Chapitre deux

Les structures de données dynamiques

I. L'allocation et libération de mémoire dynamique

La fonction **malloc()** qui veut-dire allocation de mémoire permet de demander au système d'exploitation une réservation d'un espace mémoire durant l'exécution d'un programme l'appel de cette fonction affecte à un pointeur l'adresse de cette espace. La déclaration de cette fonction se trouve dans le fichier stdlib.h:

```
Type *Pointeur = malloc(int nb * sizeof(type));
```

Exemple 1 :

```
int *p ;
p=malloc(sizeof(int)) ; //rappel sizeof(int)==4
```

Cette instruction à l'exécution du programme à l'appel de la fonction malloc(), le système d'exploitation permet de réserver un espace mémoire de (4) octets disponibles (consécutifs) et l'adresse de cet espace est renvoyée comme valeur à p. Bien entendu cela se déroule correctement s'il reste suffisamment de places disponibles en mémoire, sinon la valeur NULL est retourné.

La fonction **sizeof(type)** renvoie le nombre d'octets nécessaires pour représenter une valeur d'un certain type.

```
printf("char : %d octets\n", sizeof(char));  // char 1 octets
printf("int : %d octets\n", sizeof(int));       //  int   4 octets
```

Exemple 2 : un tableau dont la réservation est dynamique

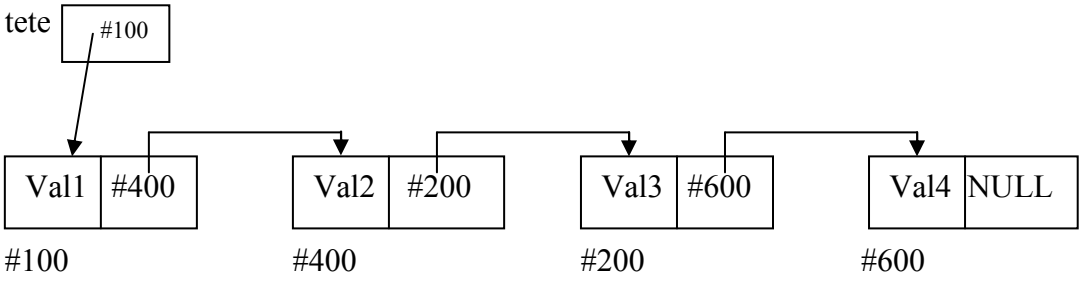
```
 main()
{
   int nbre = 0, i = 0;
   int *Tab = NULL; // Ce pointeur va servir de tableau après l'appel de  malloc()
   // On demande le nombre
   printf("Combien est la valeur de nbre ? ");
   scanf("%d", &nbre);
   if (nbre > 0) // Il faut qu'il ait au moins un
   {  Tab = malloc(nbre * sizeof(int)); // On alloue de la mémoire pour le tableau
      if (Tab == NULL) // On vérifie si l'allocation a marché ou non
      {   exit(0); // On arrête tout
      }
  for (i = 0 ; i < nbre ; i++)
      {  printf("%d ? ", i + 1);
         scanf("%d", &Tab[i]);
      }
      printf("\n\le tableau est\n");
      for (i = 0 ; i < nbre ; i++)
      {printf("%d \n", Tab[i]);
      }
      // On libère la mémoire allouée avec malloc(), on n'en a plus besoin
      free(Tab);
 }}
```

free(pointeur) permet au système d'exploitation de récupérer un espace préalablement alloué par malloc(). L'espace dont l'adresse est la valeur de pointeur sera libéré.

II. Les listes chaînées

Définition :

Une liste linéaire chaînée est un ensemble d'emplacements mémoire de même type, **alloués dynamiquement** et **chaînés** entre eux. Schématiquement, on peut la représenter comme suit :



Dans ce schéma la liste est composée de quatre éléments chaînés entre eux qui sont situés aux emplacements mémoires #100, #400, #200 et #600.

Chaque élément de la liste chainée est composée de deux champs qui sont la **valeur utile** (val1, val2, val3 et val4) appelé champ clé et un champ pointeur pour valeur de l'adresse de l'élément suivant dans la liste(#100, #400, #200 et #600). Chaque liste chainée est représentée par un pointeur spécifique (tete) appelé la tête de la liste indiquant l'adresse du premier élément. Le dernier élément de la liste, sa valeur d'adresse du suivant est égale à NULL.
Les valeurs utiles dans la même liste chaînée sont toutes de même type (int, float, char, double, tableau ou struct…).

a-Déclaration de la structure de base d'une liste chaînée d'entier :

1. **La déclaration récursive :**

Dans le schéma précédent nous avons représenté qu'un élément de la liste est composé de deux champs : un champ clé (val) et un champ suivant, donc, une déclaration d'un type structure s'impose. Ce qu'il faut remarquer est que le champ suivant doit avoir comme valeur l'adresse d'un espace de même type que la structure, d'où apparait la déclaration récursive comme suit :

```
struct liste {
              int val ;
             struct liste *suivant ;
            } ;
```

Note tete est un pointeur qui doit pointer le premier élément de la liste donc sa déclaration est :

Struct liste *tete ;

Ou bien en utilisant typedef :

```
typedef struct liste {
                      int val ;
                      struct liste *suivant ;
                      } liste ;
liste *tete ;
```

2. **Un pointeur à une structure et notation:**

Si par exemple on déclare une variable et un pointeur comme suit
liste l ; liste *p ;
p=&l ;
En langage C pour affecter une valeur de 5 au champ val de l en utilisant p :

(*p).val =5 ;

Ou par un opérateur spécifique → de la manière suivante :

P→val=5 ;

b-Description fonctionnelle:

Il est à noter que seule une variable est déclarée pour représenter une liste chaînée et tous les éléments de cette liste verront leur existence durant l'exécution du programme qui permet son implémentation. Il s'agit d'élaborer des fonctions qui assurent l'initialisation d'une liste à vide à sa création la possibilité d'ajouter d'éléments dans la liste, le parcours de la liste pour une recherche, l'affichage d'une liste suppression d'éléments et autres …Nous donnons différentes manières(A,B,C..) d'écrire ces fonctions :

1. **Une liste vide et/ou initialisation à vide (création d'une liste) :**

A) A la déclaration :

```
main()
 {
    liste *tete;
    tete=NULL ;
 }
```

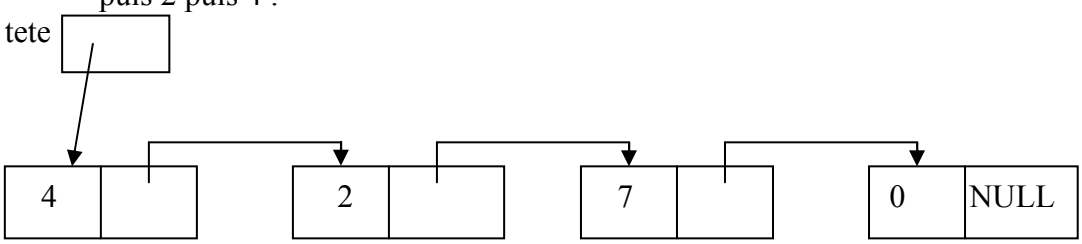
B) Appel d'une fonction init() qui retourne NULL :

```
liste *init()
{return  NULL ;
}
main()
{liste *tete ;
  tete=init() ;
}
```

C) Appel d'une fonction void init(liste**t) qui affecte la valeur NULL :

```
void init(liste **t)
   {*t=NULL ;
   }
main()
{liste *tete ;
  init(&tete) ;
}
```

2. **Ajout d'un élément <u>en tête de la liste</u> :** nous allons faire appel plusieurs fois à une fonction qui va ajouter à la liste, **vide** au départ, les éléments dont les valeurs : 0 puis 7 puis 2 puis 4 :



A) Appel d'une fonction ajout(liste *t,int x) qui retourne une nouvelle valeur pour la tête:

```
liste *init()
      {return  NULL ;
      }
liste *ajout( liste*t, int x)
{liste *nouv;
 nouv=malloc(sizeof(liste))
 nouv→val=x;
 nouv→suivant=t;
 return nouv;
}
main()
{liste *tete ;
  tete=init() ;
   tete=ajout(tete,0) ;
   tete=ajout(tete,7) ;
   tete=ajout(tete,2) ;
   tete=ajout(tete,4) ;
}
```

B) Appel d'une fonction void ajout(liste **t,int x):

```
void init(liste **t)
      {*t=NULL  ;
      }
 void  ajout( liste**t, int x)
 {liste *nouv;
  nouv=malloc(sizeof(liste))
  nouv→val=x;
  nouv→suivant=*t;
  *t= nouv;
}
main()
{liste *tete ;
  init(&tete) ;
   ajout(&tete,0) ;
   ajout(&tete,7) ;
   ajout(&tete,2) ;
   ajout(&tete,4) ;
afficheliste( tete) ;
}
```

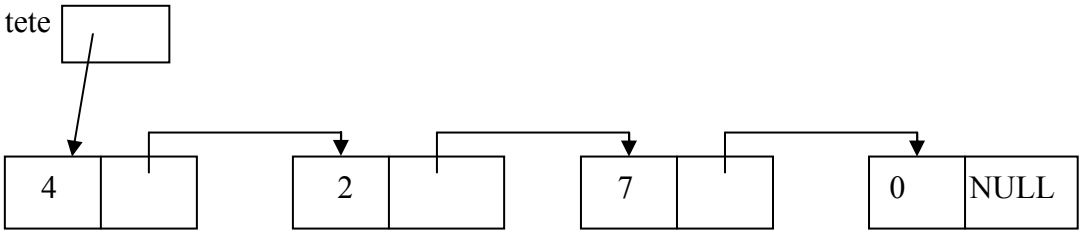
Exercice : Ecrire une fonction qui permet d'ajouter un élément en fin de la liste.

3. Parcourir une liste pour afficher les éléments de la liste :

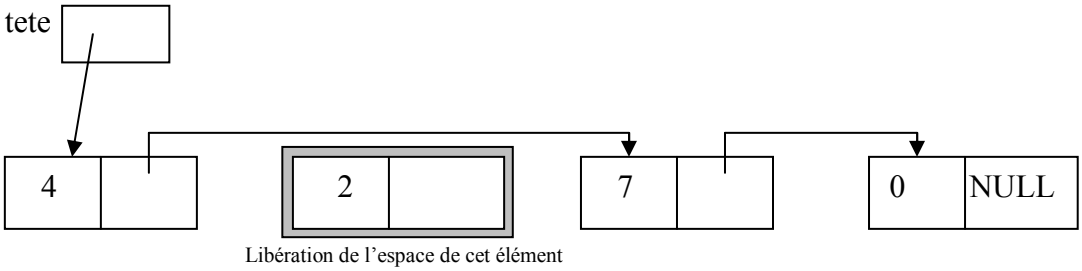
```
void  affiche liste( liste*t)
 {liste *p=t;
  while(p !=NULL)
  { printf(" %d \n", p→val);
    p=p→suivant;
  }
 }
```

4. **Suppression d'un élément d'une liste de valeur x**

Exemple : Soit à supprimer l'élément dont la valeur est égale à 2 de la liste suivante :



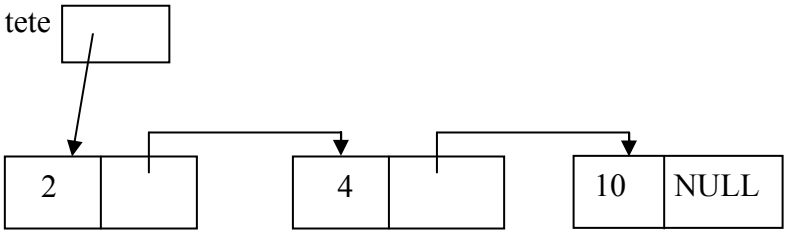
La suppression nous permet d'obtenir la nouvelle liste en libérant l'espace occupée par l'élément dont la valeur est 2 qui sera restitué par le système via la fonction free() :



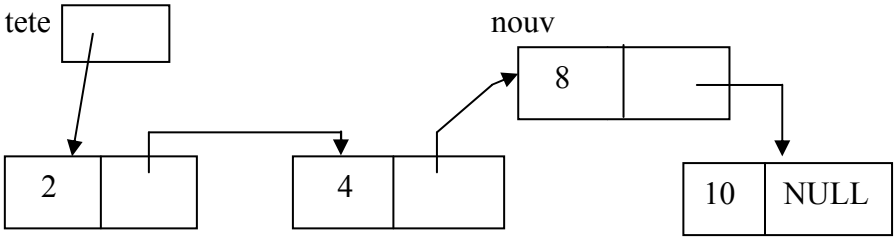

```
void  Supp( liste**t,int x)
    {
      If(*t !=NULL && (*t)→val==x){// suppression en tête de la liste
                                    liste *p=*t ;
                                    *t=(*t)→suivant ;
                                     free(p) ;
                                   }
                                 else{ //suppression au milieu de la liste
                                       liste *prec; p=*t ;
                                       while(p !=NULL && p→val != x)
                                        { prec=p;
                                          p=p→suivant;
                                        }
                                       if (p !=NULL){ prec→suivant=p→suivant ;
                                                      free(p) ;
                                                    }
                                     }
      }
```

5. **Fonction qui permet d'ajouter un élément de valeur x dans une liste ordonnée.**

Exemple : pour insérer dans la liste suivante un élément d'une valeur 8 :



On doit allouer un nouvel espace et changer le champ suivant de l'élément qui doit précéder ce nouveau.



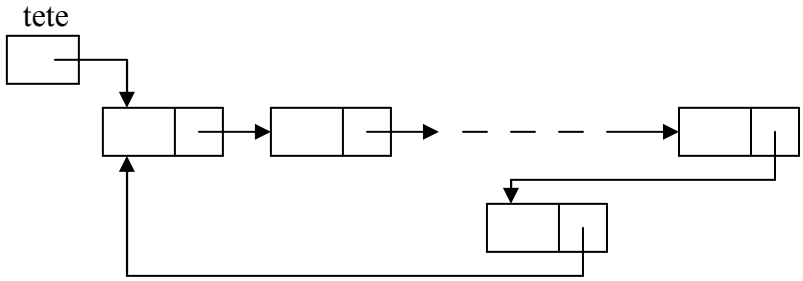

```
Void AjoutListOrd(liste **t, int x)
{
  If((*t !=NULL && (*t)→val>=x)|| *t==NULL)
                                {// ajout en tête de la liste
                                    liste *nouv;
                                    nouv=malloc(sizeof(liste))
                                     nouv→val=x;
                                     nouv→suivant=*t;
                                     *t= nouv;
                                  }
                                else{  // ajout au milieu ou en fin de la liste
                                        liste *prec;  liste p=*t ;
                                                while(p !=NULL && p→val < x)
                                                 { prec=p;
                                                    p=p→suivant;
                                                  }
                                        liste *nouv;
                                        nouv=malloc(sizeof(liste))
                                        nouv→val=x;
                                        nouv→suivant=p ;
                                        prec→suivant=nouv ;// doit modifier le champ 'suivant' de l'élément précédent
                                      }

 }
```

III D'autres types de listes chaînées :

1. Liste circulaire :

Une liste est dite circulaire car le dernier élément de la liste a pour suivant le premier dans la liste : c'est-à-dire le champ pointeur du dernier a la même valeur que tête.

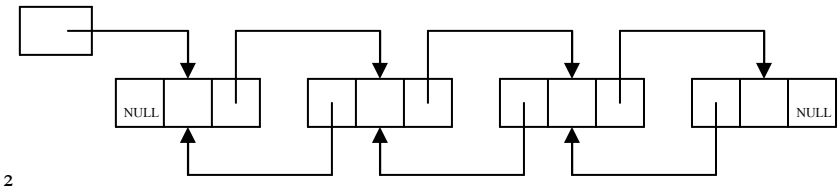


Exercice

Ecrire une fonction qui permet de transformer une liste chainée simple en une liste chainée circulaire.

2- Liste bidirectionnelle : (liste doublement chaînée)

Chaque élément d'une ***liste doublement chaînée*** *L* est un objet contenant un champ clé pour la valeur utile et deux autres champs pointeurs : *suivant* et *précédent*.



2

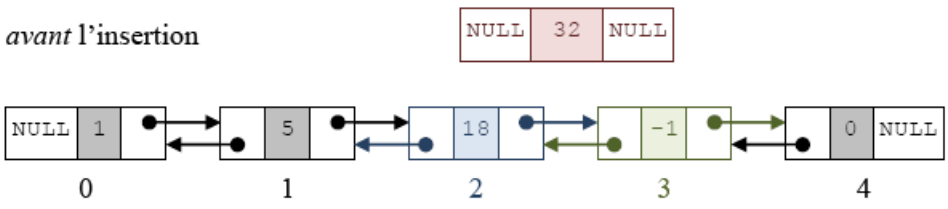
Exercice

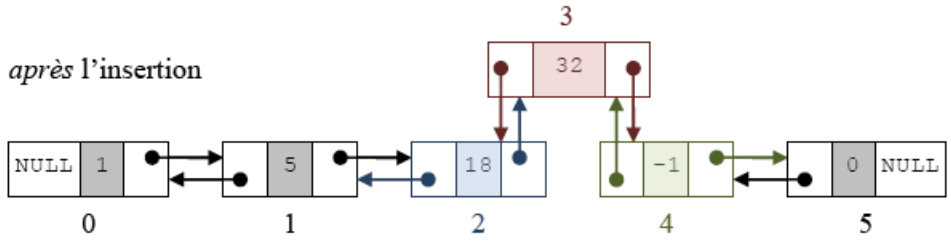
1-Donner la déclaration de la structure représentant les éléments de cette liste dont la valeur clé est entier.

```
typedef struct bidliste
  { int val ;
    struct bidliste *precedent;
    struct bidliste *suivant;
  } bidliste;
```

Le pointeur `precedent` du *premier* élément d'une liste pointera vers `NULL`, exactement comme le pointeur `suivant` du *dernier* élément.

Exemple : Fonction d'insertion d'un élément dans une liste bidirectionnelle après les trois premiers éléments :






```
int insert (bidliste **t, int x, int pos) // cette fonction retourne 0 si insertion réussi -1 sinon
{ bidlist *nouv,*p;
  int erreur = 0;
            nouv=malloc(sizeof(liste))
            nouv→val=x;

  if(pos==0)    // si on insère en position 0, on doit modifier la tete
               if(*t != NULL) {
                                  nouv→suivant=*t;
                                  nouv→precedent=Null;
                                  (*t)->precedent = nouv;
                                   *t=nouv
                                }
                               else
                                  {     nouv→suivant=NULL;
                                        nouv→precedent=NULL;
                                     *t=nouv
                                  }
              else
              {   int i=0 ; p=*t;
                  While(p!=NULL &&i<pos)
                    {   i++;
                        p=p→suivant;
                    }
                    if (i==pos){   // si la position existe
                                  bidlist *prec=p→precedent ;
                                  nouv→precedent=prec ;
                                  nouv→ suivant=p;
                                  p→precedent=nouv;
                                  prec→suivant=nouv;
                                 }
                               else // sinon une erreur de position qui ne peut être assurée
                                    { erreur=-1
                                    }
  return erreur;
}
```