Chapitre 2 : Les listes linéaires

1. Introduction

Une liste linéaire est la forme la plus courante d'organisation des données. **On l'utilise pour stocker des données qui doivent être traitées de manière séquentielle.** La structure doit également être évolutive, c'est-à-dire que l'on doit pouvoir ajouter et supprimer des éléments.

2. Définition d'une liste

Une liste est une suite finie (éventuellement vide) d'éléments de même type repérés selon leur rang dans la liste.

Exemple :

- La liste d'entiers L = {1, 5, 9,0, 10, 25}, Le rang de 5 est 2.
- La liste dés étudiants de la 2 ^{eme} année informatique.

3. Types

Selon la manière de réservation de la mémoire centre, on peut distingue deux types de listes linéaires:

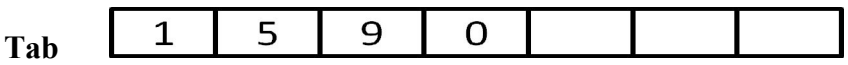
- **Listes statiques :** Réservation statique de la mémoire et implémentation par des tableaux de taille fixe
- **Listes dynamiques (chainée):** allocation dynamique de la mémoire et implémentation en utilisant des pointeurs.

4. Liste statique

Une liste statique est représentée par un tableau de taille fixe avec une variable longueur contient le nombre d'éléments déjà ajouté dans la liste.

Exemple :

Soit la liste statique suivante avec un tableau de 7 éléments (taille fixe) et une variable longueur contient la valeur 4 qui est le nombre des éléments existant dans la liste.



Longueur =4

4.1. Définition du type liste

Type Structure Liste

début

Tab: Tableau[MAX] d'Éléments ;

longueur : Entier ;// garde le nombre d'éléments stockés dans la liste

Fin

Exemple: liste d'entiers

Type Structure ListeEntiers

début

Tab: Tableau[1000] d'entiers ;

longueur : Entier;

Fin

4.2. Déclaration en C++

```
struct liste {
        type_des_élements Tab [Max];
         int   longueur;
   }
```

Exemple: Déclaration d'une liste d'entiers

```
# include <iostream>
const int Max=1000;
struct liste {
   int Tab [Max];
   int longueur;
}
int main ( )
{   liste L1, L2 ;
. . .
}
```

4.3. Quelques opérations sur les listes statiques

Dans cette section nous présentons les opérations les plus utilisées sur les listes statiques. On peut ajouter d'autres selon les besoins.

1) **La procédure initialiser** : initialise la variable longueur d'une liste à la valeur 0. Elle crée donc une Liste vide.

```
Procédure Initialiser(var L:liste)
Début
      L.longueur ← 0;
Fin
En c++
void initialiser (liste &L) {
   L.longueur =0;
}
```

2) **La fonction est_vide** : une fonction booléen qui reçois une liste en entrée et retourne vrai si la liste est vide (Longueur = 0) et faux sinon.

```
Fonction Est_vide(L : Liste) : Booléen
Début
    Retourner (Longueur(l) == 0);
Fin
En C++
bool est_vide (liste L) {
  return (L.longueur == 0);
}
```

3) **La fonction est pleine:** teste si une liste est pleine ou non. Elle retourne vrai si la longueur de la liste égal sa taille maximale (longueur = max);

```
Fonction Est_pleine (L : Liste) : Booléen
Début
    Retourner (L.longueur==Max);
Fin
En C++ :
bool est_plein (Liste L){
  return(L.longueur==Max);
}
```

4) **La procédure ajouter** : Si la liste est non pleine, il ajoute un élément dans la liste et incrémente la variable longueur de 1 sinon elle ne fait rien.

Procédure Ajouter(x : Élément, var L : Liste)

Début

Si !est_plein(L) alors

L.Tab [L.Longueur+1] ← x

L.Longueur ← L.Longueur+1

Finsi

Fin

5) **La procédure insérer** : permet d'insérer un élément dans une position donnée (pos).

```
Procédure insererpos (var L : Liste, x, pos : entier)
    i :entier ;
Début
  Si (!est_plein(L)) alors
    Pour i allant de 1 à N faire
```

```
            L.Tab[i+1]= L.Tab[i];

      Fin pour

      L.Tab[pos]= x;
      L.longueur = L.longeur +1;
   Fin si
Fin

void insererpos (liste &L, int x, int pos) {
   if (!est_plein(L))
   {
      for (int i=L.Longueur; i>=pos; i--)
            L.Tab[i+1]= L.Tab[i];

      L.Tab[pos]= x;
      L.longueur++;
   }
}
```

4.4. Avantages et inconvénients

Avantages

+ Parcours et accès faciles à l'ième élément (accès direct).
+ Possibilité de recherche efficace si la liste est triée (par exemple, recherche dichotomique).

Inconvénients :

- Réservation, lors de la compilation, de la ***taille maximale*** imposée (limitée), ce qui manque de souplesse et d'économie.
- Inefficacité de la suppression et de l'insertion à l'intérieur de la liste : **obligation de décaler tous les éléments entre l'élément inséré ou supprimé et le dernier élément.**

5. Les structures de données dynamiques et l'allocation dynamique

Les structures dynamiques sont utilisées de façon très intensive en programmation. Leur but est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé a priori et peut évoluer. Les structures dynamiques doivent répondre à un certain nombre de critères.

1. Elles doivent permettre une bonne gestion de la mémoire, par un usage économe de celle-ci.
2. Elles doivent aussi être simples à utiliser, pour ne pas échanger l'économie de l'espace en mémoire contre une difficulté d'emploi, ou un temps d'exécution disproportionné.

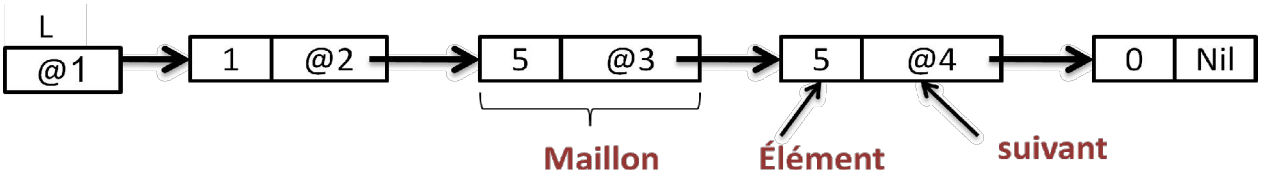
La solution que nous allons utiliser repose sur l'allocation de la mémoire au fur et à mesure des besoins. On dispose alors de toute la place nécessaire sans en gaspiller.

Allouer de la mémoire signifie réserver de la mémoire pour l'utilisation du programme. Après utilisation, on peut ensuite désallouer cette mémoire. Dans des langages tels que C/C++ ou PASCAL, il faut gérer soigneusement l'allocation et la désallocation de la mémoire.

6. Listes chainées

Une liste linéaire chaînée (LLC) est un ensemble de **maillons**, alloués dynamiquement, chaînés entre eux.

- Un **maillon** est un enregistrement avec deux champs : champ **Elément** contenant l'information. Champ **Suivant** donnant l'adresse du prochain maillon.
- Une liste chainée est représentée par un pointeur contenant l'adresse du premier élément (Maillon) de la liste.
- La liste vide, sans éléments, a pour référence **Nil**.
- ***Nil*** (NULL en C++) est utilisé dans le dernier maillon pour indiquer la fin de la liste.



6.1. Définition de type Liste

Type Structure Maillion

Ele : typeq; // typeq désigne un type quelconque (int, float, personne, étudiant ...etc.

suivant: * Maillion;

Fin

Type Liste : * Maillon; // le type liste désigne tout pointeurs vers un maillon

<u>Utilisation pour la déclaration</u>

L : Liste; équivalent à L: * Maillon;

L, P, Q : Liste // signifie que L, P, Q sont des pointeurs vers des maillons

6.2. Déclaration en C++

```
struct maillon {
   type_des_élements  ele;                    // déclaration d'un maillon en C++
   maillon *suivant;
};
typedef maillon *liste;     // définition de type liste
```

Exemple : Déclaration d'une liste chainée d'entier

```
# include <iostream>
struct maillon {
   int ele; // typeqq
   maillon *suivant;
}
typedef maillon *liste;
int main ( )
{   Liste L, L1 ;
. . .
}
```

6.3. Quelques opérations sur les listes chainées

6.3.1. Cree_maillon: crée un nouveau maillon contient comme valeur x et retourne un pointeur contenant son adresse

```
Fonction creer_maillon (x : typeq): Liste
        P: Liste              //  ou P: *Maillion
Début
   P ← Allouer (Maillon)
   P - > Élément ← x;
   P -> suivant   ← Nil;
   Retourner (P);
Fi n
```

6.3.2. Opération permettant de tester si une liste est vide

Une liste est vide si sa tête pointe la valeur NULL. Cette fonction retourne un booléen indiquant si la liste est vide ou non.

```
Fonction Est_vide(L : Liste) : Booléen
Début
    Si (L=Nil) alors
      Retourner ( vrai);
   Sinon
       Retourner (faux);
   Finsi
Fin
En c++ :
bool estVide (Liste L)                          bool estVide (Liste L) {
{                                                       if (L==NULL)  return true;
    return (L== NULL);              ⇔                   else    return false;
}                                                      }
```

6.3.3. Opération permettant de retourner le premier élément d'une liste

Cette opération retourne l'élément qui existe dans la tête d'une liste non vide.

```
Fonction Premier (l: liste): Élément,
Début
     Retourner (L -> ele);
Fin

En C++ :
type premier (liste L)
{
    return (L -> ele);
}
```

6.3.4. Opération d'ajout d'un élément en tête d'une liste

Cette opération permet de coller un nouvel élément au début d'une liste. Par conséquent l'élément ajouté devient le premier élément de la liste.

L'opération **ajouter** comme fonction :

```
Fonction Ajouter(x : typeq, L: Liste): Liste.
 P: Liste              //  ou P: *Maillion
Début
    P ← creer_maillon (x);
    P -> suivant   ← L;
    Retourner (P);
Fi n
Liste ajouter (Liste l, type ele)
{
    Liste P;
    P = new Maillon ;
    P->suivant =L ;
    return  p ;
}
```

L'opération **ajouter** comme procédure

```
Procédure Ajouter(x : Élément, var L: Liste)
 P: Liste
Début
   P ← creer_maillon (x);
    P -> suivant   ← L;
    L← P;
Fi n

void ajouter (Liste &l, type ele)
{
    Liste P;
    P=new Maillon ;
    P->suivant =L ;
    L=P ;
}
```

6.3.5. Opération permettant de retourner le reste d'une liste

C'est-à-dire une liste sans sa tête. Cette fonction retourne l'adresse du deuxième élément de la liste.

```
Fonction Reste (L: Liste): Liste
Début
   Retourner (L -> suivant);
Fi n
En C++ :
liste  reste (Liste L)
{
    return L->suivant;
}
```

6.3.6. Opération permettant de calculer la longueur d'une liste

Cette fonction retourne le nombre d'élément de la liste L. Nous allons définir deux versions de cette fonction : la première est récursive tandis que la deuxième est itérative.

```
// Version récursive
fonction longueur (L : Liste) : Liste
début
    Si estVide (L) alors
            return 0 ;
    else
          return 1 + longueur (reste(L)) ;
fin
// Version récursive
fonction longueur (L : Liste) : Liste
    Courant : Liste
    n : entier ;
Début
   Courant = L ;  n=0 ;
    Tantque  (courant != NULL)
    {
            n++;
            courant = courant->suivant ;
    }
    Retourner n ;
}
```

```
En C++ :
// Version récursive
int longueur (liste L)
{
    if (estVide (L))
            return 0 ;
    else
          return 1 + longueur (reste(L)) ;
}
// Version récursive
int longueur (liste l)
{   liste courant = L ;  int  n=0 ;
    while (courant != NULL)
    {
            n++;
            courant = courant->suivant ;
    }
    return n ;
}
```