Introduction

Les listes chaînées sont, comme les tableaux des structures de données linéaires, c'est à dire qui permettent de relier des données en séquence (on peut numéroter les éléments).
Mais contrairement aux éléments d'un tableau, les éléments d'une liste chaînée ne sont pas forcément contigus en mémoire et ils sont reliés grâce à des pointeurs. Ces éléments sont ajoutés au fur et à mesure par allocation dynamique lors de l'exécution. Les listes chaînées sont ainsi qualifiées de structures de données **dynamiques.**

I. Les listes bidirectionnelle

1. Définition

Une liste bidirectionnelle est une liste doublement chainée avec deux (2) voies d'accès : Tête et Queue.

Tête : contient l'adresse du premier élément de la liste.

Queue : contient l'adresse du dernier élément de la liste.

2. Déclaration

Exemple :

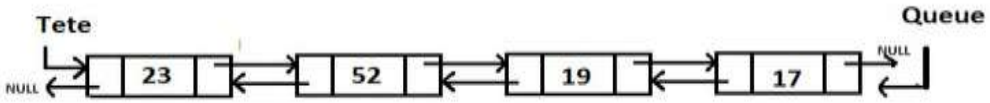
Définir la structure d'une liste bidirectionnelle d'entier puis déclarer la variable Tête et Queue à l'associer.

```
typedef struct liste*  listebi;
struct liste{
int info;
listebi suiv;
listebi prec;

};
listebi * T=(listebi*)malloc(sizeof(listebi));
listebi * Q=(listebi*)malloc(sizeof(listebi));
```

3. Manipulation de la liste bidirectionnelle

Soit la liste bidirectionnelle suivante :



Tete contient l'adresse du premier élément de la liste.

Tete->info = 23

Tete->suiv = L'adresse du 2e élément

Tete->suiv->info = 52

Q->suiv = NULL

Q->prec->info = 19

4. Création d'une liste bidirectionnelle

Pour créer une liste bidirectionnelle, il faut trois (3) pointeurs : **Tete, pc et Q.**

- « **Tete** » permet de sauvegarder l'adresse du premier élément de la liste.
- « **pc** » permet de créer tous les éléments à placer dans la liste.
- « **Q** » va jouer 2 rôles : un rôle de marquage du dernier élément et un rôle de chainage.

Exemple :

```
listebi pc=( listebi)malloc(sizeof(liste));
    pc->info=emp;
    pc->suiv=NULL;
    pc->prec=NULL;
   if(T==NULL){
       T=pc;
   }else{
      pc->prec= Q;
      Q->suiv=pc;
```

```
}
    Q=pc;
```

5. Parcours d'une liste bidirectionnelle

Pour parcourir une liste bidirectionnelle, il faut déclarer un pointeur de parcours à initialiser à **Tete** de la liste à parcourir. Ce pointeur sera associé à une boucle. La boucle peut être « **do while** » ou «**while** ».

Exercice d'application :

```
listebi  p=*T;
while(p!=NULL){
 cout<< p->info;
      p=p->suiv;
}
```

6. Les opérations applicables aux listes

Il est possible d'appliquer plusieurs types de traitements aux valeurs d'une liste. Les traitements de base sont en fonction des types de valeur des éléments de la liste mais également il est possible de faire la recherche de valeur, l'ajout, la modification, la suppression, l'intersection, la fusion etc.

Exercice d'application :

```
cout<< ("1-Ajout En fin");
cout<< ("2-Ajout En Tete");
cout<< ("3-Ajout à une position");
cout<< ("4-Affichage");
cout << ("5-Creation");
cout<< ("6-Supprimer En fin");
cout<< ("7-Supprimer En Tete");
cout<< ("8-Supprimer à une position");
cout << ("9-Quitter");
```