DOSSIERS ALGORITHMIQUES SDU TP DE L'EC 2:STRUTURES DONNES

PROGRAMME 1 : LISTE SIMPLEMENT CHAÎNÉE AVEC INSERTION TRIÉE

1. PROBLÈME

Créer une liste simplement chaînée d'entiers et y insérer un nouvel élément de manière triée (ordre croissant).

2. PRINCIPE

Le principe est le suivant :

- Créer une liste chaînée en demandant à l'utilisateur de saisir n éléments
- Parcourir la liste pour trouver la position d'insertion d'un nouvel élément
- Insérer l'élément tout en maintenant l'ordre croissant
- L'insertion peut se faire en tête, au milieu ou en queue selon la valeur

3. DICTIONNAIRE DE DONNÉES

Variable	Type	Rôle
`l`	Liste (pointeur)	Pointeur vers la tête de la liste
`p`	Liste (pointeur)	Pointeur de parcours de la liste
`el`	int	Élément à insérer
`n`	int	Nombre d'éléments de la liste
`i`	int	Compteur de boucle
`val`	int	Valeur d'un nœud
`suiv`	Liste (pointeur)	Pointeur vers le nœud suivant
`t`	Liste (pointeur)	Pointeur temporaire pour l'insertion

4. ALGORITHME

Fonction creer_liste(Liste *l)

```
DÉBUT
   Lire (n) ; //nombre d'éléments
   p ← NULL ;

   POUR i DE 0 À n-1 FAIRE
      Lire (el) ;
      SI l ==NULL  ALORS
         Allouer (l) ;
         l→val ← el ;
         l→suiv ← NULL ;
         p ← l ;
      SINON
         allouer(p→suiv) ;
```

```
            p ← p→suiv ;
            p→val ← el ;
            p→suiv ← NULL ;
        FIN SI
    FIN POUR
FIN


Fonction insertion(Liste *l, int el)

DÉBUT
    Allouer(t) ;
    t→val ← el ;
    p ← l ;

    SI l=NULL OU el < l→val ALORS
        // Insertion en tête
        t→suiv ← l ;
        l ← t ;
    SINON
        // Recherche de la position d'insertion
        TANT QUE (p→suiv ≠ NULL FAIRE)
            SI p→suiv→val > el ALORS
                t→suiv ← p→suiv ;
                p→suiv ← t ;
                RETOURNER
            FIN SI
            p ← p→suiv ;'
        FIN TANT QUE

        // Insertion en queue
        p→suiv ← t ;
        t→suiv ← NULL ;
    FIN SI
FIN


Fonction afficher_liste(Liste l)

DÉBUT
    p ← l
    TANT QUE p ≠ NULL FAIRE
        Afficher p→val ;
        p ← p→suiv ;
    FIN TANT QUE
FIN


Fonction destruction_de_liste(Liste l)
DÉBUT
    p ← l
    TANT QUE p ≠ NULL FAIRE
```

l ← l→suiv
Libérer p
p ← l
FIN TANT QUE
FIN

5. COMPLEXITÉ

- Complexité temporelle :
*creer_liste : O(n) où n est le nombre d'éléments
*insertion : O(n) dans le pire cas (insertion en queue)
*afficher_liste : O(n)
*destruction_de_liste: O(n)
*Complexité globale : O(n)

- Complexité spatiale :
*O(n) pour stocker n éléments dans la liste
*O(1) pour les variables auxiliaires

PROGRAMME 2 : INSERTION ET SUPPRESSION DANS UNE LISTE SIMPLEMENT CHAÎNÉE CIRCULAIRE

1. PROBLÈME

Implémenter une liste chaînée circulaire avec des opérations d'insertion en tête et en queue.

2. PRINCIPE

Le principe est le suivant :
- Une liste circulaire où le dernier élément pointe vers le premier
- Pour l'insertion en tête : créer le nouveau nœud, le faire pointer vers la tête actuelle, puis faire pointer le dernier vers ce nouveau nœud qui devient la nouvelle tête
- Pour l'insertion en queue : parcourir jusqu'au dernier, créer le nouveau nœud, le faire pointer vers la tête, et mettre à jour le pointeur du dernier

3. DICTIONNAIRE DE DONNÉES

Variable	Type	Rôle
`head`	cellule*	Pointeur vers la tête de la liste
`nouveau`	cellule*	Nouveau nœud à insérer
`temp`	cellule*	Pointeur de parcours
`valeur`	int	Valeur à insérer
`n`	int	Nombre de valeurs initiales
`choix`	int	Choix du menu

4. ALGORITHME

Fonction insererTete(cellule **head, int valeur)

```
DÉBUT
   Allouer (nouveau) ;
   nouveau→val ← valeur ;

   SI head=NULL  ALORS
      nouveau→suiv ← nouveau ;
      head ← nouveau ;
   SINON
      temp ← head
      // Trouver le dernier élément
      TANT QUE (temp→suiv ≠ head) FAIRE
         temp ← temp→suiv ;
      FIN TANT QUE

      nouveau→suiv ← head ;
      temp→suiv ← nouveau ;
      head ← nouveau ;
   FIN SI
FIN


Fonction insererQueue(cellule **head, int valeur)

DÉBUT
   Allouer (nouveau) ;
   nouveau→val ← valeur ;

   SI head=NULL ALORS
      nouveau→suiv ← nouveau ;
      head ← nouveau ;
   SINON
      temp ← head ;
      // Trouver le dernier élément
      TANT QUE (temp→suiv ≠ head) FAIRE
         temp ← temp→suiv ;
      FIN TANT QUE

      temp→suiv ← nouveau ;
      nouveau→suiv ← head ;
   FIN SI
FIN
```

5. COMPLEXITÉ

- Complexité temporelle :
    * insererTete : O(n) - doit parcourir pour trouver le dernier
    * insererQueue : O(n) - doit parcourir pour trouver le dernier
    * afficher : O(n)

- Complexité spatiale :
    * O(n) pour la liste

* O(1) pour les variables auxiliaires

Note d'amélioration: Avec un pointeur vers la queue, les insertions seraient en O(1).

PROGRAMME 3 : INSERTION EN TETE ET QUEUE DANS UNE LISTE DOUBLEMENT CHAÎNÉE CIRCULAIRE

1. PROBLÈME

Implémenter une liste doublement chaînée circulaire avec insertion en tête et en queue.

2. PRINCIPE

Le principe est le suivant :

- Chaque nœud a deux pointeurs : vers le suivant et vers le précédent
- Structure circulaire : le premier pointe vers le dernier (prec) et le dernier vers le premier (suiv)
- Insertion en tête : modifier les pointeurs du nouveau nœud, de l'ancienne tête et du dernier
- Insertion en queue : créer en tête puis déplacer le pointeur de tête

3. DICTIONNAIRE DE DONNÉES

Variable	Type	Rôle
`l`	liste*	Pointeur vers la tête de la liste
`p`	liste*	Nouveau nœud à insérer
`q`	liste*	Pointeur de référence (tête)
`val`	int	Valeur du nœud
`prec`	liste*	Pointeur vers le nœud précédent
`suiv`	liste*	Pointeur vers le nœud suivant

4. ALGORITHME

Fonction insererQueue(liste **l, int val)

```
DÉBUT
   Allouer (p) ;
   p→val ← val ;

   SI l=NULL ALORS
      p→suiv ← p ;
      p→prec ← p ;
      l ← p ;
   SINON
      q ← l ;
      p→suiv ← q ;
      p→prec ← q→prec. ;
      q→prec→suiv ← p ;
      q→prec ← p ;
   FIN SI
FIN
```

Fonction insererTete(liste **l, int val)

```
DÉBUT
   insererQueue(l, val) ;
   l ← l→prec ; // Déplacer la tête vers le nouveau dernier élément
FIN


 Fonction afficher(liste *l)

DÉBUT
   SI l=NULL ALORS
      RETOURNER
   FIN SI

   p ← l ;
   RÉPÉTER
      Afficher p→val ;
      p ← p→suiv ;
   JUSQU'À p = l
FIN
```

5. COMPLEXITÉ

- Complexité temporelle :
    * insererQueue : O(1) - accès direct au dernier via l→prec
    * insererTete : O(1) - réutilise insererQueue puis déplace pointeur
    * afficher : O(n)

- Complexité spatiale :
    * O(n) pour la liste
    * O(1) pour les variables auxiliaires

Avantage : Les insertions sont en temps constant grâce à la double liaison !

PROGRAMME 4 : SUPPRESSSION D'UN ELEMENT (AINSI QUE TOUTES SES OCCURENCES) DANS UNE LISTE DOUBLEMENT CHAÎNÉE (NON CIRCULAIRE)

1. PROBLÈME

Créer une liste doublement chaînée et supprimer toutes les occurrences d'un élément donné.

2. PRINCIPE

Le principe est le suivant :

- Créer une liste avec des pointeurs vers le suivant et le précédent
- Parcourir la liste pour rechercher l'élément à supprimer
- Gérer les cas particuliers : suppression en tête, au milieu, en queue
- Réajuster les pointeurs après chaque suppression

3. DICTIONNAIRE DE DONNÉES

Variable	Type	Rôle
`l`	Liste	Pointeur vers la tête de la liste
`p`	Liste	Pointeur de parcours
`t`	Liste	Pointeur temporaire pour suppression
`el`	int	Élément à supprimer
`val`	int	Valeur d'un nœud
`suiv`	Liste	Pointeur vers le suivant
`pre`	Liste	Pointeur vers le précédent

4. ALGORITHME

```
Fonction creer_liste(Liste *l)
DÉBUT
   Lire (n) ;
   POUR i DE 0 À n-1 FAIRE
      Lire (el) ;
      SI l est vide ALORS
         Allouer (p) ;
         p→val ← el ;
         l ← p ;
         l→suiv ← NULL ;
         l→pre ← NULL ;
      SINON
         Allouer (p→suiv) ;
         p→suiv→pre ← p ;
         p ← p→suiv ;
         p→val ← el ;
      FIN SI
   FIN POUR
   p→suiv ← NULL ;
FIN
```

Algorithme de suppression (dans main)

```
DÉBUT
   Lire( el ) ;
   p ← l ;

   TANT QUE p→suiv ≠ NULL FAIRE
      SI l→val = el ALORS
         // Suppression en tête
         l ← l→suiv ;
         DESALLOUER(P) ;
         p ← l ;
      FIN SI
```

```
        SI p→suiv→val = el ALORS
            // Suppression au milieu
            t ← p→suiv ;
            p→suiv ← p→suiv→suiv ;
            DESALLOUER(P) ;
        SINON
            p ← p→suiv ;
        FIN SI
    FIN TANT QUE
FIN
```

5. COMPLEXITÉ

Complexité temporelle :

* creer_liste : O(n)
* Suppression : O(n) - un seul parcours
* afficher_liste : O(n)
* destruction_de_liste : O(n)

- Complexité spatiale :

* O(n) pour la liste
* O(1) pour les variables auxiliaires

COMPARAISON DES STRUCTURES

Structure	Insertion tête	Insertion queue	Parcours	Espace mémoire
Simple	O(1)	O(n)	O(n)	n pointeurs
Simple circulaire	O(n)*	O(n)*	O(n)	n pointeurs
Double	O(1)	O(1)	O(n)	2n pointeurs
Double circulaire	O(1)	O(1)	O(n)	2n pointeurs