REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
ECOLE NATIONALE SUPERIEURE D'INFORMATIQUE

CPI 1 **Année 2021 / 2022**

TP
EN ALGORITHMIQUE ET STRUCTURES DE DONNEES DYNAMIQUES (ALSDD)

REALISE PAR :
SERIR Mouloud
MAMOUNI Mahdi
Section : D

Groupe N° : 11

Semestre 2

Sommaire :

I. Introduction :

Notre travail consiste principalement à garder des identifiants uniques représentés par une suite alphanumérique de taille 16 caractères dans une structure dynamique à base des liste linaires chainées.

on utilisants 4 representations differentes .

A priori , on songerait probablement à utiliser une structure de tableau dans lequel chaque case contiendrait un identifiants , toutefois , dans le cas où nous sommes face à un texte d'une taille considérable , cette tâche s'avère être très fastidieuse et pas très pratique et optimale en terme de gestion de mémoire , c'est ici qu'interviennent les listes linéaires chaines offrant la possibilité de stocker un nombre non-défini au préalable de données s'allongeant au fur et à mesure du traitement du texte ,

Les listes chaînées représentent une façon d'organiser les données en mémoire de manière beaucoup plus flexible. Comme à la base le langage C ne propose pas ce système de stockage, nous allons devoir le créer nous-mêmes de toutes pièces.

II-Les structures de données proposées :

-Represenation 4:

-Une LLC contiguë est une liste linéaire chainée représenté dans un même tableau où chaque élément renferme généralement 03 champs : la valeur, l'indice du prochain maillon de la liste et un booléen qui permet de savoir si l'espace est libre ou pas.

Afin de travailler sur les LLCs contiguës, on abordera les structures suivantes :

Type ELt_Tab_LLC = enregistrement

val : typelem_li

suiv : entier

vide : booléen

End ;

Type Tab_LLC = Tableau [1...Taille_Max] de ELt_Tab_LLC.

Type LLC_contigue = enregistrement

Tab : Tab_LLC

Tête : entier

Taille : entier

End ;

-Typelem_li: est un pointeur vers caracteres

val : typelem_li suiv : entier vide : booléen					

-Represenation 2:

La structure est une liste linéaire chainees circulaire unidirectionnelle ,chaque maillon contient un champs val (pointeur vers 16 caractères) , et un champs suivants (adresse du prochain maillon) , tel que le champs adresse du dernier maillon contient l'adresse du premier maillion

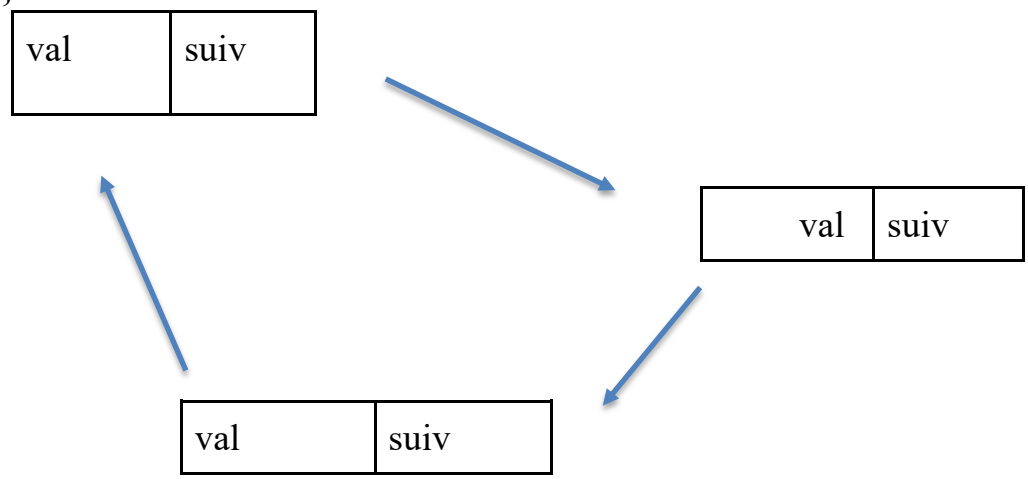
struct Maillon_Li

{

Typeelem_Li Val ;

Pointeur_Li Suiv ;

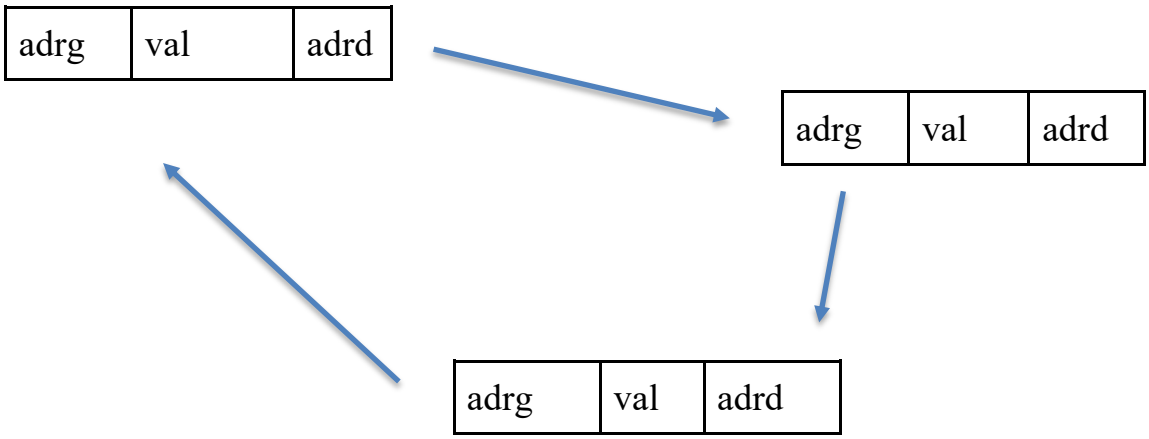
}



-Represenation 3:

La structure est une liste linéaire chainees circulaire bidirectionnelle ,chaque maillon contient un champs val (pointeur vers 16 caractères) , et un champs adrd (adressesdu prochain maillon) , et un champs adrg (adresse du maillon précédent) ,tel que le champs adrd du dernier maillon contient l'adresse du premier maillon

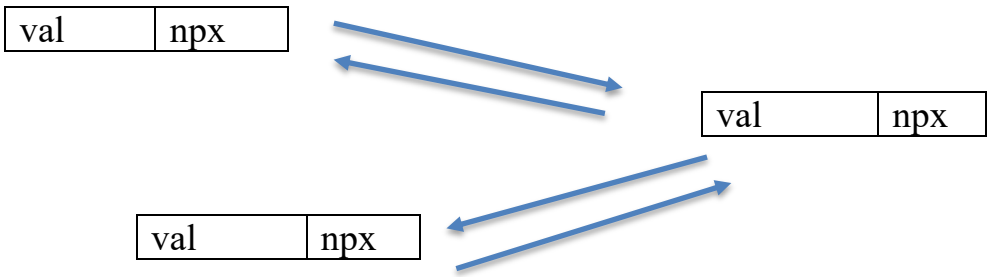
```
struct Maillon_Lb

{

Typeelem_Li Val ;

Pointeur_Lb Adrg ;          // adresse vers le maillon à gauche

Pointeur_Lb Adrd ; // adresse vers le maillon à droite

} ;
```



-Represenation 1:

La structure est une liste linéaire chainees bidirectionnelle permettant d'acceder a l element suivant ou au precedent a l aide d'un seul champ de pointeur, on l'appelle la liste lineaire chainee XOR. En effet, dans chaque champ de pointeur d'un maillon de cette liste on garde le XOR de l'adresse de l'element precedent avec l'adresse de l'element suivant. Si on veut obtenir l'adresse de l'element suivant on effectue l'operation du XOR entre l'adresse stockee dans le maillon avec celle de l'element precedent et si on veut obtenir l'adresse de l'element precdent on effectue l'operation du XOR entre l'adresse stockee dans le maillon avec celle de l'element suivant.

```
typedef char* type_elem_XOR ;

typedef struct Maillon_XOR *Pointeur_XOR ;

struct Maillon_XOR {

        type_elem_XOR   Val ;

        Pointeur_XOR   npx ;

} ;
```



III-Fonctions et procédures utilisés pour chaque representation:

-representation 2:

-void Allouer_Li (Pointeur_Li *P) // allouer un espace et retourner son adresse

-void Liberer_Li (Pointeur_Li P) //libérer l'espace déjà alloué

-Pointeur_Li Suivant_Li(Pointeur_Li P) // retourner l'adresse du prochain ÈlÈment contenu dans le maillon P

-Typeelem_Li Valeur_Li(Pointeur_Li P) // retourner l'information contenue dans le maillon P

-void Aff_val_Li(Pointeur_Li P, Typeelem_Li Val) // affecter l'information au champ vam du maillon

-void Aff_adr_Li(Pointeur_Li P, Pointeur_Li Q)// lier P a une autre adresse

-void create_llc_2(Pointeur_Li *P, int length) // créer la liste
-int rech_val2 (Pointeur_Li P, char val[]) // recherche par valeur dans la llc
-void insersion_val2 (Pointeur_Li *P,char val[])// pour insérer un identifiant
-void Longuer_LLC2(Pointeur_Li P)// retourne la longueur de la liste
-void afficher_LLC2(Pointeur_Li P)// affiche la liste
-void supp_LLC2(Pointeur_Li *P, char val[])// pour supprimer un identifiant

-representation 3:

-void Allouer_Lb (Pointeur_Lb *P) // allouer un espace et retourner son adresse

-void Liberer_Lb (Pointeur_Lb P) // libérer l'espace déja alloué

-Pointeur_Lb Suivant_Lb(Pointeur_Lb P) // retourner l'adresse du prochain élément contenu dans le maillon P

-Pointeur_Lb Precedent_Lb(Pointeur_Lb P) // retourner l'adresse du l'élément qui precède le maillon P

-Typeelem_Li Valeur_Lb(Pointeur_Lb P) // retourner l'information contenue dans le maillon P

-void Aff_val_Lb(Pointeur_Lb P, Typeelem_Li Val) // affecter l'information au champ vam du maillon P

-void Aff_adrd(Pointeur_Lb P, Pointeur_Lb Q)// lier P à une autre adresse vers la droite

-void Aff_adrg(Pointeur_Lb P, Pointeur_Lb Q)// lier P à une autre adresse vers la gauche

-void create_llc3(Pointeur_Lb *P, char* mat[])// pour créer la liste
-void affich_llc3(Pointeur_Lb P)// pour afficher la liste
-int rech_val3 (Pointeur_Lb P, char val[])// recherche par valeur dans la liste
-void insersion_val3 (Pointeur_Lb *P,char val[16])// insérer les identifiants

-representation 4:

-void init_vide_T(LLC_contigue *Lc);// pour initialiser la liste a vide

-Typeelem_Li valeur_T(LLC_contigue Lc, int p);// retourne le champ valeur

-int suivant_T(LLC_contigue Lc, int p);// retourne le champ suivant

-void aff_adr_T(LLC_contigue *Lc, int p, int adr);// affecte une adresse au champ suiv

-void aff_val_T(LLC_contigue *Lc, int p, Typeelem_Li val);//affecte une valeur au champ val

-void insertval_contig(LLC_contigue *L,Typeelem_Li v)// pour inserer les identifiants par ordre
-int rechval_contigue(LLC_contigue L,Typeelem_Li v)// rechercher un identifiant
-void suppval_contigue(LLC_contigue *L,Typeelem_Li v)//suppression par valeur

-representation 1:

-void Allouer_XOR(Pointeur_XOR* P)// allouer un espace et retourner son adresse

-type_elem_XOR Valeur_XOR(Pointeur_XOR P)//retourne le contenu du champ Val

-Pointeur_XOR XOR(Pointeur_XOR P,Pointeur_XOR Q) // Fonction qui effectue l'operation du XOR entre 2 adresses

-void Liberer_XOR (Pointeur_XOR P)//Supprime le maillon a l'adresse P

-Pointeur_XOR Suivant_XOR (Pointeur_XOR Precedent,Pointeur_XOR P)//Retourne l'adresse reelle de l'element qui suit de P

-Pointeur_XOR Precedent_XOR(Pointeur_XOR Suivant, Pointeur_XOR P) // Retourne l'adresse reelle de l'element qui precede de P

-void aff_val_XOR (Pointeur_XOR P, type_elem_XOR Val) //Affecte Val dans le champ Val d'un maillon

-void aff_adr_XOR (Pointeur_XOR P, Pointeur_XOR Q)//Affecte la valeur de Q dans le champ pointeur d'un maillon

-void Creer_XOR(Pointeur_XOR* P, char* mat[])//Permet de creer une liste XOR a partir d'un maillon cree deja precedemment

-void Afficher_GD(Pointeur_XOR P)//Affiche de gauche a droite la liste XOR

-void inserer_XOR(Pointeur_XOR *P,char ID[])//Insere par valeur un ID dans la liste XOR

Conclusion :

Après l'implémentation des quatre représentations et la comparaison entre eux on deduit qu'aucune de ces quatre n'est complète est considérée comme parfaite, chaque une a ces avantages et ces inconvénients. Mais on peut dire que la 4 ème représentation est la meilleure on ce qui concerne la recherche , les représentations 1 et 3 sont les meilleures en insertion , et la deuxième en suppression .