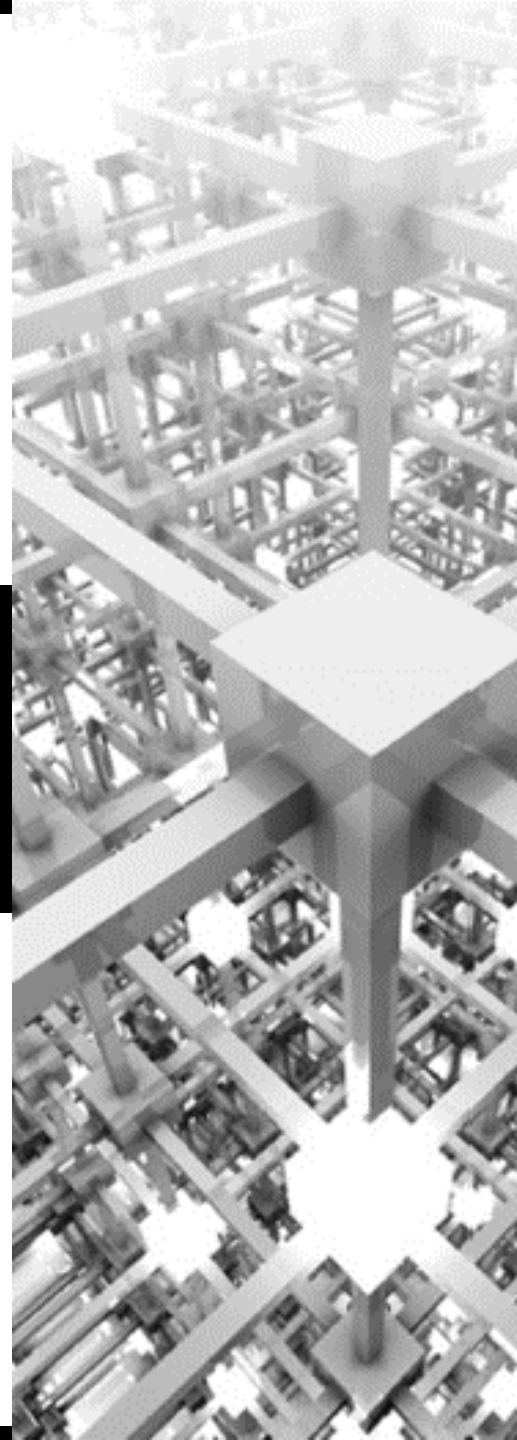
ENSAM

STRUCTURES DE DONNÉES EN C

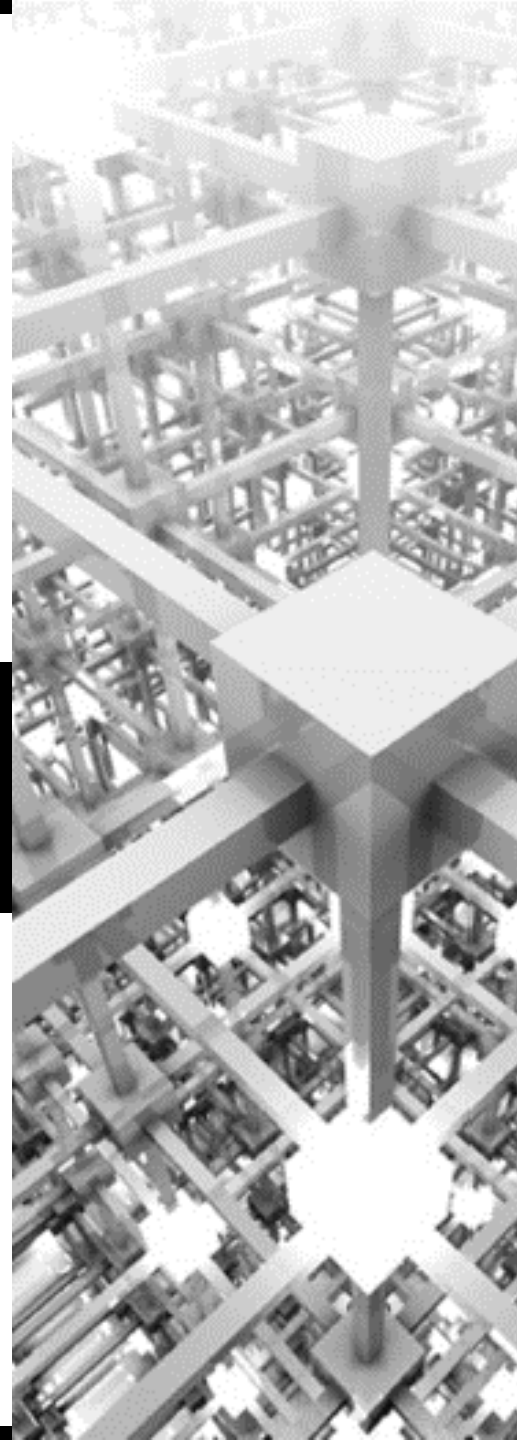
1ère Année «Cycle ingénieur : Intelligence Artificielle et Génie Informatique»

2023/2024

Dep. Informatique
Pf. CHERGUI Adil



ENSAM



Séance 2

LES LISTES CHAINÉES

Objectifs de la séance :

- Étudier le principe et les objectifs des liste chainées
- Explorer les différents types de listes

Pf. Adil CHERGUI

INTRODUCTION

L'objectif des structures de données

L'informatique a révolutionné le monde moderne grâce à sa capacité à traiter de grandes quantités de données, des quantités beaucoup trop grandes pour être traitées à la main. Cependant, pour pouvoir manipuler efficacement des données de grande taille il est en général nécessaire de bien les structurer : **tableaux (dynamiques et statiques), listes, piles, arbres, tas, graphes**... sont des structures qui servent à cette finalité.

Une multitude de structures de données existent, et une multitude de variantes de chaque structure, chaque fois mieux adaptée à un contexte ou algorithme en particulier.

Les tableaux permettent de stocker de grandes quantités de données de même type. Mais, même s'il est possible d'agrandir un tableau une fois plein (ou constaté trop petit), ceci consiste en une opération **coûteuse** (réallocation, copie, etc).
A ce genre de structure, on préfère souvent **les structures dynamiques**, qui grossissent selon les besoins. Ainsi, seule la place nécessaire est réservée.

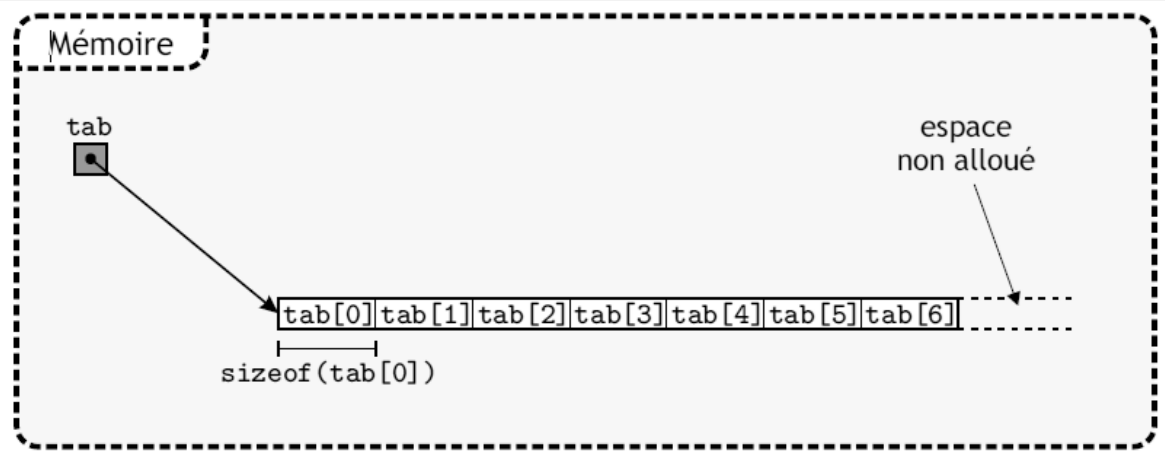
Une application de ce principe qui sera vue en détail dans ce chapitre consacré à la structure de **listes chaînées**.

STRUCTURES DE DONNÉES CLASSIQUES

Les tableaux statiques

Les tableaux représentent la structure de stockage de donnée la plus simple. Ils sont en général implémentes nativement dans la majorité des langages de programmation et sont donc simples à utiliser.

Un tableau représente une zone de mémoire consécutive d'un seul bloc (*cf.* Figure ci-dessous) ce qui présente à la fois des avantages et des inconvénients :



Avantages et inconvénients

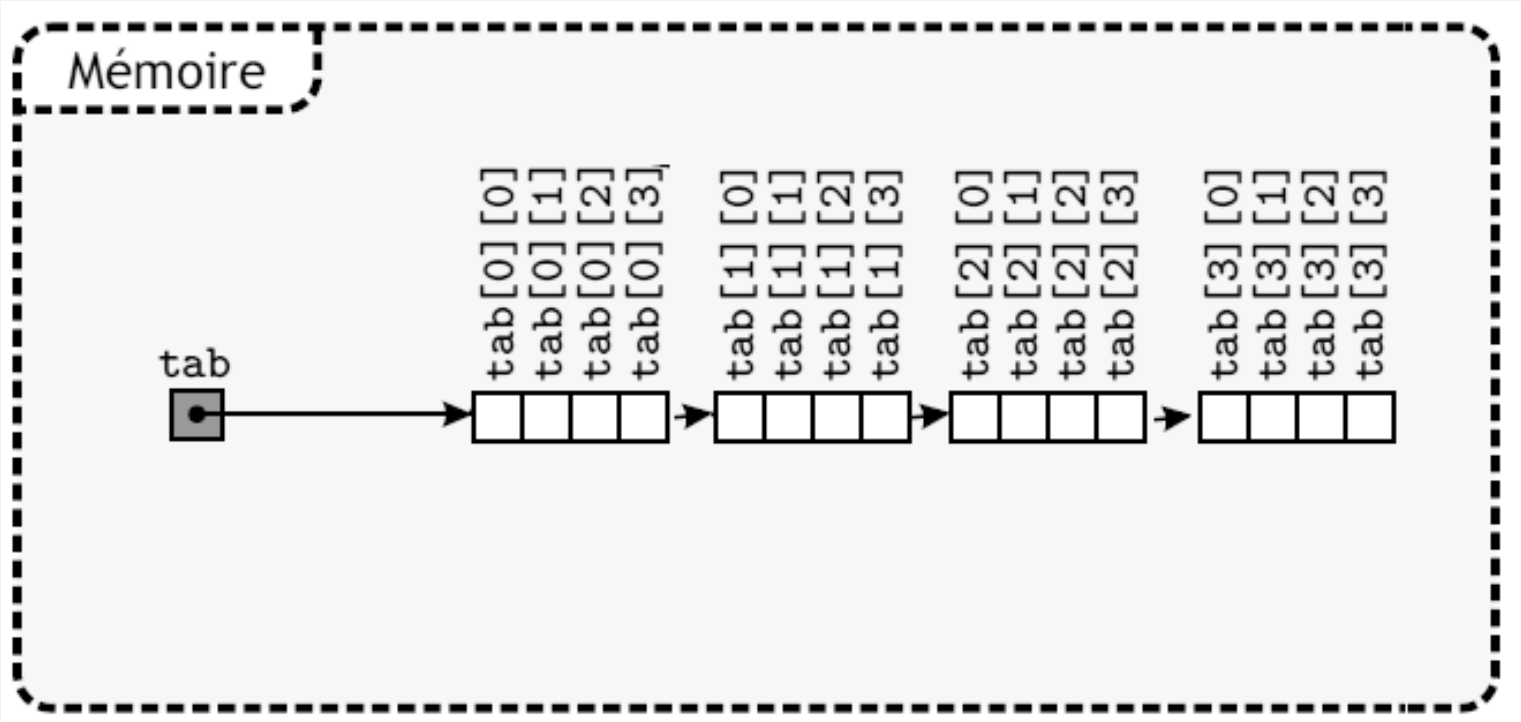
- La mémoire est en un seul bloc consécutif avec des éléments de taille constante à l'intérieur (la taille de chaque élément est défini par le type du tableau), donc il est très facile d'accéder au iéme élément du tableau. L'instruction tab[i] se contente de prendre l'adresse mémoire sur laquelle pointe tab et d'y ajouter i fois la taille d'un élément.
- Il faut fixer la taille du tableau avant de commencer à l'utiliser. Les systèmes d'exploitation modernes gardent une trace des processus auxquels les différentes zones de mémoire appartiennent : si un processus va écrire dans une zone mémoire qui ne lui appartient pas (une zone que le noyau ne lui a pas alloué) il y a une erreur de segmentation.
- La structure de type tableau pose des problèmes pour insérer ou supprimer un élément car ces actions nécessitent des décalages du contenu des cases du tableau qui prennent du temps dans l'exécution d'un programme.

STRUCTURES DE DONNÉES CLASSIQUES

Allocation statique d'un tableau

Il existe deux façons d'allouer de la mémoire à un tableau.

– la plus simple permet de faire de l'allocation ***statique***. Par exemple ***int*** **T[100];** qui va allouer un tableau de 100 entiers pour **T**. De même ***int*** **tab[4][4];** va allouer un tableau à deux dimensions de **taille 4 × 4**. En mémoire ce tableau bidimensionnel peut ressemblera à ce qu'on voit dans la ci-dessus. Ici, les éléments sont stockés ligne à côté de ligne, et il se trouve dans la même zone que toutes les variables de type ***int***. On appelle cela de l'allocation statique car on ne peut pas modifier la taille du tableau en cours d'exécution.



STRUCTURES DE DONNÉES CLASSIQUES

Allocation dynamique d'un tableau

La deuxième façon utilise soit la commande **new** (syntaxe C++), soit la commande **malloc** (syntaxe C) et permet une allocation dynamique (dont la taille dépend des entrées par exemple). L'allocation du tableau s'écrit alors :

int* tab = new int[100];

ou

int* tab = (int*) malloc(100*sizeof(int));.

En revanche cette technique ne permet pas d'allouer directement un tableau à deux dimensions. Il faut pour cela effectuer une boucle qui s'écrit alors comme dans l'exemple de code ci-contre :

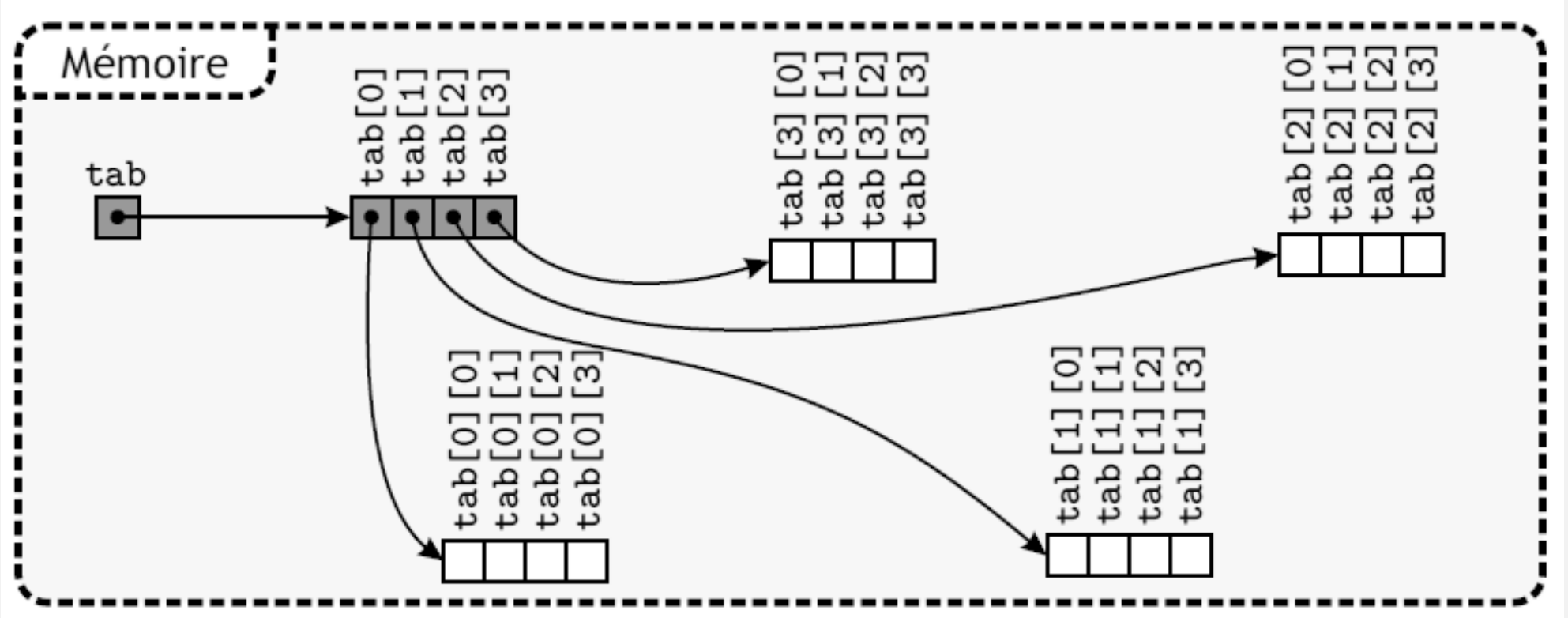
Exemple

```
1  #include <stdio.h>
2  int main()
3  {
4        int i;
5        int** tab2;
6        int** tab3;
7        tab2 = (int**) malloc(4*sizeof(int*));
8        for (i=0; i<4; i++)
9        {
10             tab2[i] = (int*) malloc(4*sizeof(int));
11       }
12       /* ou en utilisant new */
13       tab3 = new int*[4];
14       for (i=0; i<4; i++) {
15       tab3[i] = new int[4];
16       }
17     return 0;
18 }
```

STRUCTURES DE DONNÉES CLASSIQUES

Allocation dynamique d'un tableau

Attention, un tableau alloué statiquement ne se trouve pas dans la même zone mémoire qu'un tableau alloué avec l'une des méthodes d'allocation dynamique :



LES LISTES CHAINÉES

Définitions

Ce type classique de stockage de valeurs peut donc être coûteux en temps d'exécution.

Il existe une autre structure, appelée **liste chaînée**, pour stocker des valeurs, elle peut implémenter pratiquement n'importe quoi, par exemple,

- une liste d'entiers [3; 2; 4; 2; 5],
- une liste de courses [pommes; beurre; pain; fromage],
- ou une liste composée de plusieurs éléments tel qu'une pages Web contenant chacune une image et un lien vers la page Web suivante.

Ce type de structure permet plus aisément **d'insérer** et de **supprimer** des valeurs dans une liste linéaire d'éléments.

Une **liste chaînée** est une **structure linéaire** qui n'a pas de **dimension fixée** à sa création. Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs.

Sa **dimension** peut être modifiée selon la **place disponible en mémoire**.

La liste est accessible **uniquement** par sa tête de liste c'est-à-dire son premier élément.

LES LISTES LINÉAIRES

Définitions

Une **liste linéaire** est une structure de données correspondant à **une suite d'éléments**. Les éléments **ne** sont **pas indexés** dans la liste, mais pour chaque élément (sauf le dernier) on peut **accéder** à **l'élément suivant**. Par conséquent, on ne peut accéder à un élément qu'en passant par **le premier élément** de la liste et en parcourant tous les éléments jusqu'à ce qu'on atteigne l'élément recherché. Ce type de structure permet plus aisément d'insérer et de supprimer des valeurs dans une liste linéaire d'éléments.

Les différents types de liste chaînée

Il existe différents types de listes chaînées :

- **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
- **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la liste reste triée.
- **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. Ceci permet de lire la liste dans les deux sens, du premier vers le dernier élément ou inversement.
- **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.

LES LISTES SIMPLEMENT CHAINÉES

Eléments utilisées dans les listes

Une liste simple est une structure de données telle que chaque élément contient :

- des informations caractéristiques de l'application (les caractéristiques d'une personne par exemple),
- un pointeur vers un autre élément ou une marque (NULL) de fin s'il n'y a pas d'élément successeur.

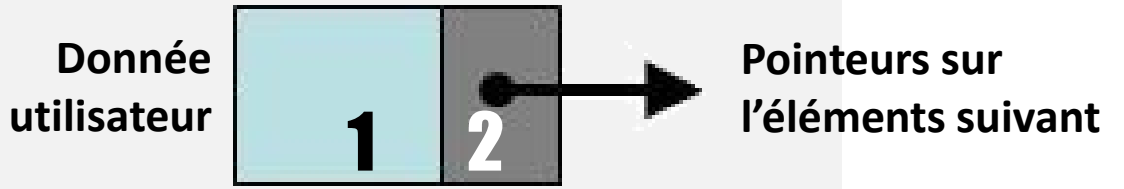
Chaque élément est créé à partir d'un **structure prototype** qui ce fait nommé dans communauté par plusieurs noms : **Maillon, Cellule, Element, Nœud, Case**, …

Nous allons choisir dans ce cours les nominations suivantes :

```
typedef struct cell
{
        int val;
        struct cell *next;
}cellule;
```

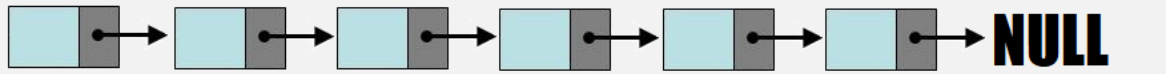
Représentation graphique

Pour bien comprendre les manipulations des listes chaînées, il est très intéressant de les représentées graphiquement. La forme d'une cellule est composée de deux rectangles, le rectangle 1 représente le contenu de la cellule, le rectangle 2 représente le lien vers la cellule suivante.



représentation d'un élément d'une liste simplement chaînée.

Ainsi, les étapes de composition et de manipulation des listes chaînées se font assimiler de manière graphique avant d'implémenter leurs codes.

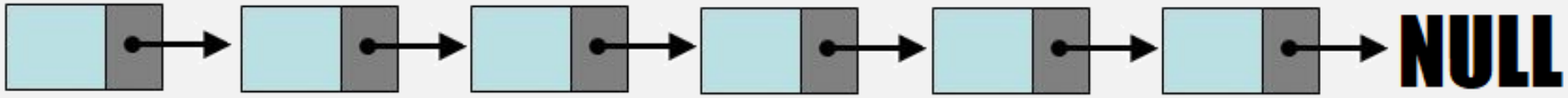


TYPES DE LISTES CHAINÉES

Types de liste

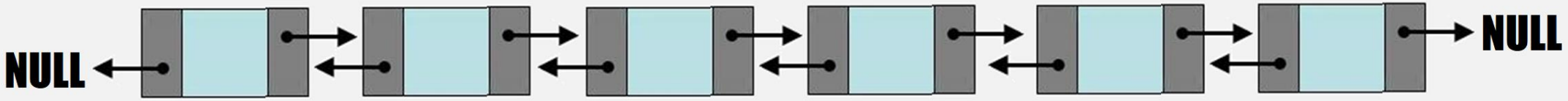
Liste simplement chainnée

La liste chainée simple permet de circuler que dans un seul sens, c'est ce modèle :



Liste symétrique ou doublement chainée

Avec le modèle double chaque élément possède l'adresse du suivant et du précédent ou des marques de fin s'il n'y en a pas. Il est alors possible de parcourir la chaine dans les deux sens :

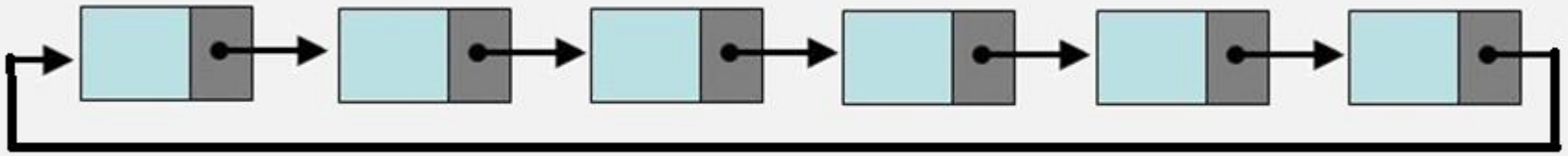


TYPES DE LISTES CHAINÉES

Types de liste

Liste circulaire simple

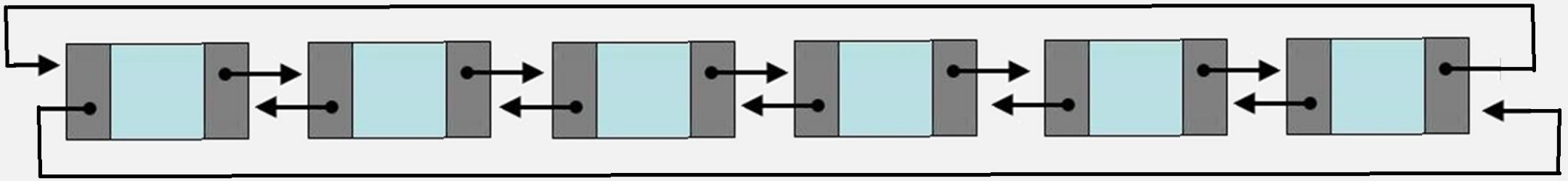
Dans une liste circulaire simple le dernier prend l'adresse du premier et la circulation est prévue dans un seul sens :



Dans ce modèle premier et dernier n'ont plus la même importance, le premier peut être n'importe quel maillon de la chaine et le dernier celui qui le précède.

Liste circulaire double

Même principe que précédemment mais avec une circulation possible dans les deux sens :



LES ACTIONS SUR UNE LISTE CHAINÉE

Fonctions attendues

Au vu de l'utilisation des listes chaînées, il se dessine clairement quelques fonctions indispensables :

- Initialisation
- Ajout d’un élément
- Suppression d’un élément
- Accès à l’élément suivant
- Accès aux données utilisateur
- Accès aux premiers éléments de la liste
- Calcul de la taille de la liste
- Suppression de la liste entièrement
- Copier une liste
- Copier la liste sur fichier ou récupérer la liste dans le programme

Comparaisons: Listes chainées contre tableaux

Tableaux	Liste Chainées
Taille en mémoire : Par nature, la tableau a une taille définie même dans le cas d'un tableau dynamique dont la taille peut être réévaluée périodiquement. La taille du tableau fait partie de la définition du tableau.	La liste chainée dynamique n'a pas pour sa définition de nombre d'éléments. Ils sont ajoutés ou soustraits à la demande, pendant le fonctionnement du programme.
Soustraire un élément : impossible dans un tableau d'enlever une case du tableau. Il est éventuellement possible de masquer un élément mais pas de retirer son emplacement mémoire.	Aucun problème pour soustraire un élément de la liste et libérer la mémoire correspondante.
Accès élément : Le tableau permet d'accéder à chaque élément directement. C'est très rapide.	Pour accéder à un élément il faut parcourir la liste jusqu'à lui, ça peut être long.
Tris : Les tris de tableau ne nécessitent pas de reconstruire les liens entre les emplacements mémoire du tableau. Il y a juste à manipuler les valeurs afin de les avoir dans l'ordre voulu	Il ne suffit pas de manipuler les valeurs il faut aussi reconstruire la chaine. Le mieux est de construire sa chaine en mettant les éléments dans l'ordre dés le départ plutôt que d'avoir à réorganiser l'ordre des éléments dans la chaine. Trier une chaine revient à construire une autre chaine en insérant dans l'ordre voulu chaque élément.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

Nous allons détailler ici tous les aspects de l'implémentation d'une liste chainée.

Des exemples sont notamment donnés pour l'insertion et la suppression d'éléments sur critère. Un exemple est donné également pour la sauvegarde et le chargement d'une liste.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

a. Structure de donnée d'un maillon

Chaque élément de la liste est une structure qui contient des informations pour l'application (ici deux variables (un nombre et une chaine), juste pour faire des tests) et un pointeur sur une structure de même type :

```
typedef struct cel
{
    // 1 : les datas pour nos tests
    int val;
    char s[80];
    // 2 : le pointeur pour l'élément suivant
    struct cel* next;
}cell;
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

b. Début et fin de la liste

Deux positions sont très importantes dans une liste chainée : **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 et sans le **dernier** impossible de savoir où elle s'arrête.

Le début est donné par l'adresse du premier maillon. Une chaine prend ainsi le nom du premier maillon :

```
cell *T=NULL;
```

Lorsque la liste est vide le premier maillon prend **TOUJOURS** la valeur **NULL**.

La fin de la liste est indiquée par une sentinelle à savoir une valeur spécifique reconnaissable. Il y a plusieurs possibilités. Nous utiliserons ici la plus fréquente : la valeur **NULL**.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

b. Début et fin de la liste

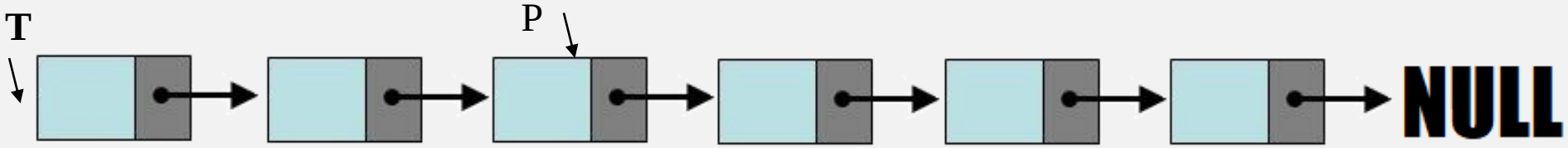
Deux positions sont très importantes dans une liste chainée : **le début et la fin**, souvent désignées par "premier et dernier" ou "tête et queue". Sans le premier **impossible** de savoir où commence la chaine et sans le **dernier** impossible de savoir où elle s'arrête.

Le début est donné par l'adresse du premier maillon. Une chaine prend ainsi le nom du premier maillon :

```
cell *T=NULL;
```

Lorsque la liste est vide le premier maillon prend **TOUJOURS** la valeur **NULL**.

La fin de la liste est indiquée par une sentinelle à savoir une valeur spécifique reconnaissable. Il y a plusieurs possibilités. Nous utiliserons ici la plus fréquente : la valeur **NULL**.



La fin est connue si : p->next = = NULL

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

b. Début et fin de la liste

Attention

Avec les listes chainées il est recommandé de TOUJOURS METTRE A NULL UN POINTEUR NON ALLOUE. En effet un pointeur peut contenir une adresse mémoire qui n'est pas allouée, par exemple :

Cell *T;

T contient ce qui traine en mémoire, une adresse quelconque non allouée. Une adresse mémoire non allouée est indétectable et écrire à une adresse non réservée fait planter le programme. La valeur NULL est détectable, elle signifie que le pointeur n'est pas alloué et le programme peut le détecter.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

c. Initialiser un maillon

Il est indispensable d'allouer la mémoire pour chaque nouvel élément de la liste. Le mieux est de le faire avec l'initialisation des données dans une fonction d'initialisation.

Pour nos tests le champ val prend une valeur aléatoire entre 0 et 26 et le champ s prend une chaine de caractères composée d'une seule lettre de l'alphabet choisie au hasard. Le pointeur **next** est initialisé à **NULL**.

A la fin, la fonction d'initialisation retourne l'adresse du nouvel élément alloué et initialisé :

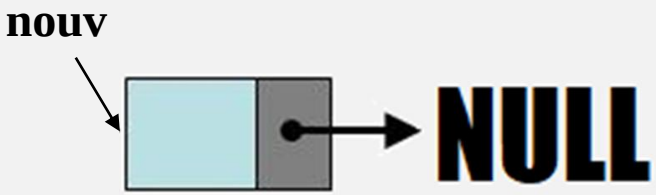
```
cell* init()
{
    char* n[]={"A","B","C","D","E","F","G","H","I","J","K",
    "L","M","N","O","P","Q","R","S","T","U","V","X","Y","Z"};
    cell *e=malloc(sizeof(t_elem));
    e->val=rand()%26;
    strcpy(e->s,n[rand()%26]);
    e->next=NULL;
    return e;
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

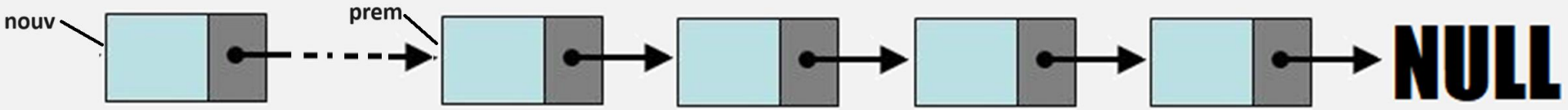
d. Ajouter au début

Pour construire une liste il faut simplement pouvoir ajouter des maillons. Le plus simple est d'ajouter en début de liste, avant la tête et de déplacer la tête ensuite.

Soit un élément déjà initialisé avec la fonction init() :



Le nouvel élément est ajouté au début de la liste :



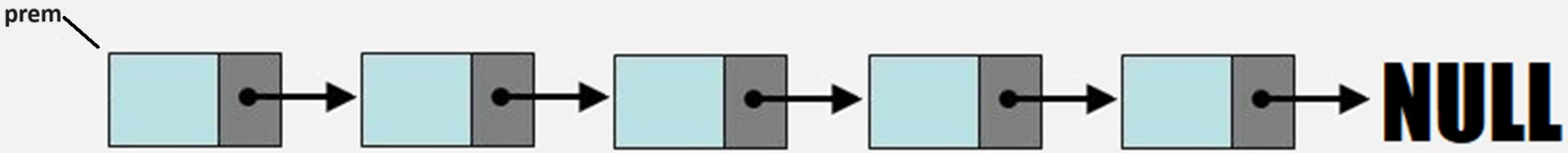

```
nouv->next = premier
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

d. Ajouter au début

Le pointeur "premier" qui donne l'adresse de la liste, ne contient plus l'adresse du début qui est maintenant celle de nouveau, premier doit donc prendre cette adresse :

prem=nouv



IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

d. Ajouter au début

Pour la fonction il y a deux possibilités d'écriture. Soit prendre en paramètre l'adresse de début et utiliser le mécanisme de retour pour renvoyer la nouvelle adresse, soit prendre en paramètre l'adresse "perso" de la variable premier, à savoir passer le pointeur de début par référence.

Première version. La fonction ajout() prend en argument l'adresse du début de la liste, l'adresse du nouvel élément et retourne la nouvelle adresse du début :

```
cell* ajout_debut1(cell*prem,cell*e)
{
    e->next=prem;
    prem=e;
    return prem;
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

d. Ajouter au début

Exemple d'appel, dans le programme ci-dessous une chaine de 10 éléments est fabriquée :

Attention à bien initialiser premier sur NULL à la déclaration sinon la chaine n'aura pas de signe final et il ne sera pas possible de repérer sa fin.

```
int main()
{
    cell* premier=NULL;
    cell*nouveau;
    int i;
    for (i=0; i<10; i++)
    {
        nouveau=init();
        premier=ajout_debut1(premier,nouveau);
    }
    return 0;
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

d. Ajouter au début

Deuxième version. Le mécanisme de retour n'est pas utilisé. Le premier paramètre est un pointeur de pointeur afin de pouvoir prendre comme valeur l'adresse de la variable pointeur qui contient l'adresse du premier élément : &premier

```
void ajout_debut2(cell **prem,cell*e)
{
    e->next=*prem;
    *prem=e;
}
```

```
int main()
{
    cell* premier=NULL;
    cell*nouveau;
    int i;

    for (i=0; i<10; i++)
    {
        nouveau=init();
        ajout_debut2(&premier,nouveau);
    }
    return 0;
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

e. Parcourir la liste

Pour parcourir une liste chainée il suffit avec un pointeur de prendre successivement l'adresse de chaque cellule. Au départ le pointeur prend l'adresse du premier élément qui est l'adresse de la liste, ensuite il prend l'adresse du suivant et ainsi de suite jusqu'il retrouve la fin de liste (NULL) :

```
void print_liste(cell *L)
{
    cell *p=NULL;
    if(L==NULL)
    {
        printf("La liste est VIDE!\n");
    }
    else
    {
        for(p=L;p!=NULL;p=p->next)
        {
            printf("%s %d\n",p->s,p->val);
        }
    }
}
```

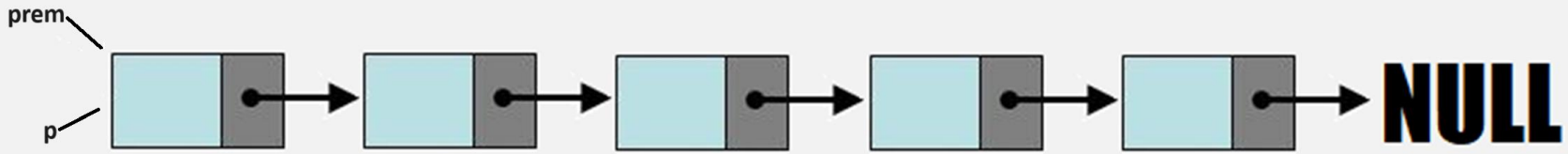
```
int main()
{
    cell* premier=NULL;
    cell*nouveau;
    int i;
    for (i=0; i<10; i++)
    {
        nouveau=init();
        ajout_debut2(&premier,nouveau);
    }
    printf("Affichage de la liste\n");
    print_liste(premier);
    return 0;
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

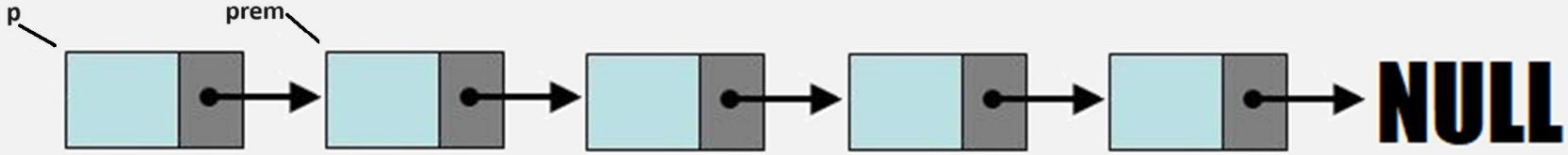
f. Supprimer au début

La suppression du premier élément suppose de bien actualiser la valeur du pointeur premier qui indique toujours le début de la liste, ce qui donne si la liste n'est pas vide :

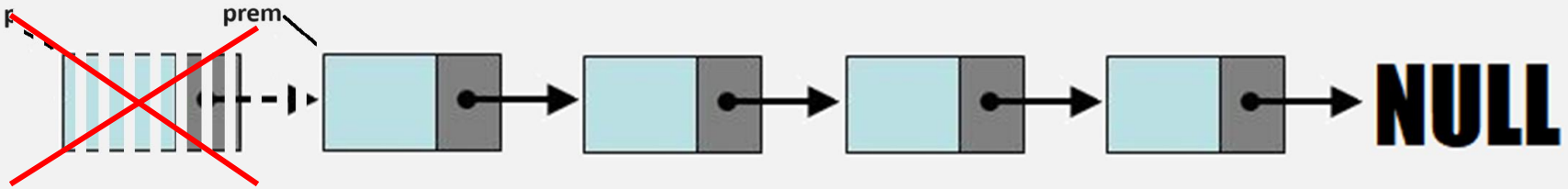
1. p=prem



2. prem=prem->next



3. Free(p)



IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

f. Supprimer au début

La modification du pointeur premier doit effectivement être répercutée sur le premier pointeur de la liste. De même que pour les fonctions d'ajout il faut retourner la nouvelle adresse de début ou utiliser un passage par référence afin de communiquer cette nouvelle adresse du début au contexte d'appel. La version présentée est avec un passage par référence du pointeur premier :

```
void supprimer_debut(cell**prem)
{
    cell*p;
    if(*prem!=NULL)
    {
        p=*prem;
        *prem=(*prem)->next;
        free(p);
    }
}
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

g. Supprimer un élément sur critère

L'objectif cette fois est de supprimer une cellule quelleconque de la liste en fonction d'un critère donné. Le critère ici est que le champ val de l'élément soit égal à une valeur donnée. La recherche s'arrête si un élément répond au critère et nous supprimons uniquement le premier élément trouvé s'il y en a un. Comme pour les suppressions précédentes la liste ne doit pas être vide. Il faut prendre en compte le cas où l'élément à supprimer est le premier de la liste et pour tous les autres éléments il faut disposer de l'élément qui précède celui à supprimer afin de pouvoir reconstruire la chaine.

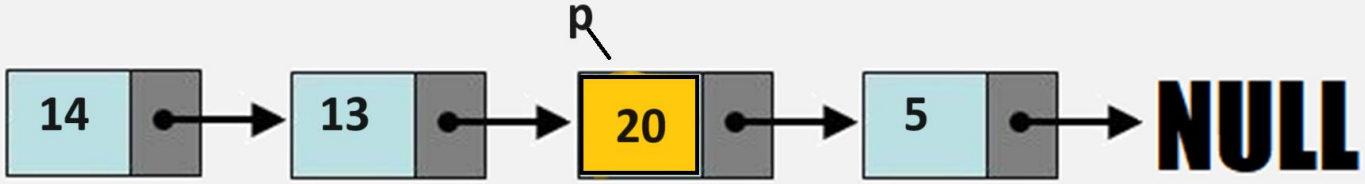
IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

g. Supprimer un élément sur critère

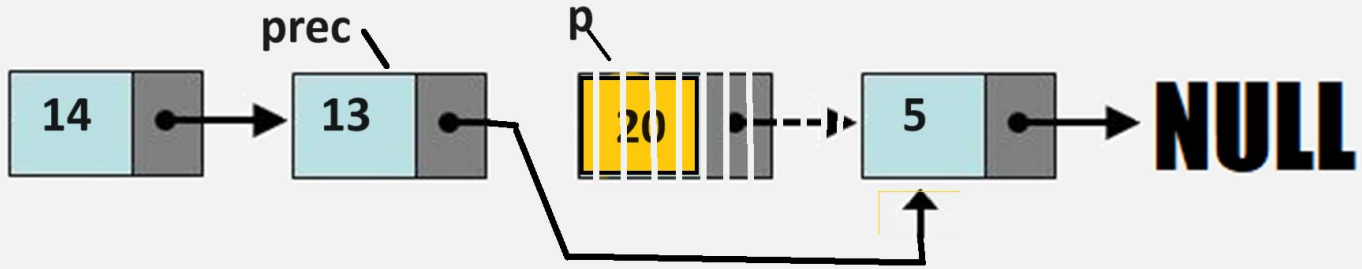
Prenons par exemple 20 comme valeur à rechercher. Si un maillon a 20 pour valeur entière il est supprimé et la recherche est terminée. Le principe est le suivant :

1. si c'est le premier élément : supprimer le premier comme indiqué plus haut

2. sinon trouver dans la liste un maillon où il y a 20 :



3. le supprimer, ce qui suppose d'avoir l'adresse du précédent afin de rétablir la chaine avec le suivant du maillon 20 :



IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

g. Supprimer un élément sur critère

La fonction reçoit en paramètre l'adresse du début de la liste et la valeur entière critère de sélection de l'élément à supprimer. Le premier élément trouvé qui a cette valeur dans son champ val est supprimé de la liste. Comme l'adresse de la liste est susceptible d'être modifiée, la fonction retourne cette adresse afin que cette modification puisse être récupérée dans le contexte d'appel. Nous proposons l'algorithme suivant :

```
Si la liste n'est pas vide
  S'il s'agit d'enlever le premier
          - l'adresse du premier est sauvée en p
          - le premier devient le suivant du premier
          - l'adresse sauvée en n est libérée
  Si ce n'est pas le premier, voir parmi les autres, pour ce
faire
          - le précédent est le premier
          - le courant p est le suivant de premier
          - Tant que p!= NULL
                - si val est trouvée
                   - créer lien entre prec et suivant de p
                   - libérer la mémoire de l'élément p
                   - provoquer la sortie de la boucle
                - sinon continuer la recherche
                   - precedent = p et
                   - p= suivant
              FinTant que
à l'issue retourner prem qui a peut-être été modifié.
```

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

g. Supprimer un élément sur critère

TP 1 : Exercice 1

A vous de programmer la fonction on proposant les deux versions (celle qui retourne la liste, et celle qui utilise le passage par référence)

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

h. Détruire la liste

Pour détruire une liste il faut parcourir la liste, récupérer l'adresse de la cellule courante, passer au suivant et désallouer l'adresse précédente récupérée. Au départ bien vérifier que la liste n'est pas vide. La tête de liste est modifiée, elle doit à l'issue passer à NULL. Il y a les deux possibilités habituelles :

- retourner la valeur NULL à la fin et la récupérer avec le pointeur de tête premier dans le contexte d'appel
- ou passer le pointeur premier par référence.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

h. Détruire la liste

TP 1 : Exercice 2

A vous de programmer la fonction on proposant les deux versions (celle qui retourne la liste, et celle qui utilise le passage par référence)

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

k. Sauvegarder la liste dans un fichier

Ecriture sur fichier

L'objectif est de sauver une liste chainée dynamique dans un fichier binaire, c'est à dire en langage C en utilisant la fonction standard **fwrite().**

Le principe est simple : si la liste n'est pas vide, ouvrir un fichier en écriture et copier dedans dans l'ordre où il se présente chaque maillon de la liste.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

k. Sauvegarder la liste dans un fichier

TP 1 : Exercice 3

A vous de programmer la fonction

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

I. Restaurer une liste à partir de fichier

Lecture à partir du fichier

Pour récupérer dans le programme une liste chainée dynamique préalablement sauvegardée dans un fichier binaire, il est nécessaire de reconstruire la liste au fur et à mesure. En effet les adresses contenues dans les pointeurs ne sont plus allouées et il faut réallouer toute la liste La fonction ci-dessous suppose qu'il y a au moins un maillon de sauvegardé dans le fichier et il ne doit pas y avoir de sauvegarde de liste vide (c'est à dire juste une création de fichier avec rien dedans).

La fonction retourne l'adresse du début de la liste, c'est à dire l'adresse du premier élément, la tête de liste.

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

I. Restaurer une liste à partir de fichier

TP 1 : Exercice 4

A vous de programmer la fonction

IMPLÉMENTER UNE LISTE SIMPLE EN DYNAMIQUE

Problème

TP 1 : problème

Modifier le programme en entier pour qu'il puisse

- **Demander à l'utilisateur d'introduire les données.**
- **Faire l'insertion des données en ordre par rapport à la chaine s ou par rapport à la valeur val.**