ENSEIGNANT : PR. MELATAGIA

RAPPORT DE GROUPE

- Lire un élément et supprimer toutes ses occurrences dans une liste.
- Insertion d'un élément dans une liste simplement chaînée triée.
- Insertion d'un élément dans une liste doublement chaînée triée.
- Insertion en tête et en queue dans une liste simplement chaînée circulaire.
- Insertion en tête et en queue dans une liste doublement chaînée circulaire.

INF 231

PARTICIPANTS

WAFFO NZODJOU BLAISE CHISTIAN 24G2991
CHIMI YAPEWO FRESNEL 24G2227
KENFACK DZOMO RODRIGUE BROWN 24G2168
MBAMBA RAPHAEL BORIS 24G2803
AKONO OPONO CLARA IMELDA 24H2162
MECHE JENNIFER OCEANE 24F2975

Introduction

Les structures de données occupent une place essentielle en informatique car elles permettent une organisation efficace des informations. Parmi elles, les **listes chaînées** se distinguent par leur flexibilité et leur capacité à gérer dynamiquement la mémoire. Contrairement aux tableaux, elles permettent des insertions et suppressions sans nécessité de décaler les éléments.
Le présent devoir porte sur la mise en pratique des opérations de base sur différents types de listes chaînées en langage C : simples, doubles et circulaires.

1. Suppression de toutes les occurrences d'un élément dans une liste simplement chaînée

Explication

Une **liste simplement chaînée** est une structure où chaque nœud contient une valeur et un pointeur vers le suivant.
Ici, on demande de **supprimer toutes les occurrences d'un élément donné**.
Il faut donc parcourir la liste et vérifier :

- si l'élément est en **tête** → déplacer la tête,
- sinon → relier le précédent au suivant et libérer la mémoire.

Code

```c
// Suppression de toutes les occurrences d'un élément

void deleteOccurrences(Node **head, int key) {
    Node *temp = *head, *prev = NULL;

    while (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        temp = *head;
    }

    while (temp != NULL) {
        while (temp != NULL && temp->data != key) {
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL) return;

        prev->next = temp->next;
        free(temp);
        temp = prev->next;
    }
}

```

2. Insertion dans une liste simplement chaînée triée

Explication

Une **liste triée** est toujours ordonnée (croissante ici).
Quand on insère un élément, il faut :

- vérifier s'il doit aller en **tête**,
- sinon parcourir jusqu'à la bonne position pour l'insérer.

Code

```c
void insertSorted(Node **head, int data) {
    Node *newNode = createNode(data);
    Node *current;

    if (*head == NULL || (*head)->data >= data) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    current = *head;
    while (current->next != NULL && current->next->data < data) {
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
}
```

3. Insertion dans une liste doublement chaînée triée

Explication

Une **liste doublement chaînée** possède deux pointeurs par nœud :

- `prev` → vers le nœud précédent,
- `next` → vers le suivant.
  L'insertion doit garder l'ordre trié.
  Cas possibles :
- en **tête**,
- au **milieu**,

- en **queue**.

Code

```c
void insertSortedD(DNode **head, int data) {
    DNode *newNode = createDNode(data);
    DNode *current = *head;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    if (data <= (*head)->data) {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
        return;
    }

    while (current->next != NULL && current->next->data < data) {
        current = current->next;
    }

    newNode->next = current->next;
    if (current->next != NULL) {
        current->next->prev = newNode;
    }
    current->next = newNode;
    newNode->prev = current;
}
```

4. Insertion en tête et en queue dans une liste simplement chaînée circulaire

Explication

Dans une **liste circulaire simplement chaînée** :

- le dernier nœud pointe sur le premier,
- on peut parcourir la liste en boucle infinie.

On demande deux opérations :

- **Insertion en tête** : insérer un nouveau nœud avant l’ancien premier.
- **Insertion en queue** : insérer un nouveau nœud après le dernier.

Code

```c
void insertHeadC(CNode **head, int data) {
    CNode *newNode = createCNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    CNode *temp = *head;
    while (temp->next != *head) temp = temp->next;
    temp->next = newNode;
    newNode->next = *head;
    *head = newNode;
}

void insertTailC(CNode **head, int data) {
    CNode *newNode = createCNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    CNode *temp = *head;
    while (temp->next != *head) temp = temp->next;
    temp->next = newNode;
    newNode->next = *head;
}
```

5. Insertion en tête et en queue dans une liste doublement chaînée circulaire

Explication

Une **liste circulaire doublement chaînée** est plus complète :

- le `next` du dernier pointe sur le premier,
- le `prev` du premier pointe sur le dernier.

On peut donc ajouter en **tête** ou en **queue** facilement en mettant à jour les pointeurs `prev` et `next`.

Code

```c
void insertHeadDC(DCNode **head, int data) {
    DCNode *newNode = createDCNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    DCNode *tail = (*head)->prev;
    newNode->next = *head;
    newNode->prev = tail;
    tail->next = newNode;
    (*head)->prev = newNode;
    *head = newNode;
}

void insertTailDC(DCNode **head, int data) {
    DCNode *newNode = createDCNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    DCNode *tail = (*head)->prev;
    newNode->next = *head;
    newNode->prev = tail;
    tail->next = newNode;
    (*head)->prev = newNode;
}
```

Conclusion

Ce devoir a permis de :

- manipuler différents types de **listes chaînées** (simple, double, circulaire),
- apprendre à **insérer et supprimer correctement** des éléments,
- comprendre la gestion des **pointeurs** en langage C,
- voir la différence entre les structures linéaires et circulaires.

Ces opérations sont fondamentales en **structures de données** et servent de base pour des algorithmes plus avancés.