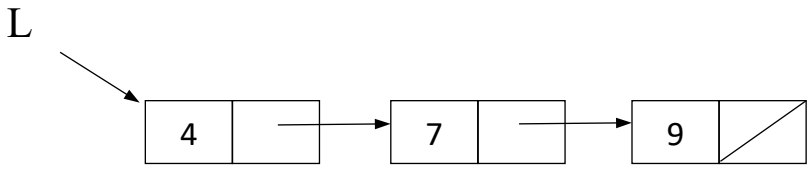
Chapitre 2 – LES LISTES CHAÎNÉES

1. Présentation

Une liste chaînée est une structure de données qui permet de relier plusieurs éléments de même type. Elle est qualifiée de structure de données dynamique car, contrairement à un tableau, ses éléments sont ajoutés au fur et à mesure par allocation dynamique durant le déroulement d'un algorithme. Il est donc nécessaire de garder au niveau de chaque élément un lien vers l'élément suivant dans la liste. Pour cela on peut utiliser des pointeurs.

2. Représentation graphique d'une liste chaînée

La figure suivante montre un exemple de liste chaînée d'entiers :



Dans cet exemple, L pointe sur le premier maillon de la liste. Donc :

*L désigne ce maillon

(*L).valeur, ou L->valeur, désigne le champ valeur du maillon et vaut 4.

(*L).suivant, ou L->suivant, désigne le champ suivant du maillon, ce qui correspond au pointeur sur le deuxième maillon.

Le dernier maillon n'a pas de suivant, son champ L->suivant vaut NULL.

3. Notion d'allocation dynamique de mémoire

L'allocation dynamique de mémoire est la réservation d'un espace en mémoire pour un ou plusieurs éléments durant l'exécution d'un programme en utilisant une instruction. L'adresse de l'espace alloué peut être gardée dans un pointeur pour pouvoir y accéder par la suite.

3-1. Allocation dynamique de mémoire en pseudo-code

En pseudo-code, l'allocation dynamique de mémoire peut se faire avec la syntaxe suivante :

id_pointeur = allouer(nb, type)

où

nb est le nombre d'éléments à stocker dans l'espace mémoire

type est le type des éléments à stocker dans cet espace.

id_pointeur est un pointeur sur le type de données spécifié

La libération de l'espace mémoire allouée de manière dynamique peut se faire avec la syntaxe suivante :

libérer(id_pointeur)

Exemple

```
Variable
    p : pointeur sur entier
    p = allouer( 1 , entier )        // allocation d'espace pour un entier
 *p = 10                             // l'espace alloué reçoit la valeur 10
 libérer( p )
```

3-2. Alocation dynamique de mémoire en langage C ou C++

En langage C ou C++, l'allocation dynamique de mémoire peut se faire avec les fonctions **malloc** et **calloc** de la librairie standard **stdlib.h.**

La fonction **calloc** prend en paramètres un nombre d'éléments nb et la taille correspondant au type des éléments et permet d'allouer un espace pour nb éléments.

La fonction **malloc** prend en paramètres la taille totale en octets de l'espace mémoire à allouer.

En cas de succès, ces deux fonctions retournent l'adresse de l'espace alloué. En cas d'échec, elles retournent la valeur NULL.

L'adresse retournée par ces deux fonctions est de type void*. Il faut la convertir au type du pointeur auquel on va l'affecter.

La fonction **free** de la librairie stdlib.h permet de faire la désallocation (libération) d'un espace mémoire alloué de manière dynamique selon la syntaxe suivante:

```
free( id_pointeur ) ;
```

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
        int * p1 ;
        int * p2 ;
        int n1 = 0 , n2 = 0 ;
        p1 = ( int * )malloc( sizeof( int ) ) ;
        *p1 = 10 ;
        p2 = ( int * )calloc( 1, sizeof( int ) ) ;
        *p2 = 30 ;
        printf( "Avant : n1 contient %d et n2 contient %d" , n1 , n2 ) ;
        n1 = *p1 ;
        n2 = *p2 ;
        printf( "\nAprès : n1 contient %d et n2 contient %d" , n1 , n2 ) ;
        free( p1 ) ;
        free( p2 ) ;
        return 0 ;
}
```

Remarque

En langage C++, on peut aussi utiliser l'opérateur new pour l'allocation dynamique d'un espace mémoire pour un ou plusieurs éléments de même type.

Pour un seul élément, on utilise la syntaxe suivante :

```
type * id_pointeur = new  type ;
```

Pour un nombre nb d'éléments, on utilise la syntaxe suivante :

```
type * id_pointeur = new type[ nb ] ;
```

Exemple

```
int *p1 , *p2 ;
p1 = new int ;      // allocation d'espace pour un entier
*p1 =10 ;
p2 = new int[ 5 ] ;       // allocation d'espace pour 5 entiers
```

L'opérateur delete permet de libérer un espace mémoire avec l'opérateur new.

Si l'espace a été alloué pour un seul élément, il faut utiliser la syntaxe suivante :

```
delete id_pointeur ;
```

Si l'espace a été alloué pour plusieurs éléments, il faut utiliser la syntaxe suivante :

```
delete [ ] id_pointeur ;
```

Par exemple :

```
delete p1 ;
delete [ ] p2 ;
```

4. Implémentation d'une liste chaînée

Un élément (ou maillon) d'une liste chaînée peut être représentée par un enregistrement qui contient un champ pour la valeur de l'élément et un champ qui est un pointeur sur l'élément suivant. Pour le dernier élément, le champ pointeur sur l'élément suivant vaut NULL.

Pour une liste chaînée d'entiers, le type des éléments peut être défini comme suit :

```
Type
    maillon = Enregistrement
        valeur : entier
        suivant : pointeur sur maillon
    FinEnregistrement
```

Un pointeur sur le premier élément permet de représenter la liste chaînée. On peut donc déclarer une liste chaînée L comme un pointeur sur le type maillon :

```
Variable
    L :  pointeur sur maillon
```

Mais pour des raisons de commodité, on peut définir un type appelé Liste équivalent au type pointeur sur maillon de la manière suivante :

```
Type
    Liste = pointeur sur maillon
```

Ainsi on pourra déclarer la liste chaînée L de la manière suivante :

```
Variable
    L : Liste
```

Cette variable contient l'adresse du premier maillon et représente la liste chaînée. Elle vaut NULL pour une liste vide.

En langage C, on peut traduire de manière suivante :

```
// Définition du type des éléments de la liste chaînée
struct maillon
{
    int valeur ;
    struct maillon* suivant ;
} ;
typedef struct maillon* Liste ;  //  Liste est équivalent à maillon*
```

```
int main( )
{
    // Déclaration de la liste chaînée L et initialisation avec la valeur
    NULL
    Liste L ;    // ou bien : struct maillon* L;
    L = NULL ;
    …
    return 0 ;
}
```

5. Ajout d'un élément au début d'une liste chaînée

Pour ajouter un élément au début d'une liste chaînée, il faut d'abord allouer de l'espace pour un nouveau maillon qui va contenir sa valeur. Ensuite, le champ suivant de ce nouveau maillon va pointer sur le premier maillon de la liste. Après, le pointeur de début de liste pointe sur le nouveau maillon. D'où la fonction suivante :

```
AjouterDebut( E-S L : Liste, E v : entier ) : entier
Début
    Variable
        Q : Liste
    Q = allouer( 1 , maillon )
    Si (Q==NULL)
        Retourner 0       // en cas d'échec la fonction se termine
    FinSi
    Q->valeur = v
    Q->suivant = L
    L = Q
    Retourner 1
FinFonction
```

Remarque

Cette fonction peut être appelée plusieurs fois (dans une boucle par exemple) pour remplir une liste chaînée initialement vide.

On peut traduire cette fonction en langage C comme suit :

```
int AjouterDebut(Liste * pL , int v)     // passage par adresse
{
    Liste Q = ( Liste )malloc( sizeof(struct maillon) ) ;
    if ( Q == NULL )
        return 0 ;
    else
    {
        Q->valeur = v ;
        Q->suivant = *pL ;
        *pL = Q ;
        return 1 ;
    }
}
```

En C++ , on peut utiliser un passage de paramètre par référence :

```
int AjouterDebut(Liste &L, int v)        // passage par référence
{
    Liste Q = new maillon ;
    if ( Q == NULL )
        return 0 ;
    else
    {
        Q->valeur = v ;
        Q->suivant = L ;
        L = Q ;
        return 1 ;
    }
}
```

6. Parcours d'une liste chaînée

On peut parcourir les éléments d'une liste chaînée avec un pointeur en utilisant une boucle pour effectuer un traitement. Par exemple, la fonction ParcourirListe définie ci-après peut être appelée pour l'affichage d'une liste chaînée :

```
ParcourirListe( L : Liste )
Début
    Variable
        Q : Liste
    Q = L
    Tant que ( Q ≠ NULL )
        Ecrire( Q->valeur)
        Q = Q->suivant
    FinTantQue
FinFonction
```

7. Ajout d'un élément à la fin d'une liste chaînée

Pour ajouter un élément à la fin d'une liste chaînée, il faut allouer de l'espace pour un nouveau maillon qui va contenir sa valeur. Si la liste est vide, le pointeur de début de liste va pointer sur le nouveau maillon sinon il faut parcourir la liste jusqu'au dernier maillon pour ensuite faire pointer son champ suivant sur le nouveau maillon. D'où la fonction suivante :

```
AjouterFin( E-S L : Liste, E v : entier ) : entier
Début
    Variable
        P, Q : Liste
    Q = allouer( 1 , maillon )
    Si (Q==NULL)
        Retourner 0    // Si l'allocation échoue la fonction se termine
    FinSi
    Q->valeur = v
    Q->suivant = NULL
    Si ( L == NULL )
        L = Q
    Sinon
        P = L
        Tant Que ( P->suivant ≠ NULL )
            P = P->suivant
        FinTantQue
        P->suivant  = Q
    FinSi
    Retourner 1
FinFonction
```

8. Suppression d'un élément d'une liste chaînée

La suppression consiste à libérer l'espace occupé par le maillon contenant une certaine valeur dans une liste non vide.

Si la valeur se trouve dans le premier maillon, il suffit de supprimer ce maillon après avoir déplacé le pointeur de début sur le maillon suivant.

Si la valeur n'est pas dans le premier maillon, il faudra parcourir la liste pour placer un pointeur sur le maillon qui précède le maillon qui contient la valeur.

Ainsi, on pourra supprimer ce dernier après avoir utilisé le pointeur pour relier son précédent et son suivant. Si la valeur n'est pas dans la liste, le pointeur sera sur le dernier maillon à la fin du parcours.

D'où la fonction suivante :

```
SupprimerElement( E-S L : Liste, E v : entier )
Début
    Variable
        P, Q : Liste
    Si ( L ≠ NULL )
        Si ( L->valeur == v )
            Q = L
            L = L->suivant
            liberer( Q )
        Sinon
            P = L
            Tant Que ( ( P->suivant ≠ NULL ) ET ( P->suivant->valeur ≠ v ) )
                P = P->suivant
            FinTantQue
            Si ( P->suivant ≠ NULL )
                Q = P->suivant
                P->suivant = Q->suivant
                libérer( Q )
            FinSi
        FinSi
    FinSi
FinFonction
```

Remarque

Certaines des fonctions que nous venons peuvent être définies de manière récursive.