2.10 Les structures et les fonctions.

● **Passage d'une variable de structure en tant qu'argument d'une fonction : *Passage par valeur.***

```c
#include < stdio.h>

struct etudiant{
    char prenom[20];
    int age;
};
void afficher(struct etudiant et)
{
    printf(" prenom : %s \n", et.prenom);
    printf(" age : %d \n", et.age);
}

int main(void)
{
    struct etudiant et1 = {"Mostafa", 24};

    afficher(et1);

    return 0;
}
```

● **Passage d'une variable de structure en tant qu'argument d'une fonction :** ***Passage par référence (adresse)***

```c
#include < stdio.h>

struct complex{
    float R; // partie réelle
    float I; // Partie Imaginaire
};
void Ajouter(struct complex c1, struct complex c2, struct complex *res)
{
    res->R=(c1.R + c2.R);
    res->I=(c1.I + c2.I);
}

int main(void)
{
    struct complex c1 = {2.5, 3}, c2={1.24,4}, somme;

    Ajouter(c1, c2, &somme);

    printf("res.R = %f  et res.I = %f", somme.R,somme.I);

    return 0;
}
```

● **Passage d'un tableau de structure en tant qu'argument**

- ***Méthode 1*** :

```c
void myFunction(struct  etudiant  *t , int n )
{
. . .
}
```

- ***Méthode 2 :***

```
void myFunction(struct  etudiant t[100] )
{
. . .
}
```

- Méthode 3 :

```
void myFunction(struct  etudiant t[], int n )
{
. . .
}
```

● **Renvoyer une structure à partir d'une fonction**

- **Exemple 1 :**

```
struct etudiant{
     char nom[20];
     int age;
};

struct etudiant Saisir(){
     // variable local;
     struct etudiant e;

     printf("Saisir le prénom : ");
     scanf("%s",&e.nom);

     printf("saisir l'age : ");
     scanf("%d",&e.age);

     // retourner une copie de e;
     return e;
}

int main(void)
{
     struct etudiant et;
     et=Saisir();

     printf("voici vos infos : ");
     printf("Prénom : %s",et.nom);
     printf("age : %d",et.age);

     return 0;
```

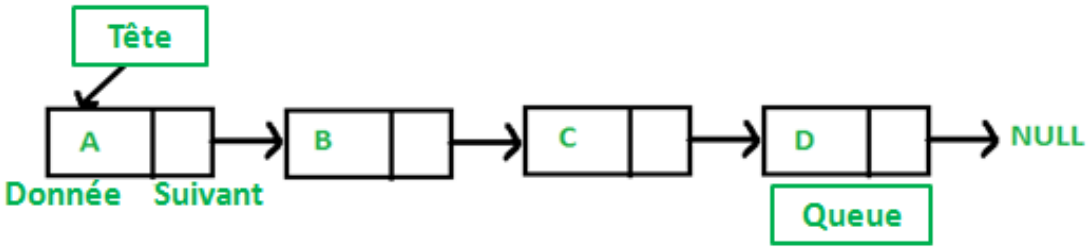
- **Exemple 2**

```c
struct etudiant{
    char prenom[20];
    int age;
};

struct etudiant *Saisir(struct etudiant *e){

    printf("Saisir le prénom : ");
    scanf("%s",&e->prenom);

    printf("saisir l'age : ");
    scanf("%d",&e->age);

    return e;
}

int main(void)
{
    struct etudiant et;
    struct etudiant *et2;
    et2=Saisir(&et);

    printf("voici vos infos : \n");
    printf("Prénom : %s \n",et2->prenom);
    printf("age : %d",et2->age);

    return 0;
}
```

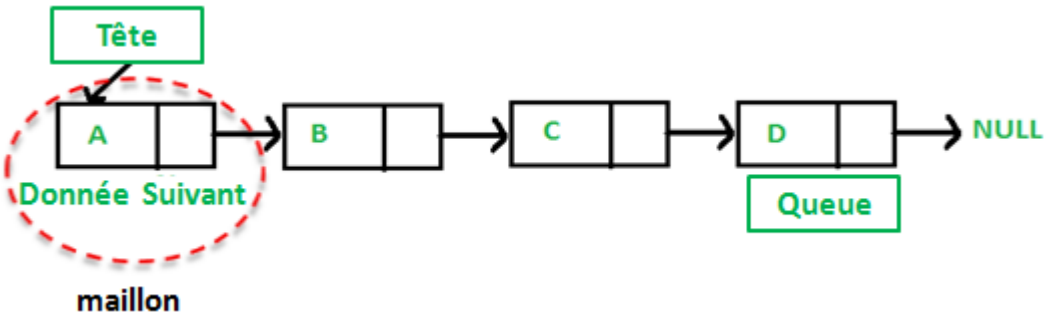
3) CHAPITRE 3 : Les listes chainées, les piles et les fils.

3.1 Définition :

- Une liste chaînée est une structure de données linéaire, dans laquelle les éléments ne sont pas stockés à des emplacements de mémoire contigus.
- Les éléments d'une liste chaînée sont liés à l'aide de pointeurs comme indiqué dans l'image ci-dessous:



- L'élément de base d'une liste chaînée s'appelle le maillon (structure). Il est constitué : d'un ou plusieurs champs de données et d'un pointeur vers le maillon suivant. S'il n'y a pas de maillon suivant, le pointeur vaut NULL.



3.2 Listes chainées VS. Tableaux.

- Dans un tableau les données sont stockées à des emplacements de mémoire contigus. Contrairement à une liste chainée.
- Contrairement au tableau, la liste n'est pas allouée en une seule fois, mais chaque élément est alloué indépendamment (maillon par maillon).
- La liste chainée offre une facilité d'insertion / suppression surtout en cas d'éléments triés. Contrairement au tableau où nous devons insérer et déplacer le reste des éléments.

  - Exemple :
    Soit le tableau suivant id [] = [1000, 1010, 1050, 2000, 2040].

Pour insérer la valeur 1005 et maintenir l'ordre trié, nous devons déplacer tous les éléments après 1000 (sauf 1000).

- L'accès aléatoire n'est pas autorisé dans une liste chainée. Nous devons accéder séquentiellement aux éléments à partir du premier nœud.
- Dans une liste chainée, un espace mémoire supplémentaire pour un pointeur sur l'élément suivant est requis avec chaque élément de la liste.

3.3 Implémentation de la liste chainée.

- La structure de données de la liste chainée contient deux parties :

  - la valeur que vous voulez stocker,

  - L'adresse de l'élément suivant.

Exemple :

```
typedef  struct Maillon
{
  int donnee;
  struct Maillon* suivant;

} Maillon;
```

3.4 Fonctions de gestion de liste

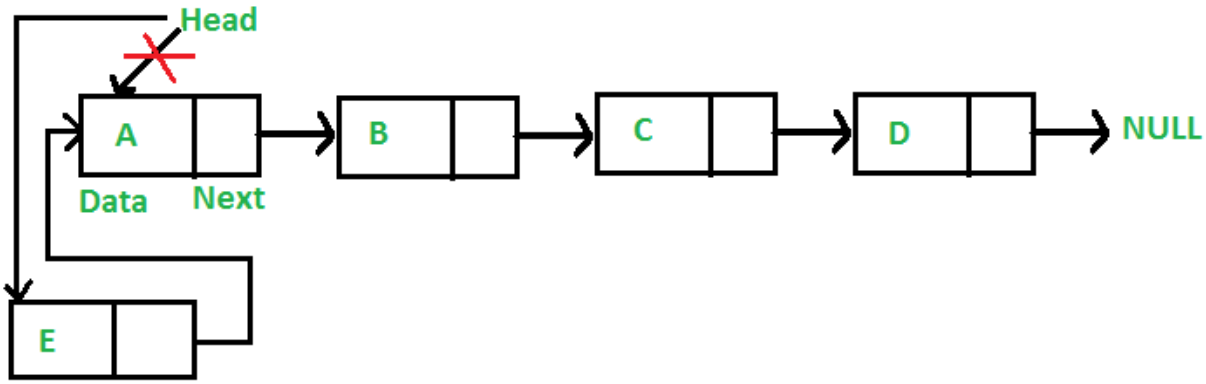
- **Remarque:**

La liste est connue par l'adresse du premier élément si cette adresse n'est pas mémorisée la tête de la liste disparait, donc il faut toujours avoir la tête de la liste mémorisée.

3.4.1 Ajout d'éléments au début de la liste

- Dans cette fonction on veut insérer un élément au début de la liste.

1) Déclarer une fonction qui prend comme paramètres la tête de la liste et la nouvelle valeur à insérer, et retourne la nouvelle tête de la liste.
2) Créer le nouvel élément en utilisant l'allocation dynamique.
3) Affecter la valeur reçue en paramètre au nouvel élément crée.
4) Comme on veut que ce nouvel élément soit en premier de cette liste, on fait le chainage vers la tête par l'instruction **`NouvelElement→suivant = tete.`**

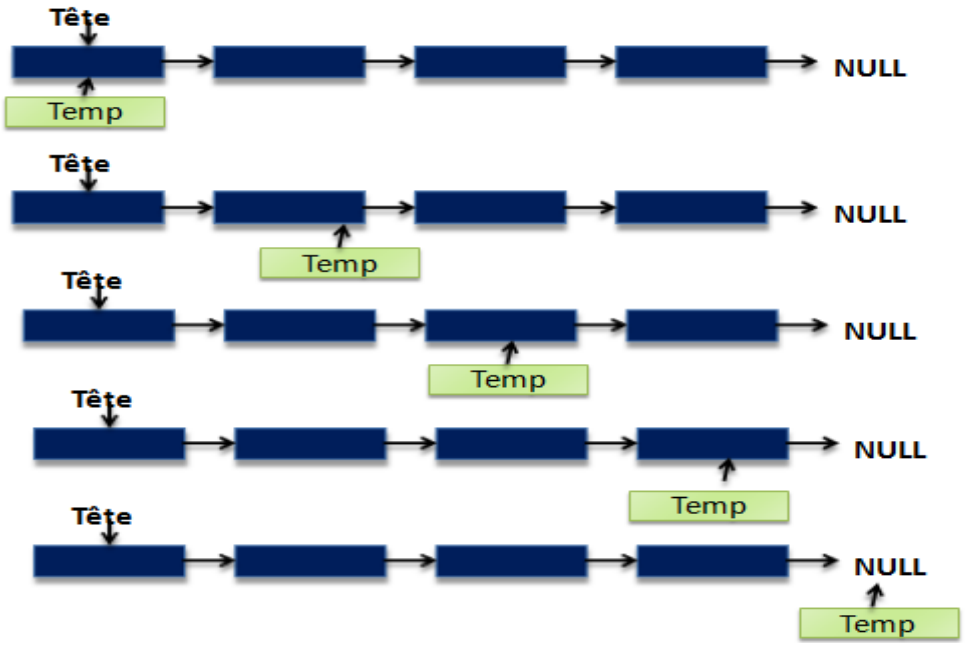



```
Maillon * ajouterEnTete(Maillon * tete, int valeur)
{
    /* créer un nouvel élément */
    Maillon * nouvelElement = malloc(sizeof(Maillon));
    if (nouvelElement == NULL)
    {         printf("Allocation échouée");
              exit(1);
    }

  /* on assigne la valeur au nouvel élément */
    nouvelElement->donnee = valeur;

    /* on assigne l'adresse de l'élément suivant au nouvel
élément */
    nouvelElement->suivant = tete;

    /* on retourne la nouvelle liste, c.-à-d. le pointeur sur le
premier élément */
    return nouvelElement;
}
```

3.5 Affichage d'éléments de la liste.

❖ Pour afficher les éléments de la liste :

1) Créer une variable temporaire qui pointe sur la tête.
2) Passer d'un élément à un autre en utilisant temporaire = temporaire ->**suivant.**



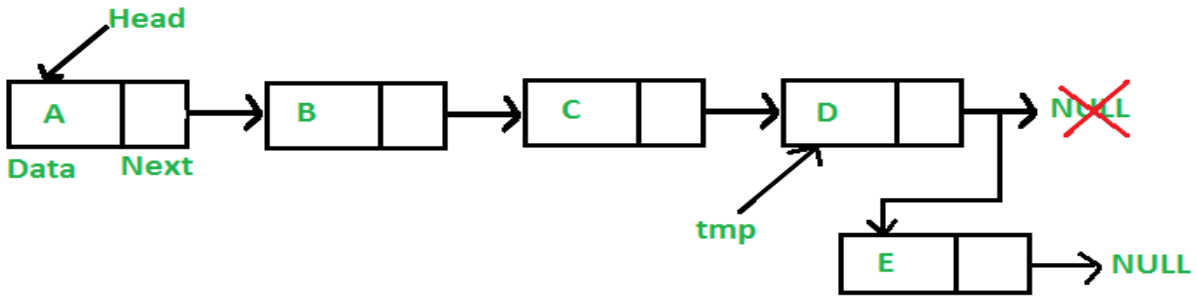

```
void AfficheListe(Maillon* tete)
{
     /* créer un Maillon temporaire qui pointe sur le début de la liste */
      struct Maillon *temp = tete;

     /* vérifier si c'est pas null */
      while (temp != NULL)
     {
       printf(" %d ", temp->donnee);
      /* pointer sur le maillon suivant */
       temp = temp->suivant;
      }
}
```

3.6 Rechercher un élément dans la liste.

```
bool RechercheListe(struct Maillon* tete, int n )
   {
       struct Maillon *temp = tete;

       while (temp != NULL)
       {

       if (temp->donnee == n) {return true; }

       temp = temp->suivant;
       }

  return false;
  }
```

3.7 Ajouter un élément à la fin de la liste.

❖ Pour ajouter un élément à la fin de la liste il faut parcourir la liste jusqu'au dernier élément qui pointe sur NULL. Puis pointer le suivant du dernier élément sur le nouvel élément créé.

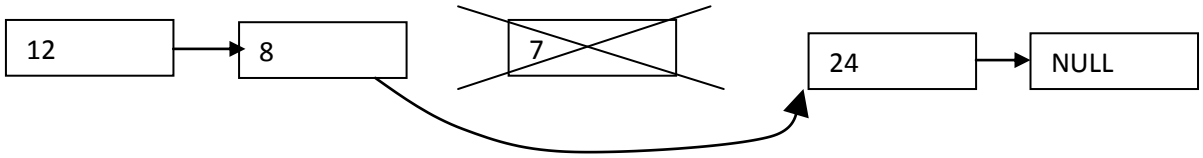



```
Maillon * ajouterEnFin(Maillon * tete, int valeur)
{
    /* On crée un nouvel élément */
    Maillon* nouvelElement = malloc(sizeof(Maillon));
    if (nouvelElement == NULL)
    {         printf("Allocation échouée");
              exit(1);
    }
    /* On assigne la valeur au nouvel élément */
    nouvelElement->donnee= valeur;
    /* On ajoute en fin, donc aucun élément ne va suivre */
    nouvelElement->suivant = NULL;

     if(tete == NULL)
    {
     /* Si la liste est vide il suffit de renvoyer l'élément créé */
        return nouvelElement;
    }
    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 */

        Maillon* temp=tete;
        while(temp->suivant != NULL)
        {
            temp = temp->suivant;
        }
        temp->suivant = nouvelElement;
        return tete;
    }
}
```

3.8 Supprimer un élément de la liste.

- Pour supprimer un élément, il faut premièrement le chercher, ensuite selon sa position dans la liste réaliser le traitement convenable.
    - Si l'élément à supprimer est le premier dans la liste (la tête), il faut un pointeur sur le deuxième élément **(nextElement)** pour le retourner ensuite, car il deviendra la nouvelle tête de la liste. Puis supprimer l'élément (le libérer avec la fonction free).
    - Si l'élément n'est pas le premier, il faut parcourir la liste avec deux pointeurs ; «Pour ne pas créer un autre pointeur temporaire, en utilisera **(nextElement**) pour pointer sur l'élément courant ; et **(precedent)** pour pointer sur l'élément précédent. Une fois l'élément est trouvé, il faut concaténer le précédent avec le suivant de l'élément courant (à supprimer).

precedent →suivant = nextElement→suivant.




```
Maillon * supprimerElement(Maillon * tete, int valeur)
{
       /* Si la liste est vide : ne rien faire */
        if (tete == NULL ) { return NULL;}

      /* Creer un pointeur sur le premier élément */
        Maillon *precedent = tete;
       /* Creer un pointeur sur le deuxième élément */
       Maillon *nextElement = tete->suivant;

       /* Si la valeur est en tête de la liste */
       if(tete->donnee == valeur )
       {
              free(tete);
                return nextElement;
       }

       /* Si la valeur N'est pas en tête de la liste */
        while (nextElement != NULL)
```

```
        {
              if(nextElement->donnee == valeur)
              {
                /* concatener le précédent et le suivant */
                   precedent->suivant =nextElement->suivant;
                   free(nextElement);
                   return tete;
              }

/* On déplace nextElement (mais precedent garde l'ancienne valeur de
NextElement */
           precedent = nextElement;
          nextElement = nextElement-> suivant ;
        }

        return tete;
    }
```

3.9 Exemple de programme principal

La liste est connue par l'adresse du premier élément si cette adresse n'est pas mémorisée la tête de la liste disparait, donc il faut toujours avoir la tête de la liste mémorisée. C'est pourquoi il suiffait de déclarer un pointeur de type structure utilisé dans la liste (Maillon) et l'initialiser par NULL.

```
Struct Maillon * tete = NULL;
```

Ensuite il ne reste que l'appel aux différentes fonctions précédemment déclarées.

```
int main()
{
    /* déclaration d'un pointeur sur le début de la liste*/
    Maillon * Tete = NULL;
          /* Ajouter au début */
    Tete =  ajouterEnTete(Tete,15);
    Tete =  ajouterEnTete(Tete,14);
          /* Ajouter à la fin */
    Tete = ajouterEnFin(Tete,25);
          /* Supprimer un élément */
    Tete = supprimerElement(Tete,15);
          /*Afficher la liste */
    AfficheListe(Tete);
}
```