Université de Yaoundé 1

Département d'informatique

Informatique Niveau 2

University of Yaoundé 1

Departement of informatique

Informatique Level 2

Dossier algorithmique : Travaux Pratique d'Algorithmique et Structures de Données.

Membres du groupe	Matricule
BOUKALA BONOKO FRANCK GABRIEL	24G2765
ZE MARTHE ANGELIQUE	24F2763
TSAGUE KENGNI PEGUY HERMAN	23U2994
MESSINA ADOUBE LYSA EMMANUELLE	24G2901
TSAGUE NDIFO DELPHIN BERTO	23U2502
DOUANLA NGUEYO CHRISTIAN FRANCK	24G2051

Sous la supervision du

Pr MELATAGIA

Enoncé : Exercice INF 231 du jeudi 25 septembre 2025.

Écrire en C

1. Lire un élément et supprimer toutes les occurence dans la liste
2. Insertion d'un élément dans une liste simplement chaîne trié
3. Insertion d'un élément dans une liste doublement chaîne trié
4. Insertion en tête et en queue dans une liste simplement chaîne circulaire
5. Insertion en tête et en queue dans une liste doublement chaîne circulaire

Partie 1 : Liste simplement chaînée avec suppression d'éléments

Introduction

Dans le cadre de l'étude des structures de données dynamiques, nous allons travailler sur la liste simplement chaînée.
L'objectif est d'implémenter une liste chaînée permettant :

- l'insertion en tête et en fin,
- la suppression en tête et en fin,
- l'affichage,
- et la suppression de toutes les occurrences d'un élément donné.

Analyse

Une liste simplement chaînée est constituée de nœuds reliés entre eux par des pointeurs.
Chaque nœud contient :

- une valeur entière (val),
- un pointeur vers le suivant (suiv).

Schéma : **[val | suiv] -> [val | suiv] -> ... -> NULL**

Algorithme en pseudo-code

Structure de donnée

Structure Node

  entier val

  pointeur Node suiv

Fin Structure

createLol(valeur)

```
Fonction createLol(valeur) -> Node*
  allouer mémoire pour newNode
  si allocation échoue alors
    afficher "Erreur allocation"
    retourner NULL
  newNode.val <- valeur
  newNode.suiv <- NULL
  retourner newNode
Fin Fonction
```

Complexité : O(1)

Insertion en tête : InsertTete(&tete, valeur)

```
Procédure InsertTete(&tete, valeur)
  newNode <- createLol(valeur)
  si newNode == NULL alors
    retourner   // échec allocation
  newNode.suiv <- tete
  tete <- newNode
Fin Procédure
```

Complexité : O(1)

Insertion en fin : InsertFin(&tete, valeur)

```
Procédure InsertFin(&tete, valeur)
  newNode <- createLol(valeur)
  si newNode == NULL alors
    retourner
  si tete == NULL alors
    tete <- newNode
    retourner
```

```
    temp <- tete
    Tant que temp.suiv != NULL faire
        temp <- temp.suiv
    Fin Tant que
    temp.suiv <- newNode
Fin Procédure
```

Complexité : O(n) (n = longueur de laliste)

Suppression de toutes les occurrences d'un élément : display(tete)

Procédure display(tete)

```
    si tete == NULL alors
        afficher "Liste vide!"
        retourner
    temp <- tete
    afficher "Liste : "
    Tant que temp != NULL faire
        afficher temp.val " -> "
        temp <- temp.suiv
    Fin Tant que
    afficher "NULL"
Fin Procédure
```

Suppression d'occurrences en tête de liste : suppression_tete(&tete)

Procédure suppression_tete(&tete)

```
    si tete == NULL alors
        afficher "Liste déjà vide!"
        retourner
    temp <- tete
    tete <- tete.suiv
    free(temp)
    afficher "Élément en tête supprimé."
Fin Procédure
```

Complexité : O(1)

Suppression d'occurrence en fin de liste : deleteEnd(&tete)

Procédure deleteEnd(&tete)

```
    si tete == NULL alors
        afficher "Liste déjà vide!"
        retourner
    // cas un seul élément
    si tete.suiv == NULL alors
        free(tete)
        tete <- NULL
        afficher "Élément en fin supprimé."
```

```
    retourner
  prev <- NULL
  temp <- tete
  Tant que temp.suiv != NULL faire
    prev <- temp
    temp <- temp.suiv
  Fin Tant que
  // temp pointe sur le dernier, prev sur l'avant-dernier
  prev.suiv <- NULL
  free(temp)
  afficher "Élément en fin supprimé."
Fin Procédure
```

Complexité : O(1)

Programme principale : main

Fonction main()

```
  tete <- NULL
  faire
    afficher menu:
      1. Insertion en tête
      2. Insertion en fin
      3. Suppression en tête
      4. Suppression en fin
      5. Affichage de la liste
      6. Quitter
      (optionnel: 7. Supprimer toutes les occurrences d'une valeur)
    lire choix b

    selon b faire
      case 1:
        afficher "Insertion en tête"
        lire a
        InsertTete(&tete, a)
        display(tete)
      case 2:
        afficher "Insertion en fin"
        lire a
        InsertFin(&tete, a)
        display(tete)
      case 3:
        afficher "Suppression en tête"
        suppression_tete(&tete)
        display(tete)
```

```
        case 4:
            afficher "Suppression en fin"
            deleteEnd(&tete)
            display(tete)
        case 5:
            afficher "Affichage de la liste"
            display(tete)
        case 6:
            afficher "Au revoir!"
            // sortir de la boucle
        case 7:   // facultatif : si tu ajoutes SupprimerOccurrences au menu
            afficher "Entrer la valeur à supprimer (toutes occurrences):"
            lire a
            SupprimerOccurrences(&tete, a)
            display(tete)
        défaut:
            afficher "Choix invalide. Veuillez réessayer."
    fin selon
  tant que b != 6

  // libération avant fin du programme
  FreeAll(&tete)
  retourner 0
```

Fin Fonction

Exemple d'exécution :

Insertion en tête : 5

Insertion en fin : 3

Insertion en fin : 5

Insertion en fin : 7

Affichage : 5 -> 3 -> 5 -> 7 -> NULL

Suppression de toutes les occurrences de 5

Résultat : 3 -> 7 -> NULL

Conclusion

Ce travail nous a permis de comprendre le fonctionnement des listes chaînées ainsi que la gestion dynamique de la mémoire en C.
La suppression d'éléments particuliers met en évidence l'importance de bien manipuler les pointeurs pour éviter les fuites de mémoire.

Partie 2 : Gestion d'une liste simplement chaînée triée en C

Objectif

L'objectif de ce programme est de construire une liste simplement chaînée qui reste triée par ordre croissant à chaque insertion d'un élément.
L'utilisateur entre des nombres entiers, et la liste est affichée dans l'ordre trié automatiquement.

Structures de données utilisées

- **Structure d'un nœud (node)**
  Chaque élément de la liste contient :
- un champ data (entier)
- un pointeur suivant vers le nœud suivant

Pseudo-code des fonctions

Création d'un nouveau nœud : Fonction creerNoeud(data)

Entrée : un entier data
Sortie : un pointeur vers le nouveau nœud

```
Début
   Allouer dynamiquement un nœud
   Si allocation échoue
      Afficher "Erreur avec malloc"
      Retourner NULL
   Fin Si
   Mettre data dans le champ data
   Mettre NULL dans suivant
   Retourner le pointeur vers le nouveau nœud
Fin
```

Affichage de la liste : Fonction afficherListe(tete) :

Entrée : pointeur vers la tête de la liste
Sortie : affichage des éléments de la liste

```
Début
   courant ← tete
   Tant que courant ≠ NULL faire
      Afficher courant.data
      courant ← courant.suivant
```

```
   Fin Tant que
   Afficher "NULL"
Fin
```

Trier le tableau : Fonction insererTrie(&tete, data)

Entrée : pointeur vers la tête de la liste, entier data
Sortie : liste mise à jour avec data inséré au bon endroit

```
Début
   Créer un nouveau nœud avec data

   Si liste vide OU (data ≤ tete.data) alors
      nouveauNoeud.suivant ← tete
      tete ← nouveauNoeud
   Sinon
      courant ← tete
      Tant que courant.suivant ≠ NULL ET courant.suivant.data < data faire
         courant ← courant.suivant
      Fin Tant que
      nouveauNoeud.suivant ← courant.suivant
      courant.suivant ← nouveauNoeud
   Fin Si
Fin
```

Programme principale : Fonction main()

```
Début
   Initialiser la liste à NULL
   Afficher un message pour entrer des entiers
   Lire valeur
   Tant que valeur ≠ 0 faire
      insererTrie(&liste, valeur)
      Lire valeur
   Fin Tant que
   Afficher la liste
Fin
```

Programme source en C :

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct node{
   int data;
```

```
    struct node *suivant;  // Un seul pointeur pour liste simplement chaînée
} node;

// Fonction pour créer un nouveau nœud
node *creerNoeud(int data){
    node *nouveauNoeud = (node*)malloc(sizeof(node));
    if(!nouveauNoeud){
        printf("Erreur avec malloc\n");
        return NULL;
    }
    nouveauNoeud->data = data;
    nouveauNoeud->suivant = NULL;  // Uniquement le pointeur suivant, pas de prec
    return nouveauNoeud;
}

void afficherListe(node *tete){
    node *courant = tete;
    while(courant != NULL){
        printf(" %d -> ", courant->data);
        courant = courant->suivant;
    }
    printf("NULL\n");
}

// Fonction pour insérer un élément dans une liste simplement chaînée triée
void insererTrie(node **tete, int data){
    node *nouveauNoeud = creerNoeud(data);

    // Si la liste est vide ou si le nouveau nœud doit être en tête
    if(*tete == NULL || (*tete)->data >= nouveauNoeud->data){
        nouveauNoeud->suivant = *tete;
        *tete = nouveauNoeud;
    }
    else{
        node *courant = *tete;
        // Trouver la position appropriée
        while(courant->suivant != NULL && courant->suivant->data < nouveauNoeud->data){
            courant = courant->suivant;
        }
        // Insérer le nouveau nœud
        nouveauNoeud->suivant = courant->suivant;
        courant->suivant = nouveauNoeud;
    }
}

int main(){
    node *maListe = NULL;
```

```
    printf("\n=== Liste simplement chaînée triée ===\n");
    printf("Entrer des entiers (0 pour terminer) : \n");
    int valeur;
    scanf("%d", &valeur);
    while(valeur != 0){
        insererTrie(&maListe, valeur);
        scanf("%d", &valeur);
    }
    printf("Liste triée: ");
    afficherListe(maListe);

    return 0;
}
```

Exemple d'exécution :

```
=== Liste simplement chaînée triée ===
Entrer des entiers (0 pour terminer) :
34
56
78
98
0
Liste triée:  34 ->  56 ->  78 ->  98 -> NULL
```

Conclusion :

Ce travail nous a permis de comprendre le fonctionnement des listes simplement chaînées triées ainsi que la gestion dynamique de la mémoire en C.
Le trie d'élément d'une liste.

Partie 3 : Gestion d'une liste doublement chaînée triée en C

Objectif

L'objectif de ce programme est de créer et de manipuler une **liste doublement chaînée** dans laquelle chaque élément est inséré automatiquement à la bonne position pour que la liste reste **triée en ordre croissant**.
Chaque nœud pointe à la fois vers :

- son **suivant**
- son **précédent**

Structures de données utilisées

- **Structure d'un nœud (dnode)**
  Chaque nœud contient :
- un entier data
- un pointeur next vers le nœud suivant
- un pointeur prec vers le nœud précédent

Forme :

```
struct dnode {
   int data;
   struct dnode *next;
   struct dnode *prec;
};
```

Pseudo-code des fonctions

Creation d'un nœud : Fonction createNode(data)

 Opérations effectuées : allocation mémoire, affectations simples (data, next, prec).

 Complexité temporelle : O(1) (temps constant).

 Complexité spatiale : O(1) (un seul nœud est créé).

Entrée : un entier data
Sortie : pointeur vers un nouveau nœud

Début
   Allouer dynamiquement un nœud

```
   Si allocation échoue
      Afficher "erreur malloc"
      Retourner NULL
   Fin Si
   Placer data dans le champ data
   Placer NULL dans next
   Placer NULL dans prec
   Retourner le pointeur du nouveau nœud
Fin
```

Affichage de la liste : **Fonction printflist(head)**

- Parcourt toute la liste une seule fois.
- Si la liste contient n éléments :
- Complexité temporelle : O(n).
- Complexité spatiale : O(1) (aucune structure supplémentaire n'est utilisée).

```
Entrée : pointeur vers la tête de la liste
Sortie : affichage des éléments de la liste

Début
   courant ← head
   Tant que courant ≠ NULL faire
      Afficher courant.data
      courant ← courant.next
   Fin Tant que
   Afficher "NULL"
Fin
```

Trie : **Fonction insertDsorted(&head, data)**

Trois cas possibles :

1. Insertion en tête :
    - Comparaison + quelques affectations.
    - Complexité : O(1).
2. Insertion au milieu ou à la fin :
    - Parcours de la liste jusqu'à trouver la bonne position.
    - Dans le pire cas (insertion en queue), on parcourt n éléments.
    - Complexité temporelle : O(n).
3. Mise à jour des pointeurs next et prec : opérations constantes.

- Complexité spatiale : O(1) (on crée seulement un nœud supplémentaire).

👉 Donc la complexité globale de insertDsorted est O(n) dans le pire cas.

Entrée : adresse de la tête de la liste, entier data

Sortie : liste mise à jour avec data inséré à la bonne position

```
Début
   Créer un nouveau nœud avec data

   Si la liste est vide OU data ≤ head.data alors
      newNode.next ← head
      Si head ≠ NULL alors
         head.prec ← newNode
      Fin Si
      head ← newNode
   Sinon
      courant ← head
      Tant que courant.next ≠ NULL ET courant.next.data < data faire
         courant ← courant.next
      Fin Tant que
      newNode.prec ← courant
      newNode.next ← courant.next
      Si courant.next ≠ NULL alors
         courant.next.prec ← newNode
      Fin Si
      courant.next ← newNode
   Fin Si
Fin
```

Fonction principale : main()

 Complexité temporelle globale :

- Construction de la liste avec n insertions : $O(n^2)$
- Affichage : $O(n)$
- Total = $O(n^2)$

 Complexité spatiale globale :

- Une structure par élément : $O(n)$

Début

Initialiser la liste à NULL

Afficher un message "Entrer des entiers (0 pour terminer)"

Lire val

Tant que val ≠ 0 faire

Appeler insertDsorted(&liste, val)

Lire val

Fin Tant que

Afficher "Liste triée :"

Appeler printflist(liste)

Fin

Programme en C :

/*===== liste doublement chainées :

5. Insertion en tête et en queue dans une liste doublement chaîne circulaire */

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct dnode{
   int data;
   struct dnode *next;
   struct dnode *prec;
}dnode;
```

```
//Fonction pour creer pour un nouveau noeud
dnode *createNode(int data){
   dnode *newNode = (dnode*)malloc(sizeof(dnode));
   if(!newNode){
      printf("erreur avec malloc\n");
      return NULL;
   }
   newNode-> data= data;
   newNode->next= NULL;
   newNode-> prec =NULL;
   return newNode;
}

void printflist(dnode *head){
   dnode *current = head;
   while(current != NULL){
      printf(" %d <-> ",current->data);
      current = current->next;
   }
   printf("NULL\n");
}

//Fonction pour inserer un element dans une liste doublement chaine trie
void insertDsorted(dnode **head,int data){
   dnode* newNode= createNode(data);

   //si la liste est vide ou si le nouveau noeud doit etre en tete
   if(*head == NULL || (*head)->data >=newNode->data){
      newNode->next = *head;
      if(*head != NULL){
         (*head)->prec = newNode;
      }
      *head = newNode;
   }
   else{
      dnode *current= *head;
      while(current->next && current->next->data<newNode->data){
         current= current->next;
      }
      newNode->prec = current;
      newNode->next = current->next;
      if(current->next != NULL){
         current->next->prec = newNode;
      }
      current->next = newNode;
   }
```

```
}

int main(){
      dnode *Mylistzone = NULL;
   printf("\n=== Liste doublement chaînée triée ===\n");
   printf("Entrer des entiers (0 pour terminer) : \n");
   int val;
   scanf("%d",&val);
   while(val !=0){
      insertDsorted(&Mylistzone,val);
      scanf("%d",&val);
   }
   printf("Liste triée: ");

  printflist(Mylistzone);


   return 0;
}
```

Exemple d'exécution :

```
=== Liste doublement chaînée triée ===
Entrer des entiers (0 pour terminer) :
7
2
9
4
0

Liste triée:  2 <-> 4 <-> 7 <-> 9 <-> NULL
```

Partie 4 : Insertion en tête et en queue dans une liste simplement chaîne circulaire

Objectif

L'objectif de ce programme est de créer et de manipuler une **liste simplement chaînée** dans laquelle chaque élément est inséré automatiquement à la bonne position pour que la liste reste **triée en ordre croissant**. Une **liste circulaire simplement chaînée** est une liste où le dernier élément pointe vers le premier, formant un cycle. Ici, chaque nœud contient :

Opérations principales

1. **Insertion en tête** : O(n)
2. **Insertion en fin** : O(n)
3. **Suppression en tête** : O(n)
4. **Suppression en fin** : O(n)
5. **Affichage** : O(n)

Avantages

- Structure circulaire adaptée aux parcours cycliques (ex. jeux, gestion circulaire de processus).
- Pas de "fin de liste", on peut tourner en boucle.

Limites

- Les opérations coûtent **O(n)** à cause de la recherche du dernier élément.
- Le champ prec est inutilisé (il pourrait être utile si tu voulais une **liste doublement chaînée circulaire**, qui rendrait certaines opérations plus rapides).

- Une valeur (val)
- Un pointeur vers le suivant (suiv)

Chaque nœud pointe à la fois vers :

- son **suivant**

Structures de données utilisées

- **Structure d'un nœud (dnode)**
  Chaque nœud contient :
- un entier data
- un pointeur next vers le nœud suivant

Forme :

```
struct dnode {
   int data;
   struct dnode *next;
};
```

Pseudo-code des fonctions

Structure de données :
Structure Noeud:
  entier val
  pointeur Noeud suiv
  pointeur Noeud prec (pas utilisé ici)

creation d'un nouveau nœud : fonction CreatLol(Valeur)

Entrée : valeur (entier)
Sortie : pointeur vers un nouveau noeud

Créer un nouveau noeud
newlol.val ← valeur
newlol.suiv ← newlol
newlol.prec ← newlol
Retourner newlol

Complexites :
Temps : O(1) ;
Espace : O(1).

Insertion en tête de liste : fonction Insertete(liste, valeur)

Entrée : adresse de la tête de la liste, valeur à insérer
Sortie : liste mise à jour

Créer un nouveau noeud avec createLol(valeur)
Si liste est vide
  liste ← newlol
Sinon
  temp ← liste
  Tant que temp.suiv ≠ liste
    temp ← temp.suiv
  Fin TantQue
  newlol.suiv ← liste
  temp.suiv ← newlol
  liste ← newlol
Fin Si

Complexité :
Temps : O(n) (A cause dyu parcours jusqu'au dernier élément) ;
Espace : O(1)

Insertion en fin de liste : **Fonction InsertFin(liste, valeur)**

Entrée : adresse de la tête, valeur
Sortie : liste mise à jour

Créer un nouveau noeud avec createLol(valeur)

```
Si liste est vide
   liste ← newlol
Sinon
   temp ← liste
   Tant que temp.suiv ≠ liste
      temp ← temp.suiv
   Fin TantQue
   temp.suiv ← newlol
   newlol.suiv ← liste
Fin Si
```

Comlplexité :
Temps : O(1) ;
Espace : O(1).

Suppression d'un élément en tête liste : **Fonction suppression_tete(liste)**

Entrée : adresse de la tête
Sortie : liste mise à jour

```
Si liste est vide
   Afficher "Liste déjà vide"
Sinon si un seul élément
   Libérer liste
   liste ← NULL
Sinon
   temp ← liste
   dernier ← liste
   Tant que dernier.suiv ≠ liste
      dernier ← dernier.suiv
   Fin TantQue
   liste ← liste.suiv
   dernier.suiv ← liste
   Libérer temp
Fin Si
```

Complexité :

Temps : O(n) (Trouve le dernier chemin)

Espace : O(1).

Suppression d'un élément en fin de liste : **Fonction deleteEnd(liste)**

Entrée : adresse de la tête
Sortie : liste mise à jour

```
Si liste est vide
   Afficher "Liste déjà vide"
Sinon si un seul élément
   Libérer liste
   liste ← NULL
Sinon
   temp ← liste
   prev ← NULL
   Tant que temp.suiv ≠ liste
      prev ← temp
      temp ← temp.suiv
   Fin TantQue
   prev.suiv ← liste
   Libérer temp
Fin Si
```

Complexité :
Temps : O(n) ;
Espace : O(1).

Affichage de la liste : **Fonction display(liste)**

Entrée : tête de la liste
Sortie : affichage de la liste

```
Si liste est vide
   Afficher "Liste vide"
Sinon
   temp ← liste
   Répéter
      Afficher temp.val
      temp ← temp.suiv
   Jusqu'à temp = liste
Fin Si
```

Complexité :
Temps : O(n) ;
Espace : O(1).

Code source en C :

```c
#include <stdio.h>
#include <stdlib.h>

struct lol {
    int val;
    struct lol *suiv;
    struct lol *prec;
};

struct lol *createLol(int valeur) {
    struct lol *newlol = (struct lol*)malloc(sizeof(struct lol));
    newlol->val = valeur;
    newlol->suiv = newlol; // Pointe vers elle-même pour une liste circulaire
    newlol->prec = newlol; // Pointe vers elle-même pour une liste circulaire
    return newlol;
}

void InsertTete(struct lol **tete, int valeur) {
    struct lol *newlol = createLol(valeur);

    if (*tete == NULL) {
        *tete = newlol;
    } else {
        struct lol *temp = *tete;
        // Parcourir jusqu'au dernier élément
        while (temp->suiv != *tete) {
            temp = temp->suiv;
        }
        newlol->suiv = *tete;
        temp->suiv = newlol;
        *tete = newlol;
    }
}

void InsertFin(struct lol **tete, int valeur) {
    struct lol *newlol = createLol(valeur);

    if (*tete == NULL) {
        *tete = newlol;
    } else {
        struct lol *temp = *tete;
        while (temp->suiv != *tete) {
            temp = temp->suiv;
        }
        temp->suiv = newlol;
        newlol->suiv = *tete;
    }
```

```
}

void display(struct lol *tete) {
    if (tete == NULL) {
        printf("Liste vide!\n");
        return;
    }

    struct lol *temp = tete;
    printf("Liste : ");
    do {
        printf("%d -> ", temp->val);
        temp = temp->suiv;
    } while (temp != tete);
    printf("(retour à tête)\n");
}

void suppression_tete(struct lol **tete) {
    if (*tete == NULL) {
        printf("Liste déjà vide!\n");
        return;
    }

    if ((*tete)->suiv == *tete) { // Un seul élément
        free(*tete);
        *tete = NULL;
    } else {
        struct lol *temp = *tete;
        struct lol *dernier = *tete;
        while (dernier->suiv != *tete) {
            dernier = dernier->suiv;
        }
        *tete = (*tete)->suiv;
        dernier->suiv = *tete;
        free(temp);
    }
    printf("Élément en tête supprimé.\n");
}

void deleteEnd(struct lol **tete) {
    if (*tete == NULL) {
        printf("Liste déjà vide!\n");
        return;
    }

    if ((*tete)->suiv == *tete) { // Un seul élément
        free(*tete);
```

```
        *tete = NULL;
    } else {
        struct lol *temp = *tete;
        struct lol *prev = NULL;
        while (temp->suiv != *tete) {
            prev = temp;
            temp = temp->suiv;
        }
        prev->suiv = *tete;
        free(temp);
    }
    printf("Élément en fin supprimé.\n");
}

int main() {
    struct lol *liste = NULL;
    printf("\n==================================\n");
    printf("Liste circulaire simplement chainée\n");
    printf("==================================\n");

    int a, b;
    do {
        printf("\n1. Insertion en tête de liste\n");
        printf("2. Insertion en fin de liste\n");
        printf("3. Suppression en tête de liste\n");
        printf("4. Suppression en fin de liste\n");
        printf("5. Affichage de la liste\n");
        printf("6. Quitter\n");
        printf("\nFaites un choix : ");
        scanf("%d", &b);

        switch(b) {
            case 1:
                printf("Insertion en tête\n");
                printf("Entrer une valeur : ");
                scanf("%d", &a);
                InsertTete(&liste, a);
                display(liste);
                break;
            case 2:
                printf("Insertion en fin\n");
                printf("Entrer une valeur : ");
                scanf("%d", &a);
                InsertFin(&liste, a);
                display(liste);
                break;
            case 3:
```

```
            printf("Suppression en tête\n");
            suppression_tete(&liste);
            display(liste);
            break;
        case 4:
            printf("Suppression en fin\n");
            deleteEnd(&liste);
            display(liste);
            break;
        case 5:
            printf("Affichage de la liste\n");
            display(liste);
            break;
        case 6:
            printf("Au revoir!\n");
            break;
        default:
            printf("Choix invalide. Veuillez réessayer.\n");
      }
   } while(b != 6);

   // Libération de la mémoire
   if (liste != NULL) {
      struct lol *courant = liste;
      struct lol *suivant;
      do {
         suivant = courant->suiv;
         free(courant);
         courant = suivant;
      } while (courant != liste);
   }

   return 0;
}
```

EXEMPLE DE SORTIE :

```
==================================
Liste circulaire simplement chainée
==================================

1. Insertion en tête de liste
2. Insertion en fin de liste
3. Suppression en tête de liste
4. Suppression en fin de liste
5. Affichage de la liste
6. Quitter
```

Faites un choix : 1
Insertion en tête
Entrer une valeur : 345
Liste : 345 -> (retour à tête)

1. Insertion en tête de liste
2. Insertion en fin de liste
3. Suppression en tête de liste
4. Suppression en fin de liste
5. Affichage de la liste
6. Quitter

Faites un choix : 2
Insertion en fin
Entrer une valeur : 678
Liste : 345 -> 678 -> (retour à tête)

Conclusion :

Ce code nous a permit de comprendre le fonction des données liées entre elles de façon circulaire à l'instard dans un système d'exploitation. C'est pourquoi en plus de la consigne on ajouté les fonctions de suppression d'occurrences.

Partie 5 : Insertion en tête et en queue dans une liste doublement chaîne circulaire

Objectif

L'objectif de ce programme est de créer et de manipuler une **liste doublement chaînée circulaire** dans laquelle le dernier élément est relié au premier élément et vice versa.Une **liste doublement circulaire** est une liste chaînée où chaque nœud a :

- suiv → vers le suivant
- prec → vers le précédent

Le dernier nœud pointe vers le premier, et le premier vers le dernier.

Avantages

- Accès direct au dernier nœud grâce à prec
- Insertion/suppression en tête ou en fin : O(1)
- Parcours bidirectionnel possible

Limites

- La complexité d'accès à un élément intermédiaire reste O(n)
- La structure est un peu plus lourde que la liste simplement circulaire (stockage de prec)

Pseudo-code des fonctions

Structure de données :

Structure Noeud:

entier val

pointeur Noeud suiv

pointeur Noeud prec

Creation d'un nœud : Fonction createLol(valeur)

Entrée : entier valeur
Sortie : pointeur vers un nouveau noeud

Créer un nouveau noeud newlol
Si allocation échoue
afficher "Erreur d'allocation" et quitter

```
Sinon
   newlol.val ← valeur
   newlol.suiv ← newlol
   newlol.prec ← newlol
Retourner newlol
```

Complexité :

Temps : O(1) ;

Espace : O(1).

Insertion en tête de liste : **Fonction InsertTete(liste, valeur)**

Entrée : adresse de la tête de la liste, entier valeur
Sortie : liste mise à jour

```
Créer un nouveau noeud newlol avec createLol(valeur)
Si liste vide
   liste ← newlol
Sinon
   last ← liste.prec  // dernier noeud
   newlol.suiv ← liste
   newlol.prec ← last
   last.suiv ← newlol
   liste.prec ← newlol
   liste ← newlol  // mise à jour de la tête
Fin Si
```

Complexité :

Temps :O(1) pas besoin de parcourir toute la liste grace à **prec**.

Affichage de la liste : **Fonction display(liste)**

Entrée : tête de la liste
Sortie : affichage des éléments

```
Si liste vide
   afficher "Liste vide"
Sinon
   temp ← liste
   Répéter
      afficher temp.val
      temp ← temp.suiv
```

```
    Jusqu'à temp = liste
Fin Si
```

Complexité :

Temps : O(n) ;

Espace : O(1)

Liberation de la mémoire :

```
Si liste non vide
    courant ← liste
    Répéter
        suivant ← courant.suiv
        libérer courant
        courant ← suivant
    Jusqu'à courant = liste
```

Complexité :

Temps : O(n) ;

Espace : O(1).

Programme principale :

```
Déclarer liste ← NULL
Afficher "Liste circulaire doublement chainée circulaire"

Répéter
    Afficher le menu :
        1. Insertion en tête
        2. Insertion en fin
        3. Affichage de la liste
        4. Quitter
    Lire choix b

    Selon b faire
        Cas 1 :
            Afficher "Insertion en tête"
            Lire valeur a
            Appeler InsertTete(&liste, a)
            Appeler display(liste)
        Cas 2 :
            Afficher "Insertion en fin"
```

```
            Lire valeur a
            Appeler InsertFin(&liste, a)
            Appeler display(liste)
        Cas 3 :
            Afficher "Affichage de la liste"
            Appeler display(liste)
        Cas 4 :
            Afficher "Au revoir !"
        Cas par défaut :
            Afficher "Choix invalide, veuillez réessayer"
    Fin Selon
Jusqu'à ce que b = 4

Si liste non vide alors
    courant ← liste
    Répéter
        suivant ← courant.suiv
        Libérer mémoire de courant
        courant ← suivant
    Jusqu'à ce que courant = liste
Fin Si
```

Complexité du main

- **Insertion en tête/fin** : O(1) (grâce au champ prec)
- **Affichage** : O(n)
- **Libération mémoire** : O(n)
- **Espace supplémentaire** : O(1)

Code source en C :

```c
#include <stdio.h>
#include <stdlib.h>

struct lol {
    int val;
    struct lol *suiv;
    struct lol *prec;
};

struct lol *createLol(int valeur) {
    struct lol *newlol = (struct lol*)malloc(sizeof(struct lol));
```

```
    if (newlol == NULL) {
        printf("Erreur d'allocation mémoire!\n");
        exit(1);
    }
    newlol->val = valeur;
    newlol->suiv = newlol;
    newlol->prec = newlol;
    return newlol;
}

void InsertTete(struct lol **tete, int valeur) {
    struct lol *newlol = createLol(valeur);

    if (*tete == NULL) {
        *tete = newlol;
    } else {
        struct lol *last = (*tete)->prec; // le dernier noeud

        // connecter le nouveau noeud
        newlol->suiv = *tete;
        newlol->prec = last;

        // mettre à jour les noeuds environnants
        last->suiv = newlol;
        (*tete)->prec = newlol;

        // mettre à jour la tête
        *tete = newlol;
    }
}

void InsertFin(struct lol **tete, int valeur) {
    struct lol *newlol = createLol(valeur);

    if (*tete == NULL) {
        *tete = newlol;
    } else {
        struct lol *last = (*tete)->prec; // le dernier noeud

        // connecter le nouveau noeud
        newlol->suiv = *tete;
        newlol->prec = last;

        // mettre à jour les noeuds environnants
        last->suiv = newlol;
        (*tete)->prec = newlol;
        // la tête reste inchangée
```

```
    }
}

void display(struct lol *tete) {
    if (tete == NULL) {
        printf("Liste vide!\n");
        return;
    }

    struct lol *temp = tete;
    printf("Liste : ");
    do {
        printf("%d -> ", temp->val);
        temp = temp->suiv;
    } while (temp != tete);
    printf("(retour à tête)\n");
}

int main() {
    struct lol *liste = NULL;
    printf("\n================================================\n");
    printf("Liste circulaire doublement chainée circulaire\n");
    printf("================================================\n");

    int a, b;
    do {
        printf("\n1. Insertion en tête de liste\n");
        printf("2. Insertion en fin de liste\n");
        printf("3. Affichage de la liste\n");
        printf("4. Quitter\n");
        printf("\nFaites un choix : ");
        scanf("%d", &b);

        switch(b) {
            case 1:
                printf("Insertion en tête\n");
                printf("Entrer une valeur : ");
                scanf("%d", &a);
                InsertTete(&liste, a);
                display(liste);
                break;
            case 2:
                printf("Insertion en fin\n");
                printf("Entrer une valeur : ");
                scanf("%d", &a);
                InsertFin(&liste, a);
                display(liste);
```

```
            break;
        case 3:
            printf("Affichage de la liste\n");
            display(liste);
            break;
        case 5:
            printf("Au revoir!\n");
            break;
        default:
            printf("Choix invalide. Veuillez réessayer.\n");
    }
} while(b != 4);

// Libération de la mémoire
if (liste != NULL) {
    struct lol *courant = liste;
    struct lol *suivant;
    do {
        suivant = courant->suiv;
        free(courant);
        courant = suivant;
    } while (courant != liste);
}

return 0;
}
```

Exemple d'exécution :

==========

Liste circulaire doublement chainée circulaire

==========

1. Insertion en tête de liste

2. Insertion en fin de liste

3. Affichage de la liste

4. Quitter

Faites un choix : 1

Insertion en tête

Entrer une valeur : 34

Liste : 34 -> (retour à tête)

1. Insertion en tête de liste
2. Insertion en fin de liste
3. Affichage de la liste
4. Quitter

Faites un choix : 245

Choix invalide. Veuillez réessayer.

1. Insertion en tête de liste
2. Insertion en fin de liste
3. Affichage de la liste
4. Quitter

Faites un choix : 2

Insertion en fin

Entrer une valeur : 56

Liste : 34 -> 56 -> (retour à tête)

Conclusion :

Ce code nous a permit de comprendre le fonction des données liées entre elles de façon circulaire à l'instard dans un système d'exploitation. C'est pourquoi en plus de la consigne on ajouté les fonctions de suppression d'occurrences.