École Polytechnique Sousse
Département Informatique
Section : Génie Informatique
Niveau : 3^{eme} Année , AU : 2020-2021

LANGAGE C

Les listes simplement chaînées

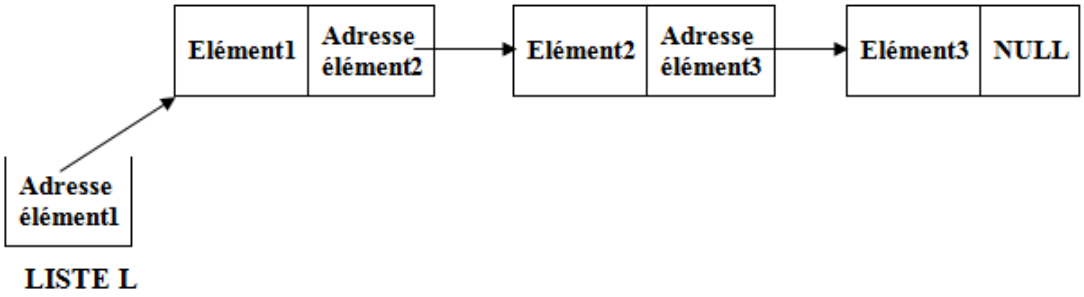
Dans un tableau, la taille est connue, l'adresse du premier élément aussi. Lorsque vous déclarez un tableau, la variable contiendra l'adresse du premier élément de votre tableau. Comme le stockage est contigu, et la taille de chacun des éléments connus, il est possible d'atteindre directement la case i d'un tableau.

Dans une liste chaînée, la taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet. Il est en revanche impossible d'accéder directement à l'élément i de la liste chaînée. Pour ce faire, il vous faudra traverser les éléments précédents de la liste.

Une liste chaînée est une application typique de l'allocation dynamique de mémoire. C'est pourquoi il est judicieux d'avoir des connaissances sur les pointeurs, les structures et l'allocation dynamique de mémoire pour aborder aux listes chaînées.

1 Définition

Une liste simplement chaînée (LSC) est une liste d'éléments à chaînage simple où à partir d'un élément on peut accéder à son suivant. A chaque élément de la liste est associé, en mémoire, un emplacement représentant un enregistrement contenant au moins deux champs où le premier contient l'élément et le deuxième contient l'adresse de l'emplacement mémoire de l'élément suivant. C'est un pointeur sur un enregistrement de même type. En l'absence d'un élément suivant, le champ pointeur contiendra la constante NULL.
En mémoire, une liste chaînée simple a la représentation suivante :



2 Caractéristiques d'une liste simplement chaînée

2.1 Quelques Principes

— Une liste simplement chaînée est utilisée pour stocker des données qui doivent être traitées de manière séquentielle

— Les éléments de la liste appelés cellule, noeuds ou maillons, ne sont pas rangés les uns à côté des autres
— On a besoin de connaître, pour chaque élément, l'adresse de l'élément qui le suit
— On dispose d'un pointeur de tête qui contient l'adresse du premier élément de la liste
— le dernier noeud ne pointe vers rien, il est nécessaire de renseigner son pointeur avec la valeur NULL

2.2 Inconvénients

Contrairement au tableau, pour accéder à un élément il faut parcourir la liste.

2.3 Avantages

— L'insertion est plus rapide
— La destruction au milieu est plus rapide

3 Création d'une liste simplement chaînée

3.1 Définition

Syntaxe :

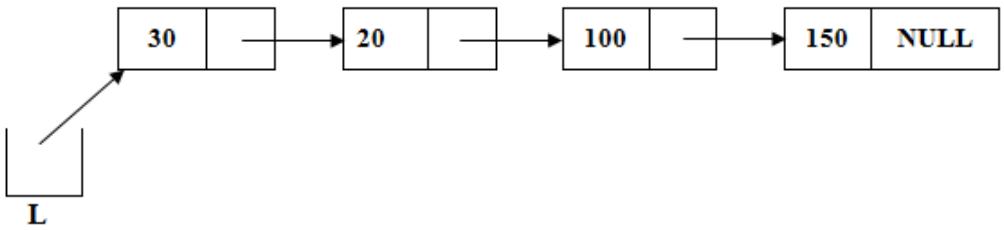
```
struct <Nom_Cellule>
{
    // Déclaration des champs de la cellule
    // pointeur sur l'élément suivant
};

typedef struct <Nom_Cellule> * <Pointeur_Sur_Premier_Elément>;
```

Exemple : Liste chaînée d'entiers

```
struct cellule
{
    int info ;
    struct cellule *suivant ;
};

typedef struct cellule* liste;/*"struct cellule*" est synonyme à "liste"*/
```



3.2 Déclaration

liste L ;

3.3 Création d'une liste vide

Pour créer une liste vide, il suffit d'affecter la valeur NULL à la variable L qui est de type "pointeur sur Liste" : L = NULL ;

3.4 Création d'une liste contenant un seul élément

Les différentes étapes à suivre sont :

1. Déclarer tout d'abord le pointeur

```
struct cellule * nouveau;
            ou
liste nouveau;
```

2. Allouer la mémoire nécessaire au nouveau maillon grâce à la fonction malloc

```
nouveau=(struct cellule*)malloc(sizeof(struct cellule));
```

3. Renseigner ce nouveau maillon

```
nouveau->info = Val ; //Val de type entier
nouveau->suiv = NULL ;
```

4. Définir le nouveau maillon comme maillon de tête de liste
   L = nouveau ;

4 Ajout d'un élément à une liste simplement chaînée

L'opération ajout, appelée aussi insertion, peut être effectuée de trois manières :
— Insertion en tête de liste,
— Insertion en fin de liste,
— Insertion à une position donnée.

4.1 Ajout d'un élément au sommet de la liste

```
liste ajout_tete(liste l, int x )
{
    liste p;
    p=(struct cellule*)malloc(sizeof(struct cellule));
    p->suivant=l;
    p->info=x;
    l=p;
    return l;
}
```

4.2 Ajout d'un élément en queue de la liste

```
liste ajout_queue(liste l, int x )
{   liste p1,p2;
    p1=(struct cellule*)malloc(sizeof(struct cellule));
    p1->info=x;
    p1->suivant=NULL;
    if(L==NULL)
```

```
        L=p1;
    else
    {
        p2=L;
        while(p2->suivant!=NULL)
            p2=p2->suivant;
        p2->suivant=p1;
    }
    return L;
}
```

4.3 Insertion d'un élément à une position donnée (avec pos<>de 1 et pos <> taille(l)+1)

```
void ajout_position(liste l, int pos,int x )
{ liste p2,p1=l;
  int i=1;
  if(pos!=1 && pos!=TAILLE(l)+1)
  {
    while(i<pos-1)
    {
      p1=p1->suivant;
      i++;
    }
    p2=(struct cellule*)malloc(sizeof(struct cellule));
    p2->info=x;
    p2->suivant=p1->suivant;
    p1->suivant=p2;
  }
}
```

5 Taille d'une liste simplement chaînée

```
int taille(liste l)
{
    liste p=l;
    int t=0;
    while(p!=NULL)
    {   t++;
        p=p->suivant;
    }
    return  t;
}
```

6 Parcours de la liste : Afficher les éléments de la liste

```
void affiche(liste l)
{
    if l == NULL)
```

```
        printf("Liste Vide");
    else
        while(l!=NULL)
        {
            printf("%d\t",l->info);
            l=l->suivant;
        }
}
```

7 Suppression d'un élément de la liste

L'opération de suppression, peut être effectuée de quatre manières :
- Suppression de l'élément en tête de liste,
- Suppression de l'élément en fin de liste,
- Suppression de l'élément à une position donnée,
- Suppression d'un élément donné.

7.1 Suppression d'un élément au début de la liste

```
liste suppression_tete(liste l)
{
    liste p=l;
    if(l==NULL)
        printf("liste vide\n");
    else
    {   l=l->suivant;
        free(p);
    }
    return(l);
}
```

7.2 Suppression d'un élément en queue de la liste

```
liste suppression_queue(liste l)
{   liste p=l;
    if(l==NULL)
        printf("Suppression impossible\n");
    else
        if(l->suivant==NULL)
        {   l=l->suivant;
            free(p);
        }
        else
        {   while((p->suivant)->suivant != NULL )
                p=p->suivant;
            free(p->suivant);
            p->suivant=NULL;
        }
    return l;
}
```

7.3 Suppression d'un élément à une position donnée(avec pos<>de 1 et pos <> taille(l))

```
void suppression_position(liste l,int pos)
{   liste p1=l,p2;    int i=1;
    if(pos !=1 && pos!=taille(l) )
    {
        while ( i<pos-1)
        {   i++;
            p1=p1->suivant;
        }
        p2=p1->suivant;
        p1->suivant=p2->suivant;
        free(p2);
    }
}
```

7.4 Suppression d'un élément donnée

```
liste suppression_element(liste l, int x )
{   liste p=l,q;    int ok =0;
    if(p->info==x)
    {   l=p->suivant;
        free(p);
    }
    else
    {  while(p->suivant!=NULL)
            if((p->suivant)->info==x)
            {   ok=1;
                break;
            }
            else
                p=p->suivant;
        if(ok)
        {   q=p->suivant;
            p->suivant=q->suivant;
            free(q);
        }
        else
            printf("%d n'existe pas dans la liste\n",x);

    }
    return l;
}
```