**TP INF 231. DOSSIER ALGORITHMIQUE 1**

**I. Liste doublement chaînée circulaire**

**1. Problème :**

Il s’agit ici pour nous de manipuler une « liste doublement chaînée circulaire» en appliquant

des opérations telles que :

• l’insertion en tête et fin.

**Contraintes :** Cependant nous faisons face à des difficultés tes que la création de la structure, la gestion dynamique de la mémoire et déterminer une complexité optimal pour les insertion.

**2 . Principe :**

Afin de résoudre se problème une panoplies de solution s’offre à nous et nous comptons procéder comme suit :

i. Utiliser une liste doublement chaînée circulaire vu que cette structure propose des Avantages tels que : un Accès bidirectionnel nous permettent ainsi de de nous déplacer dans les deux sens, le dernier nœud pointe vers le premier et vice-versa. Le plus impactant des avantages est que nous restons en complexité O(1) lors de l’insertion ou la suppression en tête et fin.

**3. Algorithme :**

**Algorithme** GestionListeDoublementChaineeCirculaire

Type

Cellule = Enregistrement

val : entier;

suiv : ^Cellule;

prec : ^Cellule;

FinEnreg;

Liste = ^Cellule;

**Var**

n, a, b, i : entier;

l, p, k, last, current, nod : Liste;

**Debut**

// Programme principal

ecrire("Entrez la longueur de la liste : ");

lire(n);

// **3.1. CRÉATION DE LA LISTE**

Si (n <= 0) Alors

nod <- NULL;

Sinon

l <- NULL;

last <- NULL;

Pour i de 0 à n-1 Faire

Allouer(p);

Si (p = NULL) Alors

ecrire("Erreur d'allocation");

Retourner -1;

FinSi;

ecrire("Entrez l'élément : ");

lire(p^.val);

Si (l = NULL) Alors

// Premier élément - création liste circulaire

p^.suiv <- p;

p^.prec <- p;

l <- p;

last <- p;

Sinon

// Ajout en fin de liste

p^.prec <- last;

p^.suiv <- l;

last^.suiv <- p;

l^.prec <- p;

last <- p;

FinSi;

FinPour;

nod <- l;

FinSi;

// **3.2. INSERTION EN TÊTE**

ecrire("Entrez la valeur à ajouter en tête : ");

lire(a);

Allouer(k);

Si (k = NULL) Alors

ecrire("Erreur d'allocation");

Retourner -1;

FinSi;

k^.val <- a;

Si (nod = NULL) Alors

// Liste vide - création singleton circulaire

k^.suiv <- k;

k^.prec <- k;

nod <- k;

Sinon

// Insertion avant la tête actuelle

k^.suiv <- nod;

k^.prec <- nod^.prec;

nod^.prec^.suiv <- k;

nod^.prec <- k;

nod <- k; // Nouvelle tête

FinSi;

// Affichage après insertion tête

ecrire("Éléments de la liste : ");

Si (nod = NULL) Alors

ecrire("Liste vide");

Sinon

current <- nod;

Répéter

ecrire(current^.val);

Si (current^.suiv <> nod) Alors

ecrire(" <-> ");

FinSi;

current <- current^.suiv;

Jusquà (current = nod);

ecrire("");

FinSi;

// **3.3. INSERTION EN FIN**

ecrire("Entrez la valeur à ajouter en fin : ");

lire(b);

Allouer(k);

Si (k = NULL) Alors

ecrire("Erreur d'allocation");

Retourner -1;

FinSi;

k^.val <- b;

Si (nod = NULL) Alors

// Liste vide - création singleton circulaire

k^.suiv <- k;

k^.prec <- k;

nod <- k;

Sinon

// Insertion après le dernier élément

last <- nod^.prec; // Dernier élément actuel

k^.suiv <- nod;

k^.prec <- last;

last^.suiv <- k;

nod^.prec <- k;

// La tête reste inchangée

FinSi;

// Affichage final

ecrire("Éléments de la liste : ");

Si (nod = NULL) Alors

ecrire("Liste vide");

Sinon

current <- nod;

Répéter

ecrire(current^.val);

Si (current^.suiv <> nod) Alors

ecrire(" <-> ");

FinSi;

current <- current^.suiv;

Jusquà (current = nod);

ecrire("");

FinSi;

// **3.4. LIBÉRATION MÉMOIRE**

Si (nod <> NULL) Alors

current <- nod^.suiv;

// Libérer tous les nœuds sauf la tête

TantQue (current <> nod) Faire

p <- current;

current <- current^.suiv;

Désallouer(p);

FinTantQue;

// Libérer la tête

Désallouer(nod);

nod <- NULL;

FinSi;

ecrire("Programme terminé avec succès");

Retourner 0;

**Fin.**

**4. Dictionnaire de données :**

**4.1. Type de données.**

| **Type** | **Description** |
| --- | --- |
| liste | Structure représentant un nœud de la liste |
| list | Pointeur vers un nœud de la liste |

**4.2. Structures**

**typedef struct cellule {**

**int val; // Donnée stockée dans le nœud**

**struct cellule\* suiv; // Pointeur vers le nœud suivant**

**struct cellule\* prec; // Pointeur vers le nœud précédent**

**} liste;**

**4.3. Variables principales**

| **Variable** | **Type** | **Description** |
| --- | --- | --- |
| l | list | Pointeur vers la tête de la liste |
| p, k | list | Pointeurs temporaires pour nouveaux nœuds |
| current | list | Pointeur pour le parcours |
| last | list | Pointeur vers le dernier nœud |
| n | int | Taille de la liste à créer |
| info | int | Valeur à insérer |

**5. Complexité :**

**5.1. Analyse asymptotique**

| **Opération** | **Complexité temporelle** | **Complexité spatiale** |
| --- | --- | --- |
| createnode(n) | O(n) | O(n) |
| inserthead(l, info) | **O(1)** | O(1) |
| insertend(l, info) | **O(1)** | O(1) |
| display(l) | O(n) | O(1) |

### **5.2 Justification des complexités**

**inserthead() et insertend() en O(1)** :

* Grâce à la structure circulaire, on accède directement au dernier élément via l->prec
* Pas besoin de parcourir toute la liste
* Nombre constant d'opérations (4 affectations de pointeurs)

**createnode() en O(n)** :

* Chaque insertion est en O(1)
* On répète n fois → O(n)

**display() en O(n)** :

* Parcours de tous les n éléments
* S'arrête quand on revient au point de départ