptmp** sur l'avant-dernier. On lie , **ptmp** à NULL et on libère **tmp.** L'avant-dernier se place alors à la fin et on retoure la nouvelle liste l qui pointe toujours sur le premier élément.



Supprimer un élément en fin de la liste

Il faut parcourir la liste jusqu'à son dernier élément, indiquer que l'avant-dernier élément va devenir le dernier de la liste et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine.

```
liste SupprimerFin(liste l)
{
    /* Si la liste est vide, on retourne NULL */
    if(l == NULL)
        return NULL;
    /* Si la liste contient un seul élément */
    if(l->suivant == NULL)
    {
        /*On le libère et on retourne NULL (la liste est maintenant vide)*/
        free(l);
        return NULL;
    }
     /* Si la liste contient au moins deux éléments */
     element* tmp = l;
     element* ptmp = NULL;
    /* Tant qu'on n'est pas au dernier élément */
    while(tmp->suivant != NULL)
    {
        /* ptmp stock l'adresse de tmp */
        ptmp = tmp;
        /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
         tmp = tmp->suivant;
    }
    /* A la sortie de la boucle, tmp pointe sur le dernier élément, et ptmp
    sur l'avant-dernier. On pointe l'avant-dernier élément vers NULL
    et on supprime le dernier élément */
    ptmp->suivant = NULL;
    free(tmp);
    return l;
}
```

Recherche d'un élément dans la liste

- Le but est de renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra **NULL**.
- L'intérêt est de pouvoir, une fois le premier élément trouvé, chercher la prochaine occurrence en recherchant à partir de l'élément qui suit l'élément recherché. On parcourt donc la liste jusqu'au bout, et dès qu'on trouve un élément qui correspond à ce que l'on recherche, on renvoie son adresse.

```
liste RechercherElement(liste l, int x)
{
    element *tmp=l;
    /* Tant que l'on n'est pas au bout de la liste */
    while(tmp != NULL)
    {
        if(tmp->valeur == x)
          /* Si l'élément a la valeur recherchée, on renvoie son adresse */
          return tmp;
        tmp = tmp->suivant;
    }
    return NULL;
}
```

Compter le nombre d'occurrences d'une valeur

- On cherche une première occurrence: si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.
- Il est aussi possible d'écrire cette fonction en parcourant l'ensemble de la liste avec un compteur que l'on incrémente à chaque fois que l'on passe sur un élément ayant la valeur recherchée.

```
int NombreOccurrences(liste l, int x)
{
    int i = 0; /* initier le compteur d'occurrence à 0 */
    /* Si la liste est vide, on renvoie 0 */
    if(l == NULL)
        return 0;
    /* Sinon, tant qu'il y a encore un élément ayant la valeur = x */
    while((l = RechercherElement(l, x)) != NULL)
    {
        /* On incrémente */
        l = l->suivant;
        i++;
    }
    /* Et on retourne le nombre d'occurrences */
    return i;
}
```

Recherche du i-ème élément

Il suffit de se déplacer i fois à l'aide du pointeur temporaire **tmp** le long de la liste chaînée et de renvoyer l'élément à l'indice i. Si la liste contient moins de i élément(s), alors nous renverrons **NULL**.

```
liste element_i(liste l, int indice)
{
    int i;
    /* On se déplace de i cases, tant que c'est possible */
    for(i=0; i<indice && l!= NULL; i++)
    {
        l = l->suivant;
    }
    /* Si l'élément est NULL, c'est que la liste contient moins de
    i éléments */
    if(l == NULL)
    {
        return NULL;
    }
    else
    {
        /* Sinon on renvoie l'adresse de l'élément i */
        return l;
    }
}
```

Compter le nombre d'éléments d'une liste chaîné

Nous allons utiliser un algorithme récursif, il faut donc connaître la condition d'arrêt et la condition de récurrence. On suppose pour cela que la fonction a compter les n-1 éléments précédents, et qu'il ne reste plus qu'à traiter le dernier élément.

- Si la liste est vide, il y a 0 élément
- 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

```
int NombreElements(liste l)
{
  /* Si la liste est vide, il y a 0 élément */
   if(l == 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 1+NombreElements(l->suivant);
}
```

Effacer tous les éléments ayant une certaine valeur

```
liste SupprimerElement(liste l, int x)
{
    /*Si la liste est vide, il n'y a plus rien à supprimer */
    if(l == NULL)
        return NULL;
    /*Si l'élément en cours de traitement est de valeur x, il doit être supprimé*/
    if(l->valeur == x)
    {
        /*On le supprime en prenant soin de mémoriser l'adresse de l'élément suivant*/
        element* tmp = l->suivant;
        free(l);
        /* L'élément ayant été supprimé, la liste commencera à l'élément suivant
        pointant sur une liste qui ne contient plus aucun élément ayant la valeur
        recherchée */
        tmp = SupprimerElement(tmp, x);
        return tmp;
    }
    else
    {
        /* Si l'élément en cours de traitement ne doit pas être supprimé, alors la liste
        finale commencera par cet élément et suivra une liste ne contenant plus
        d'élément ayant la valeur recherchée*/
        l->suivant = SupprimerElement(l->suivant, x);
        return l;
    }
}
```

Conclusion

Les listes sont des structures de données informatiques qui permettent, au même titre que les tableaux, de garder en mémoire des données en respectant un certain ordre : on peut ajouter, enlever ou consulter un élément en début ou en fin de liste, ...

Les listes chaînées, par rapport aux tableaux, possèdent les avantages suivants :

- Elles sont dynamiques et n'occupent qu'un minimum d'espace lorsqu'il n'y a pas d'élément à l'intérieur,
- L'ajout ou le retrait d'une donnée dans une liste ordonnée ne nécessite pas le déplacement des autres données de la liste,
- L'ajout d'une donnée ne demande qu'une réservation minimal de mémoire (la taille d'une donnée et celle d'un ou deux pointeur(s)), alors que pour un tableau dynamique il faut pouvoir alloué un bloc important de mémoire.

Cependant, les listes chainées amènent aussi les désavantages suivants :

- Le code nécessaire pour les gérer est plus complexe que celui pour gérer des tableaux dynamiques,
- Elles occupent plus d'espaces qu'un tableau dynamique pour une même quantité de donnée (à cause des pointeurs),
- Elles ne permettent pas d'accéder directement à un élément, ce qui empêche entre autres la recherche dichotomique.