LES LISTES CHAINEES

1 Définition

Pour stocker une collection d'informations et s'affranchir du problème posé lors de la réservation statique de place mémoire (effectuée lors de la déclaration du tableau par exemple), on utilise une structure de donnée appelée liste chaînée. Une liste chaînée est constituée d'un ensemble de cellules chaînées entre elles qui contiennent les informations à mémoriser. L'emplacement mémoire nécessaire à chaque cellule est alloué dynamiquement. Dans une version simple de liste chaînée, chaque cellule comprend l'élément de la liste à mémoriser et l'adresse de la cellule suivante. La dernière cellule contient un pointeur qui contient une adresse nulle, ce qui indique la fin de la liste. C'est l'adresse de la première cellule qui détermine la liste.

2 Implantation d'une liste chaînée (en langage C)

On suppose dans la suite que les valeurs à mémoriser sont de type réel, mais évidemment tout autre type de données est possible (même tableaux, structures, listes,…) On définit un type structure cellule regroupant une valeur réelle et un pointeur.

```
struct cellule

{

float val ;

struct cellule * suivant;

} ;
```

On considère une liste L tel que L est de type struct cellule * (pointeur sur une cellule)

// Initialisation de la liste vide

```
void  initialiser_liste (struct cellule * L)

{

  L  = Null ;

}
```

// Affichage de la liste

```
void afficher_liste (struct cellule *  L)

{

struct cellule * p ;

p = L ;

while (p!=NULL)

{
```

```
printf("%f ", p -> val);
p = p -> suivant;
}
printf("\n");
}
```

// Insertion d'une nouvelle cellule en début de liste

```
struct cellule *  inserer_premier (struct cellule * L, float vale)
{
struct cellule * nouveau ;
nouveau = (struct cellule *) malloc (sizeof(struct cellule)) ;
nouveau -> val = vale;
nouveau -> suivant = L;
L = nouveau ;
Return (L) ;
}
```

// Insertion d'une nouvelle cellule en fin de liste

```
void inserer_fin (struct cellule * L, float vale)
{
struct cellule * nouveau , * parcours;
nouveau = (struct cellule *) malloc (sizeof(struct cellule)) ;
nouveau -> val = vale;
nouveau -> suivant = NULL;
if (L == NULL)
{
L= nouveau ;
}
else
{
parcours = L ;
```

```
while ( parcours -> suivant !=NULL)
{
parcours = parcours -> suivant ;
}
parcours ->suivant = nouveau ;
}
}
```

// Suppression d'une cellule en début de liste

```
float enlever_premier (struct cellule * L)
{
struct cellule * inter;
float vale ;
if (L != NULL)
{
inter = L ;
L= inter -> suivant;
vale = inter -> val ;
free(inter) ;
return (vale) ;
}
else
{
return(-1);
}
}
```

// Suppression d'une cellule en fin de liste

```
float enlever_fin (struct cellule *  L)
{
struct cellule * pred, * parcours;
```

```
float vale ;
if (L != NULL)
{
parcours = L ;
pred = L ;
while ( parcours -> suivant !=NULL)
{
pred = parcours ;
parcours = parcours -> suivant ;
}
vale = parcours -> val ;
if (parcours == L)
{
L= NULL ;
}
else
{
pred -> suivant = NULL;
}
free(parcours);
return (vale);
}
else return (-1);
}
```

// Recherche d'une valeur dans la liste

```
int recherche (struct cellule * L, float vale)
{
struct cellule * parcours;
if (L == NULL)
```

```
{
return(0) ;
}
else
{
parcours = L ;
while (( parcours -> suivant !=NULL) && (parcours -> val !=
vale))
{
parcours = parcours -> suivant ;
}
if (parcours -> val == vale) return (1);
else return (0);
}
}
```

// Suppression d'une valeur dans la liste

```
void supprime(struct cellule *  L, float vale)
{
struct cellule * pred, * parcours;
if (L != NULL)
{
parcours = L ;
pred = L ;
while ((parcours -> suivant !=NULL)&&(parcours -> val !=
vale))
{
pred = parcours ;
parcours = parcours -> suivant ;
}
```

```
if (parcours -> val == vale)
{
if (L== parcours)
L = parcours ->suivant;
else pred->suivant = parcours-> suivant ;
free (parcours) ;
}
}
}
```