Chapitre 5
Les listes simplement chaînées

Module 5: Algorithmique et programmation

Master Mathématiques, Cryptologie et Sécurité Informatique (MMCSI)

mlahby@gmail.com

22 novembre 2014

Plan

Introduction

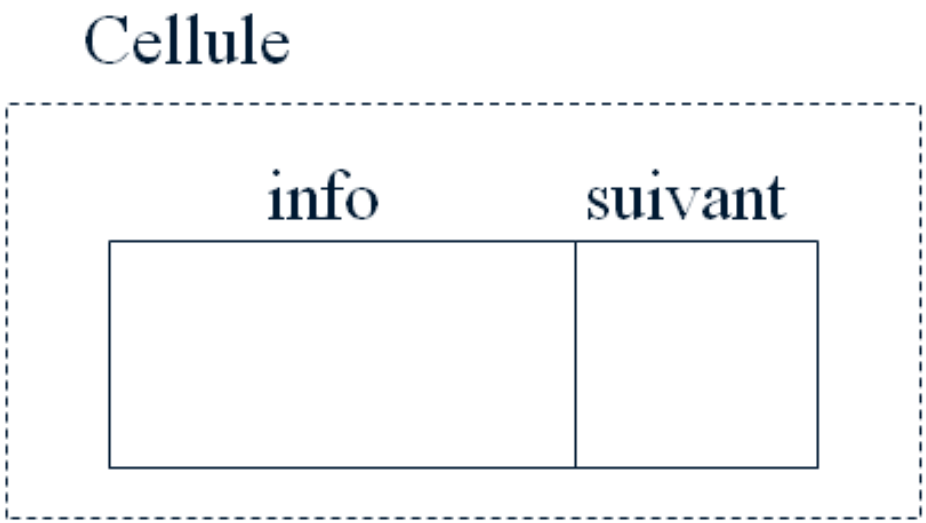
Pour stocker des données en mémoire, nous avons utilisé des variables simples (type int, double. . .), des tableaux et des structures personnalisées. Si vous souhaitez stocker une série de données, le plus simple est en général d'utiliser des tableaux. Toutefois, les tableaux se révèlent parfois assez limités. Par exemple, si vous créez un tableau de 10 cases et que vous vous rendez compte plus tard dans votre programme que vous avez besoin de plus d'espace, il sera impossible d'agrandir ce tableau. De même, il n'est pas possible d'insérer une case au milieu du tableau.
Les modèles de données dynamiques (liste chaînée, pile, file...) représentent une façon d'organiser les données en mémoire de manière beaucoup plus exible. Comme à la base le langage C ne propose pas ce système de stockage, nous allons devoir le créer nous-mêmes de toutes pièces. Les modèles de données dynamiques Une structure dynamique est une structure dont la taille peut varier au cours de l'exécution du programme , l'accès aux informations stockées dépend de la structure. Ce type de structure utilise des pointeurs, dans ce chapitre nous allons traiter un seul type de structures dynamiques : liste chainée.

Définition

- Une liste est une séquence finie d'éléments ;
- Les éléments sont repérés selon leur rang ;
- Il existe une relation d'ordre sur le rang des éléments : le rang du premier élément est 1, le rang du second est 2, ...
- L'ajout et la suppression d'un élément peut se faire à n'importe quel rang valide de la liste
- Une liste chaînée est un moyen d'organiser une série de données en mémoire ;
- Cela consiste à assembler des structures en les liant entre elles à l'aide de pointeurs.

Listes : cellule

- La cellule est la composante principale d'une liste ;
- Une liste est un ensemble de cellules (éléments) chaînées entre elles à l'aide de liens ;
- Une cellule est une structure qui comporte deux champs :
  1. Le champ info : il contient des informations sur l'élément représenté par la cellule ;
  2. Le champ suivant : il représente un pointeur qui contient l'adresse de la cellule suivante.



Déclation d'une liste

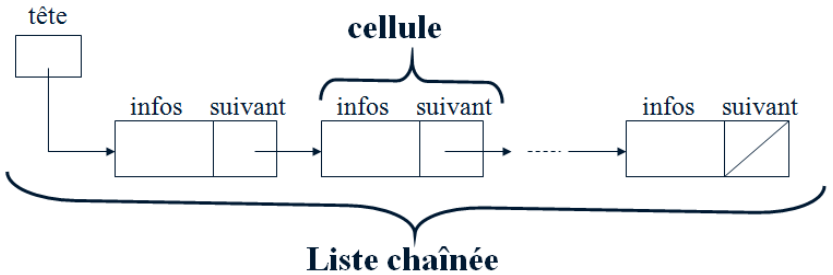
Syntaxe pour définifir le type liste

```
typedef struct cellule{
  int info ; //le champ info peut avoir n'importe quel type
  struct cellule *suiv ; //pointeur contenant l'adresse de la cellule suivante
}liste ;
```

Déclaration d'une liste

Pour créer une liste chaînée, il suffit de déclarer un pointeur qui contient l'adresse de la première cellule :

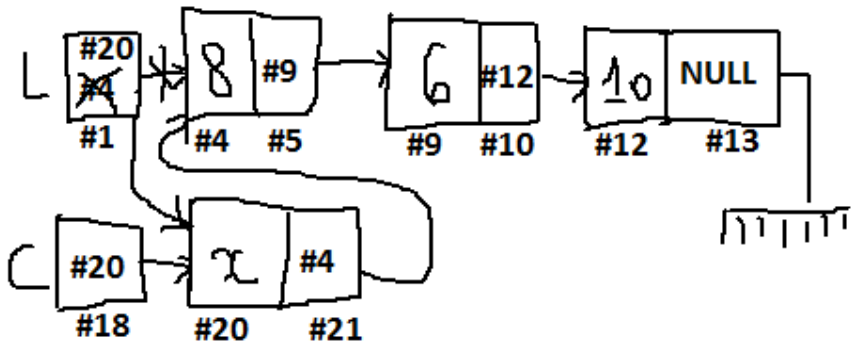
```
liste *L ;
```



Prototype : InsertDebut(liste *L, int x)

Définition de la fonction

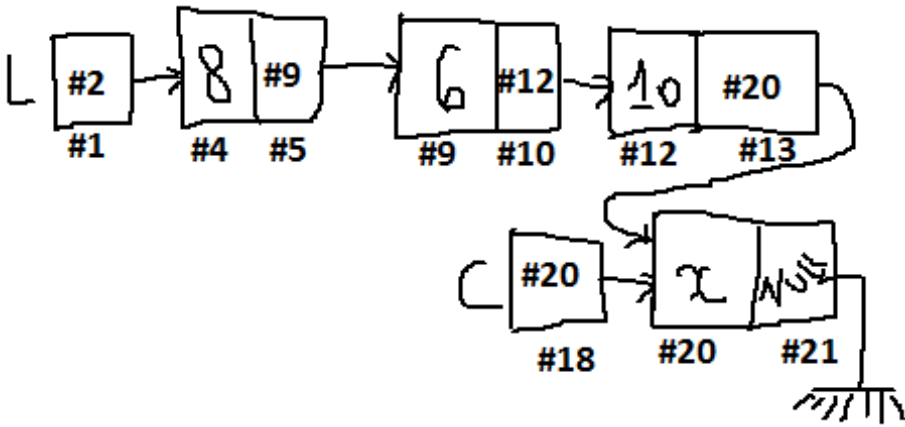
```
liste *InsertDebut(liste *L, int x)
{ liste *c;
c=(liste*)malloc(1*sizeof(liste));
if(c!=NULL)
{
      c->info = x;
      c->suiv = L;
      L=c;
      return(c);
}
else
      return(L);
}
```



Prototype : InsertFin(liste *L, int v)

Définition de la fonction

```
liste *InsertFin(liste *L, int x)
{ liste *c;
c=(liste*)malloc(1*sizeof(liste));
if(c!=NULL)
{    c->info = x;
   c->suiv = NULL
}
if(L! = NULL) //liste n'est pa vide
//Chercher l'adresse de la derniere
{    P=L;
   while(P->suiv! = NULL)
      P = P->suiv;
//brancher
P->suiv = C;
return L; }
else
L=C;
return L;
}
```



Prototype : InsertOrd(liste *L, int x)

Fonction itérative

```
liste *InsertOrd(liste *L, int x)
{ liste *C,*P,*N ;
while(C! = NULL&&C− > info! = x)
{P = C;
C = C− > suiv; }
N = (liste∗)malloc(sizeof (liste));
N− > info = x;
if (P == NULL)
L = insertdebut(L, x);
else
{   P− > suiv = N;
N− > suiv = C;
}
return(L) ;
}
```

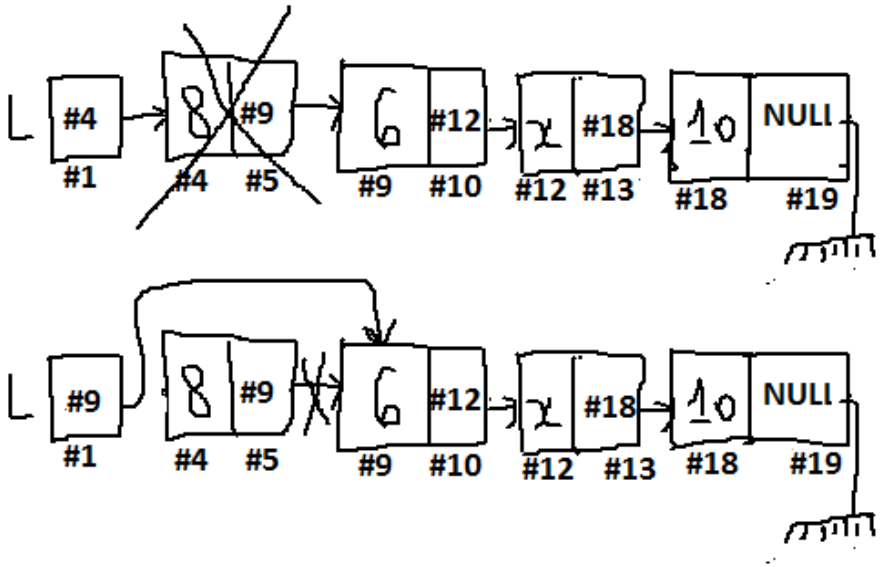
Fonction récursive

```
liste *InsertOrdR(liste *L, int x)
{
if(L !=NULL)
if (x < L− > info)
return(InsertDebut(L, x));
else
return(InsertDebut(L− > info, InsertOrdR(L− >
suiv, x)));
else
return(InsertDebut(L, x));
}
```

Prototype :SupprimerDebut(liste *L)

Définition de la fonction

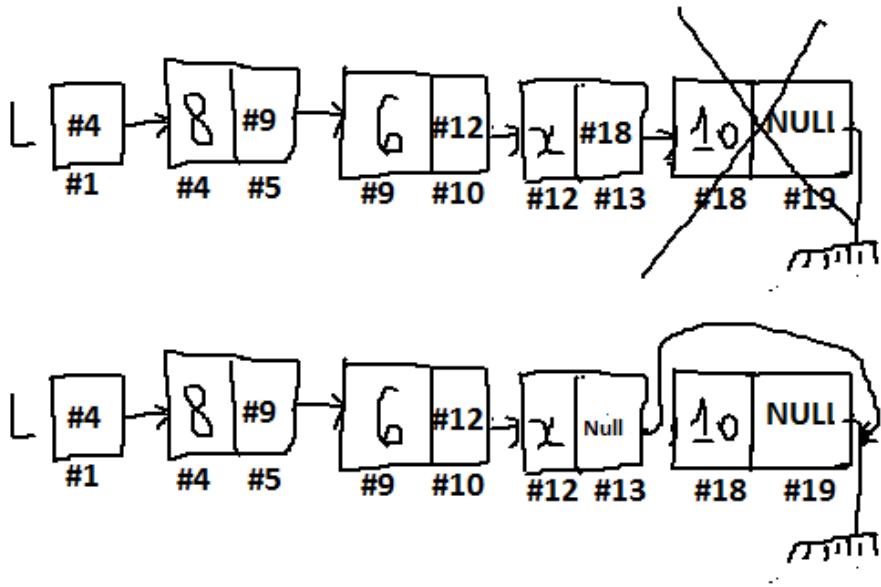
```
liste *SupprimerDebut(liste *L)
{ liste *p ; //pour libérer la cellule
qu'on va supprimer
if(L !=NULL)
{P = L;
L = L− > suiv;
free(p);
return(L);
}
else
return(NULL) ; // où return(L) ;
}
```



Prototype : SupprimerFin(liste *L)

Définition de la fonction

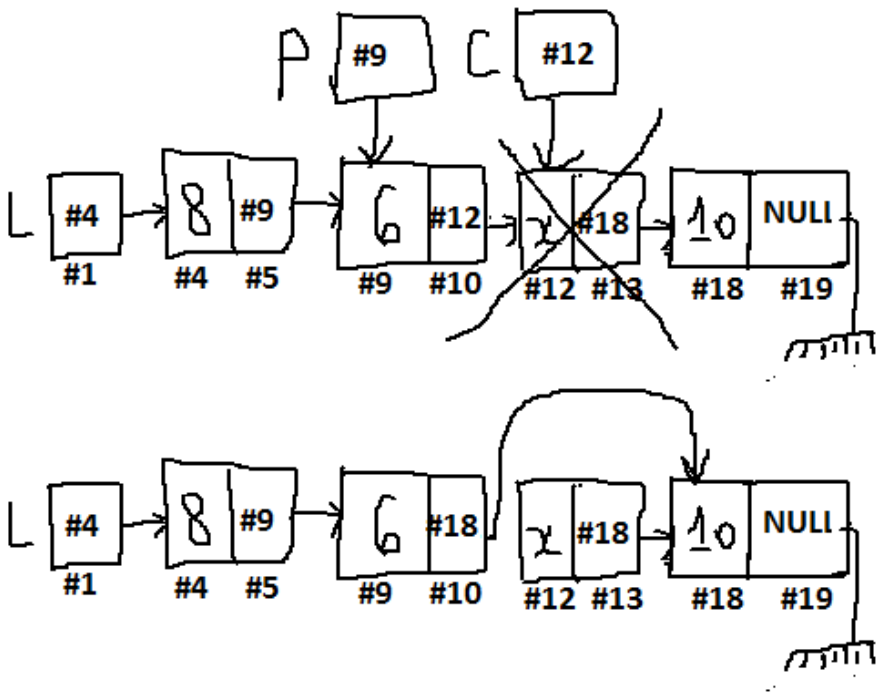
```
liste *SupprimerFin(liste *L)
{ liste *p,*c ;
p=NULL ; //pour mémoriser l'adresse
if(L! = NULL) //liste n'est pa vide
{c = L; //pour parcourir la liste
while(c− > suiv! = NULL)
{p = c;//contient l'adress de l'avt dernier
c = c− > suiv;}
if(p == NULL)//L contient un seul élément
{L = L− > suiv;
free(c);}
else
{
P− > suiv = NULL;
free(c);
}
return(L);
}
```



Prototype : SupprimerVal(liste *L, int x)

Définition de la fonction

```
liste *SupprimerVal(liste *L, int x)
{ liste *C,*P ;
if(L !=NULL){
if (L- > info == x)
{C = L;
L = L- > suiv;
free(C); //liberer la cellule
}
else
{C = L;
while(C! = NULL&&C- > info! = x)
{P = C;
C = C- > suiv;}
if (C! = NULL)
{P- > suiv = C- > suiv;
free(C);}
} }
return(L) ;
}
```



Prototype : AfficherListe(liste *L)

Fonction itérative

```
void AfficherListeIt(liste *L)
{ liste *P ;
P=L ; //pour sauvegarder la tête de L
while(P! = NULL)
{
printf("%d− > ", P− > info);
P = P− > suiv;
}
printf("NULL") ;
}
```

Fonction récursive

```
void AfficherListeRe(liste *L)
{
if(L! = NULL)
{
printf("%d− > ", L− > info);
AfficherListeRe(L− > suiv)
}
}
```

Prototype : Rechercher(liste *L,int x)

Fonction itérative

```
int RechercherIt(liste *L,int x)
{ liste *P ;
P=L ; //pour sauvegarder la tête de L
while(P != NULL&&P-> info != x))
{
P = P-> suiv;
}
if (P != NULL)
return(1);
else
return(0);
}
```

Fonction récursive

```
int RechercherRe(liste *L,int x)
{
if (L == NULL)
return(0);
if (L != NULL&&L-> info == x)
return(1);
return(RechercherRe(L-> suiv, x));
}
```

Prototype : InverserListe(liste *L, int x)

Fonction itérative : on construit une nouvelle liste

```
liste *InverserListeIt(liste *L)
{ liste *LI ;
LI=NULL ; //liste vide
while(L! = NULL)
{
LI = InsertDebut(LI, L− > info);
L = L− > suiv;
}
return(LI) ;
}
```

Fonction récursive : on affiche la liste sans constuire une nouvelle liste

```
void InverserListeRe(liste *L)
{
if(L! = NULL)
{
AfficherListeRe(L− > suiv);
printf("%d− > ", L− > info);
}
}
```

Prototype : TriSelection(liste *L)

Solution 1 : Boucle for

```
void TriSelection(liste *L)
{ liste *p,*q,*min ;
int aide ;
for(p = L; p- > suiv! = NULL; p = p- > suiv)
{min = p;
for(q = p- > suiv; q! = NULL; q = q- > suiv)
{
if (q- > info < min- > info)
min = q;
}
aide = p- > info;
p- > info = min- > info;
min- > info = aide;
}
}
```

Solution 2 : Boucle while

```
void TriSelection(liste *L)
{ liste *p,*q,*min ;
int aide ;
p=L ;
while(p- > suiv! = NULL)
{min = p;
q = p- > suiv;
while(q! = NULL)
{
if (q- > info < min- > info)
min = q;
q = q- > suiv;
}
aide = p- > info;
p- > info = min- > info;
min- > info = aide;
p = p- > suiv;
}
}
```