# TP INF 231 :INSERTION DANS UNE LISTE DOUBLEMENT CHAÎNÉE CIRCULAIRE TRIÉE

## 1. Problème

Nous voulons insérer un nouvel élément x dans une liste doublement chaînée circulaire déjà triée tout en gardant l’ordre croissant des éléments. L’algorithme doit déterminer automatiquement la position correcte où placer x.

## 2. Contraintes

- Gestion mémoire dynamique : Allouer un nouveau nœud avant l’insertion.  
- Maintenir l’ordre : Après insertion, la liste doit rester triée de façon croissante.  
- Structure circulaire : Le dernier nœud doit continuer à pointer vers le premier et vice-versa.  
- Robustesse : Gérer tous les cas possibles :  
 1. Liste vide  
 2. Insertion avant la tête (si x < premier élément)  
 3. Insertion entre deux éléments ou en fin (si x ≥ valeurs précédentes)

## 3. Principe

Nous allons parcourir la liste à partir de la tête :  
- Si la liste est vide, nous créons un singleton circulaire.  
- Si x est plus petit que la valeur de la tête, nous l’insérons avant la tête et mettons à jour la tête.  
- Sinon, nous parcourons la liste jusqu’à trouver le premier élément ≥ x, et insérons x avant cet élément.  
- Si nous revenons à la tête sans trouver d’élément ≥ x, cela signifie que x est le plus grand élément : nous l’insérons juste avant la tête (en fin de liste).

## 4. Algorithme

Algorithme InsertionListeTriee  
  
Type  
  
 Cellule = Enregistrement  
 val : entier;  
 suiv : ^Cellule;  
 prec : ^Cellule;  
 FinEnreg;  
  
 Liste = ^Cellule;  
  
Var  
 x : entier;  
 nod, newNode, current : Liste;  
  
Debut  
 ecrire("Entrez la valeur à insérer dans la liste triée : ");  
 lire(x);  
  
 Allouer(newNode);  
 Si (newNode = NULL) Alors  
 ecrire("Erreur d'allocation mémoire");  
 Retourner -1;  
 FinSi;  
  
 newNode^.val <- x;  
  
 // --- CAS 1 : LISTE VIDE ---  
 Si (nod = NULL) Alors  
 newNode^.suiv <- newNode;  
 newNode^.prec <- newNode;  
 nod <- newNode;  
 Sinon  
 // --- CAS 2 : INSERTION AVANT LA TÊTE ---  
 Si (x < nod^.val) Alors  
 newNode^.suiv <- nod;  
 newNode^.prec <- nod^.prec;  
 nod^.prec^.suiv <- newNode;  
 nod^.prec <- newNode;  
 nod <- newNode; // mise à jour de la tête  
 Sinon  
 // --- CAS 3 : INSERTION AU MILIEU OU EN FIN ---  
 current <- nod^.suiv;  
  
 TantQue (current <> nod ET current^.val < x) Faire  
 current <- current^.suiv;  
 FinTantQue;  
  
 // insertion avant current  
 newNode^.suiv <- current;  
 newNode^.prec <- current^.prec;  
 current^.prec^.suiv <- newNode;  
 current^.prec <- newNode;  
 // nod reste inchangée  
 FinSi;  
 FinSi;  
  
 // --- AFFICHAGE APRÈS INSERTION ---  
 ecrire("Éléments de la liste après insertion triée : ");  
 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;  
  
 Retourner 0;  
Fin.

## 5. Dictionnaire de données

Type :  
Cellule : Structure représentant un nœud de la liste  
Liste : Pointeur vers un nœud de la liste  
  
Variables principales :  
nod : Pointeur vers la tête de la liste  
newNode : Pointeur vers le nœud à insérer  
current : Pointeur pour parcourir la liste  
x : Valeur à insérer

## 6. Complexité

Opération : Insertion triée  
Complexité temporelle : O(n)  
Complexité spatiale : O(1)  
  
Justification : Dans le pire cas, on peut parcourir toute la liste avant de trouver la position d’insertion. Une seule allocation mémoire est utilisée, donc l’espace supplémentaire est constant.