LA STRUCTURE DYNAMIQUE

Gestion de la mémoire centrale :

Le nombre des éléments d'un tableau est limité par une constante. On peut créer un tableau en cours d'un programme. Pour pouvoir réaliser cette création, on utilise ce qu'on appelle l'allocation dynamique de la mémoire. Cette technique permet de créer des tableaux dont la taille mémoire est variable en fonction des besoins et de pouvoir libérer cet espace mémoire après son utilisation.
Pour créer un tableau en cours d'exécution , il faut réserver un emplacement dans la mémoire centrale. Le programme indique la taille de l'emplacement mémoire (le nombre d'octets), et lors de l'exécution du programme, le système réserve un emplacement mémoire et donne l'adresse de cet emplacement. Cette opération s'appelle l'allocation dynamique. Les fonctions qui réservent de l'espace en C sont malloc et calloc. On dit allocation dynamique parce que l'allocation de mémoire a lieu à mesure des besoins, par rapport à l'allocation statique de tableaux.

Allocation avec malloc :

La fonction malloc alloue un certain nombre d'octets (réservation d'octets pour une utilisation par le programme. Le nombre d'octets est passé en paramètre à la fonction malloc. La fonction malloc retourne l'adresse du premier octet réservé. Cette adresse est mémorisée dans un pointeur. La fonction sizeof donne le nombre d'octets nécessaires à une variable de type float, int, char, etc, et on multiplie par le nombre d'éléments du tableau.

Après utilisation de la mémoire, la mémoire doit impérativement être libérée avec la fonction free. Cette fonction prend en paramètre l'adresse du bloc mémoire qui doit être libéré. Lors de l'appel à cette fonction, la mémoire est rendue au système d'exploitation qui peut la réemployer pour une autre utilisation.

Exemple :

```
… ;
int n ;
… ;
float  *t ;
t = (float *)malloc(n*sizeof(float)) ;  /* réservation d'espace mémoire pour n
                                           float */
… ;
free(t) ; // libération de l'espace précédemment créé
```

Listes chainées

Les listes chainées se présentent comme une alternative aux tableaux pour le stockage de certaines données. En effet, dans certaines situations, l'usage des tableaux peut se révéler inefficace.

Vous devriez savoir que lorsque vous créez un tableau, vous connaissez sa taille à l'avance, que ce soit avec l'utilisation ou non de malloc. Ainsi, vous vous retrouvez limités par la taille de votre tableau. Il est cependant possible à tout moment d'agrandir un tableau, mais ceci n'est pas la meilleur solution envisageable. Pour pallier à cette limitation, nous allons donc utiliser des listes chainées. Alors que les éléments d'un tableau sont contigües en mémoire, les éléments d'une liste chainées sont quant à eux tous reliés via une série de pointeurs.

Une liste chainée est un ensemble de cellules liées entre elles par des pointeurs. Chaque cellule est une structure contenant les champs suivants :

- **Une ou plusieurs informations ;**
- **Un pointeur sur la cellule suivante.**

On accède à la liste par un pointeur de tête de liste qui pointe sur la première cellule, puis on parcours la liste d'une cellule à une autre en suivant le pointeur de la liste courante. Le dernier pointeur son suivant est NULL (pointe sur rien), ce qui indique la fin de la liste.

Déclaration d'une liste simplement chainée linéaire :

On utilise une liste d'entiers.

```
typedef struct cellule
            {int inf ;
              struct cellule * suiv;
             } typecellule;

typecellule  * tete;
```

Les listes doublement chainées :

Lorsque chaque élément d'une liste chainée pointe vers l'élément suivant, nous parlons de liste simplement chainée. Lorsque chaque élément d'une liste pointe à la fois vers l'élément suivant et précédent, nous parlons alors de liste doublement chainée. Donc une liste chainée nous permet de stocker un nombre inconnu d'éléments.

Chaque élément d'une liste doublement chainée contient :

- **Une donnée (ici un entier)**
- **Un pointeur vers l'élément suivant (NULL si l'élément suivant n'existe pas)**
- **Un pointeur vers l'élément précédent (NULL si l'élément précédent n'existe pas)**

Représentation d'une liste à double sens :

Nous allons tout simplement utiliser une structure. En effet, en langage C, les structures sont très pratiques pour créer de nouveaux types de données. Pour être plus précis, nous allons utiliser exactement deux structures.

Voici la première :

```
struct cellule
{
   int inf;
   struct cellule *suiv;
   struct cellule *prec;
};
```

Cette première structure va nous permettre de représenter une 'cellule' (élément) de notre liste chaînée. Nous pouvons alors voir que chaque élément de notre liste contiendra un élément de type int. D'autre part :

- **suiv pointera vers l'élément suivant (ou NULL s'il s'agit du dernier élément de la liste)**

- **prec pointera vers l'élément précédent (ou NULL s'il s'agit du premier élément)**

Les liens entre les différents éléments de notre liste chaînée sont donc assurés par nos deux pointeurs suiv et prec.

Pour représenter notre liste chaînée , nous utiliserons une deuxième liste :

```
typedef struct dliste
{
   int long;
   struct cellule *queue;
   struct cellule *tete;
} Dliste;
```

```
Dliste *list = NULL;  // Déclaration d'une liste vide
```

<u>Allouer une nouvelle liste</u> :

Avant de pouvoir commencer à utiliser notre liste chaînée, nous allons créer une fonction nous permettant d'allouer de l'espace mémoire pour notre liste chaînée. La fonction retournera la liste chaînée nouvellement créée.

```
Dliste *list_new()
{
   Dliste *newl = malloc(sizeof (Dliste));
   if (newl != NULL)
   {
      newl->long = 0;
      newl->tete = NULL;
      newl->queue = NULL;
   }
   return newl;
}
```

Implémentation d'une pile sous forme de liste chainée :

On implémente une pile d'entiers en utilisant le type synonyme suivant :

```
typedef struct cellule
            { int inf ;
              struct cellule * suiv;
             }typecellule;
typedef typecellule * pile;

pile initialiser(pile p)
{return NULL;}

int vide(pile p)
{return p==NULL;}   ou  {return (p==NULL)?1:0;}

int pleine(pile p)
{return 0;}

int accesausommet(pile p, int *nb)
{/*si l'accès est realisè 0 est retouné sinon 1 est envoyé au programme
    appellant */
 if (vide(p))
   return 1;
 *nb = p->inf;
 return 0;
}

pile empiler(pile p, int nb)
{pile q;
 q = (pile)malloc(sizeof(typecellule));
 q->inf = nb;
 q->suiv = p;
 return q;
 }

int depiler(pile *p, int * nb)
{pile q;
 if (vide(*p))
   return 1;
 q = *p;
```

```
 *p = (*p)->suiv;
 free(q);
 return 0; }
```

La gestion des piles par listes chaînées comparée aux tableaux, présente l'énorme avantage que la pile a une capacitè virtuellement illimitée(la limite se situe par rapport à la capacité de la mémoire), la mémoire étant allouée à mesure des besoins.

Implementation d'une file sous forme de liste chainée :

On implémente une file d'entiers en utilisant le type synonyme suivant :

```
typedef struct cellule
            { int inf ;
              struct cellule * suiv;
             }typecellule;



typedef  struct
             {typecellule *tete, *queue;
              }file;


file initialiser(file f)
{f.tete=NULL;
 return f;
 }

int vide(file f)
{return f.tete==NULL;}   ou  {return (f.tete==NULL)?1:0;}

int pleine(file f)
{return 0;}

int accesalatete(file f, int *nb)
{/*si l'accès est realisé 0 est retouné sinon 1 est envoyé au programme
   Appellant */
 if (vide(f))
```

```
   return 1;
 *nb = f.tete->inf;
 return 0;
}



file enfiler(file f, int nb)
{typecellule * q;
 q = (typecellule *)malloc(sizeof(typecellule));
 q->inf = nb;
 q->suiv = NULL;
  if (f.tete == NULL)
    f.tete = q;
  else
     f.queue->suiv = q;
  f.queue = q;
 return f;
 }

int defiler(file *f, int * nb)
{typecellule * q;
 if (vide(*f))
   return 1;
 *nb  = f->tete->inf;
  q = f->tete;
  f->tete = f->tete->suiv;
 free(q);
 return 0;
 }
```