IAP2 – Cours n°4 – Novembre 2007 - ENSIIE

C. DUBOIS

Allocation dynamique de la mémoire

2 types de données dans un programme :

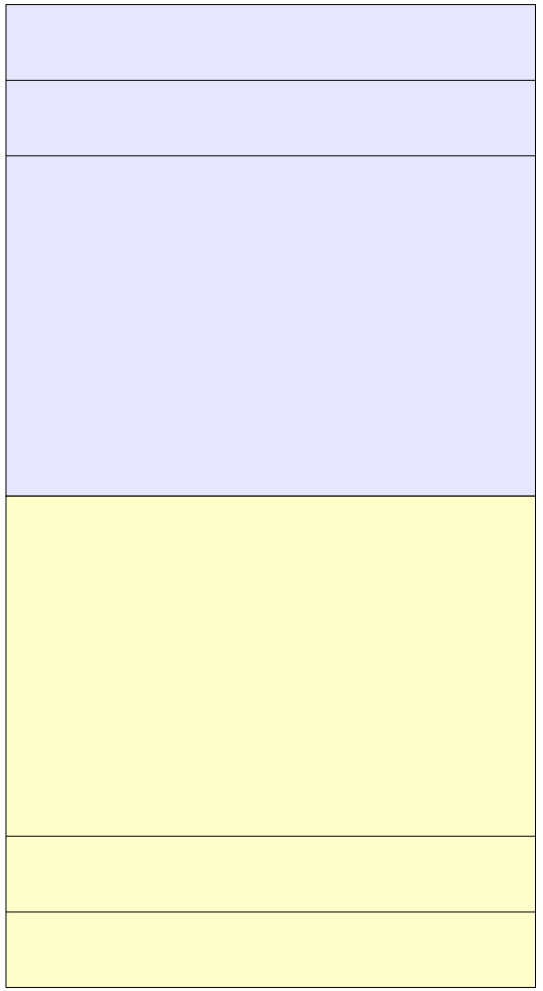
- données statiques

on connaît à l'avance (lors de l'écriture du programme) la taille des données
la mémoire peut être réservée à l'avance, avant l'exécution du programme

- données dynamiques

on ne connaît la taille des données qu'à l'exécution du programme
la réservation de la mémoire se fait au cours de l'exécution du programme

Modèle de mémoire



Données statiques

Données dynamiques : le tas

- Outils de gestion de la mémoire « dynamique »

pour réserver de la mémoire
    d'une taille donnée
    d'un type donné

pour libérer de la mémoire

- Utilisation des pointeurs pour accéder à ces espaces mémoires alloués dynamiquement

Allocation mémoire

void* malloc(size_t t)

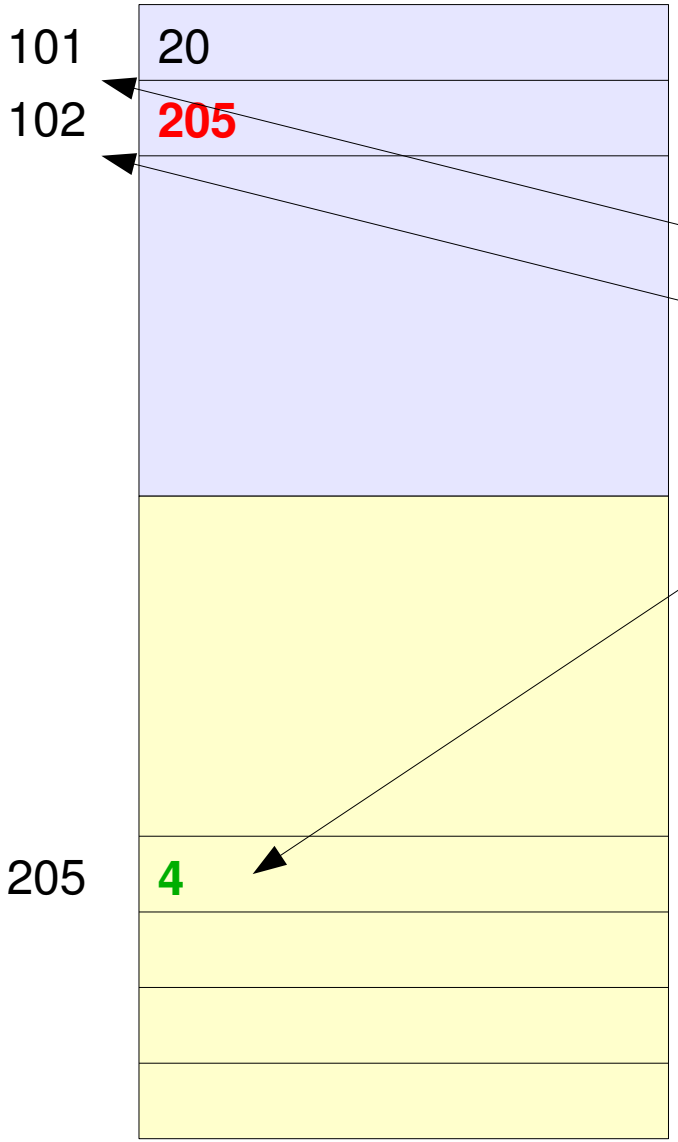
renvoie un pointeur sur l'espace alloué de la taille t
renvoie NULL si l'allocation a échoué

fonction de la librairie standard #include <stdlib.h>
size_t type non signé pour manipuler des tailles d'espaces mémoire

On peut utiliser la fonction sizeof pour obtenir la taille associée à un type

Exemple :

int *p; (ici p est non intialisé)
p=malloc(sizeof(int));
p pointe sur un espace mémoire pouvant accueillir un entier



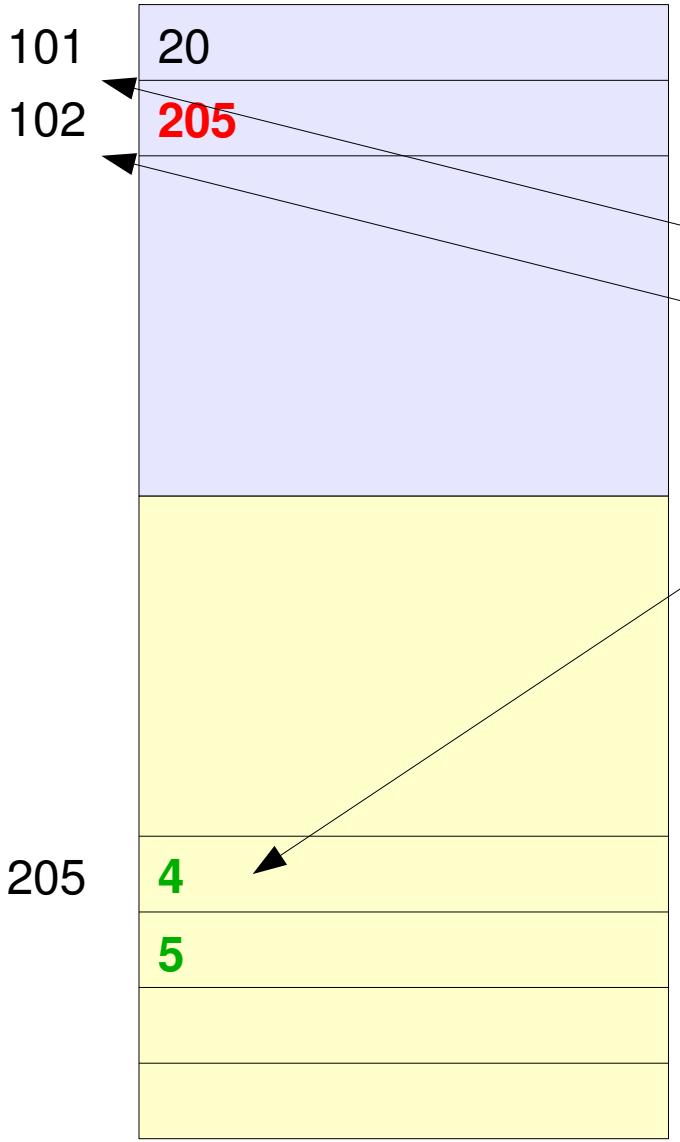
int i=20;

int *p;

p=malloc(sizeof(int));

***p = 4**

le contenu de la zone mémoire allouée vaut 4



int i=20;

int *p;

p=malloc(4*sizeof(int));

***p = 4**

***(p+1)=5**

le contenu de la zone mémoire allouée vaut 4

Mémoire pleine ?

Lorsque la mémoire dynamique a complétement été allouée, **malloc** renvoie NULL

Il faut tester cette valeur à chaque allocation de mémoire

```
int *p1, *p2;
p1 = malloc(sizeof(int));
if (p1 == NULL)
exit(0);
```

```
#include <assert.h>
int *p1;
p1 = malloc(sizeof(int));
assert(p1 != NULL);
...
```

Il existe d'autres fonctions d'allocation : calloc, realloc

void* calloc(size_t nb, size_t t)

alloue nb blocs consécutifs de taille t,
renvoie NULL si l'allocation a échoué

Exemple :

int *p; (ici p est non intialisé)
p=calloc(4,4*sizeof(int));
p pointe sur un espace mémoire de 4 blocs de mémoire pouvant accueillir 4 entiers

void* realloc(void* ptr, size_t t)

pour agrandir ou rétrécir une zone mémoire déjà allouée dynamiquement, pointée par ptr

renvoie NULL si l'allocation a échoué, ptr n'est pas modifié
renvoie l'adresse de la nouvelle zone allouée

si nouvelle taille > ancienne taille
    l'ancien contenu est intégralement conservé
si nouvelle taille < ancienne taille
    seuls les premiers éléments du début sont conservés

Libération de la mémoire

void free(void *ptr)

libère l'espace pointée par ptr
la mémoire doit avoir été préalablement allouée dynamiquement
n'a pas d'effet si ptr a la valeur NULL

Attention !

* La fonction free peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par malloc.

* La fonction free ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur NULL au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.

* Si nous ne libérons pas explicitement la mémoire à l'aide free, alors elle est libérée automatiquement à la fin du programme.

Représentation des listes

- pas de type prédéfini en C

- implantation possible à l'aide d'un tableau

    si on peut évaluer la taille maximale de la liste
    utilisation d'un tableau surdimentionné

On parle alors de *représentation contingüe*

- ou représentation avec des pointeurs

On parle alors de *listes chaînées.*

- Représentation contigüe

soit T un type

```
typedef struct liste{
   int tete
   T tab[N ]
}  tListe
```

tete : indice de la tête de la liste : un entier compris entre -1 et N-1
-1 pour signifier que la liste est vide

La liste croît vers la droite

Fonction qui retourne l'élément de tête d'une liste d'éléments de type T

```
T head( tListe l){
if (l.tete==-1)
exit(0);
else return(l.tab[l.tete]);
}
```

Procédure qui ajoute un élément en tête de la liste

Ce n'est possible que si on ne dépasse pas la longueur maximale
----> listes bornées

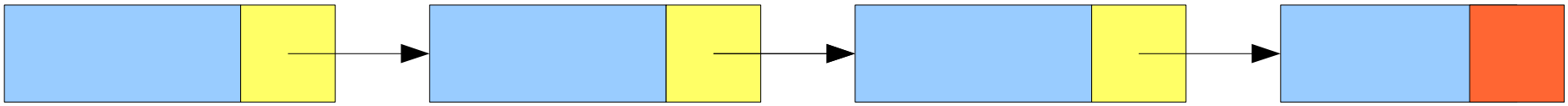
```
void cons(tListe l, T nvelt){
if (l.tete==N-1)
exit 0;
else {
l.tab[l.tete+1]=x;
l.tete=l.tete+1;
}
}
```

Fonction qui retourne la longueur d'une liste

```
int lg(tListe l){
return(l.tete+1);
}
```

- Représentation chaînée : vers plus de dynamicité

Chaque élément contient
- des données
- un lien (pointeur) vers l'élément suivant



Un élément = 1 structure avec 2 champs : un champ qui contient les données + 1 champ qui contient un pointeur sur l'élément suivant

```
typedef struct element{
   T data
   struct element * suivant
}  telement
```

structure de données récursive : auto-référence

Une liste ?

On représente une liste non vide par l'adresse de son premeir élément
On représente une liste vide par le pointeur NULL

type d'une liste chaînée **: telement *** **typedef telement* liste**

Les éléments sont chaînés les uns aux autres

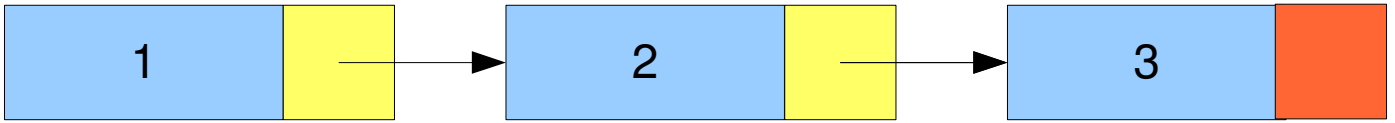
reste de la liste l : ***l.suivant**

données du 2ème élément de la liste l : ***(*l.suivant).data**
ou **(l->suivant)->data**

dernier élément === son champ suivant est NULL
parcourir toute la liste

```
telement* dernier(liste l){
telement * ptr=l;
while (*ptr.suivant!=NULL)
      ptr=*ptr.suivant;   ou ptr=ptr->suivant;
return(ptr);
}
```

Création d'une liste



La mémoire est allouée élément par élément :

On alloue le dernier élément **p3 = malloc(sizeof(telement));**
On remplit ses champs : ***p3.data = 3;** **ou** **p3->data = 3;**
***p3.suivant=NULL;** **ou** **p3->suivant=NULL;**

On recommence avec le deuxième élément
p2 = malloc(sizeof(telement));
***p2.data = 2;** **ou** **p2->data = 2;**

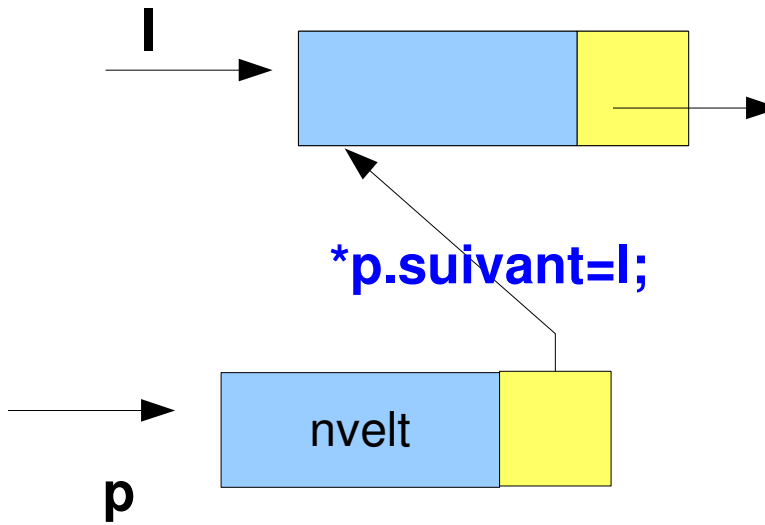
On chaîne le 2eme élément au dernier : ***p2.suivant=p3;** **ou** **p2>suivant=p3;**

Le premier : **p1 = malloc(sizeof(telement));**
***p1.data = 1;** **ou** **p1->data = 1;**
***p1.suivant=p2;** **ou** **p1->suivant=p2;**

Ajout d'un élément en tête

/* interface cons
type : liste*T -> liste
arg l, nvelt (données du nouveau élément)
pre aucune
post ajoute un élement contenant nvelt en tête de la liste l, retourne l'adresse du nouveau premier élément*/

liste cons (liste l, T nvelt){
***telement p;**
On alloue le nouvel élement, soit p son adresse
p = malloc(sizeof(telement));
On remplit son champ data
***p.data = nvelt;**
On relie ce nouvel élement à la liste l
***p.suivant=l;**
Le résultat est le nouvel élément
return(p);
}



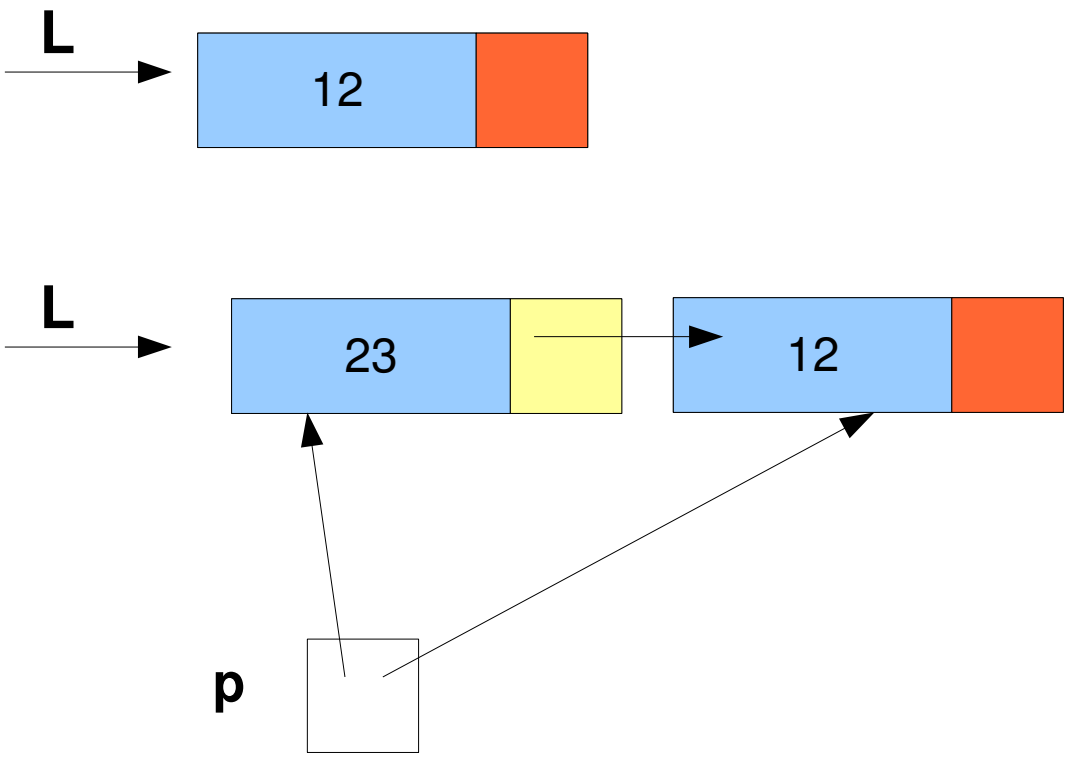
Exemple d'utilisation :

```
int main(void){
telement *L= NULL;
telement *p;

L=cons(L, 12);


L=cons(L, 23);


p=L;
while (p!=NULL) {
printf("%d -> ", p->data);
p=p->suivant;
}
printf("NULL\n");
return 0;
}
```



Ajout d'un élément en fin de liste

/* interface ajouter type : liste*T -> liste
arg l, nvelt (données du nouveau élément)
post ajoute un élement contenant nvelt en fin de la liste l, retourne l'adresse du nouveau premier élément*/
Remarque : si la liste n'est pas vide, le résultat est égal à l

```
liste ajouter (liste l, T nvelt){
*telement p, ptr;
```

On alloue le nouvel élement, soit p son adresse

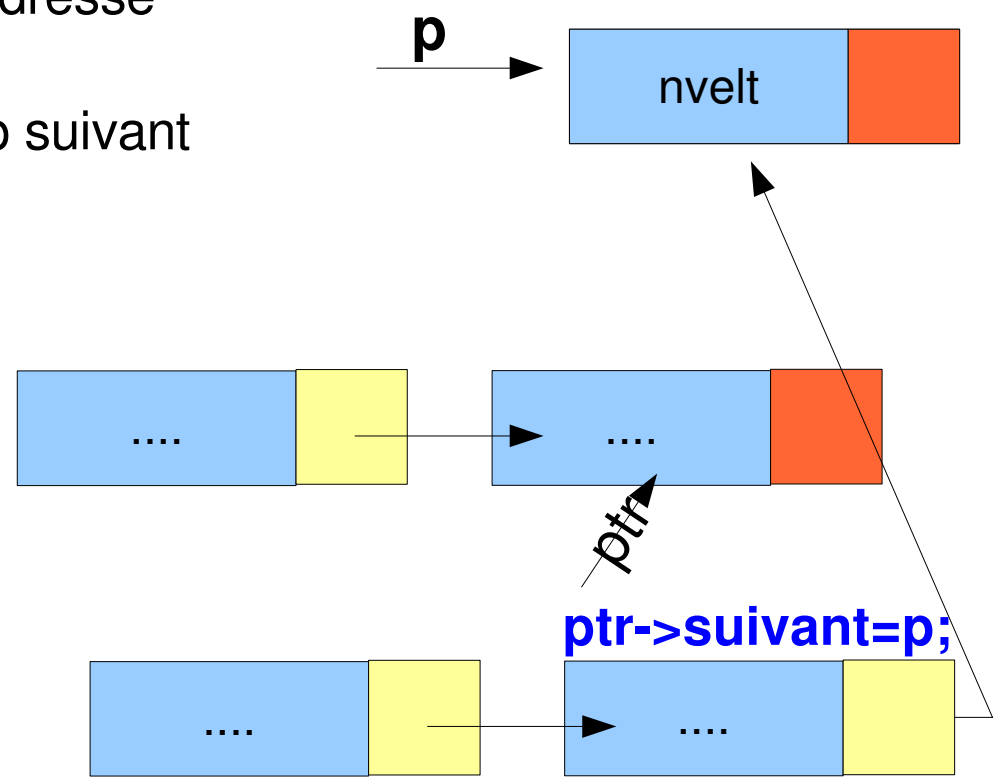
```
p = malloc(sizeof(telement));
```

On remplit son champ data et son champ suivant

```
*p.data = nvelt;   *p.suivant=NULL;
if (l==NULL)  return(p);
else {
```

On relie l'élément alloué au dernier de l

```
      ptr=l;
      while (ptr->suivant!=NULL)
      ptr=ptr.suivant;
      ptr->suivant=p;
       return l;
      }}
```



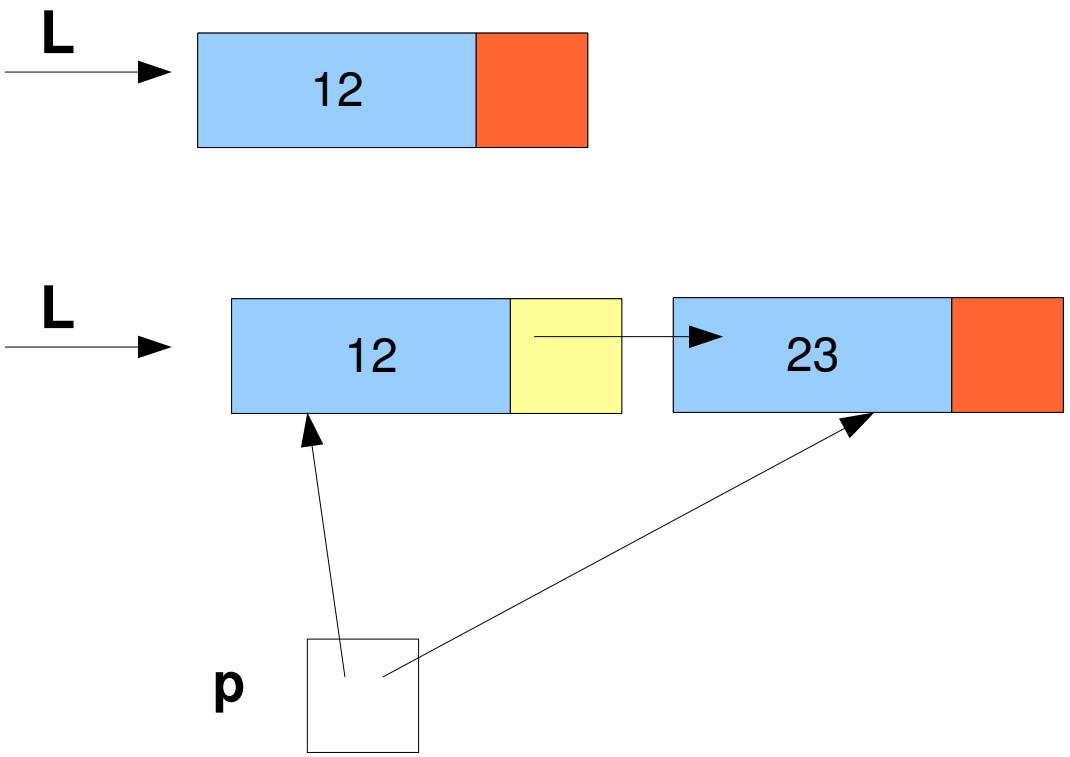
Exemple d'utilisation :

```
int main(void){
telement *L= NULL;
telement *p;

L=ajouter(L, 12);


L=ajouter(L, 23);


p=L;
while (p!=NULL) {
printf("%d -> ", p->data);
p=p->suivant;
}
printf("NULL\n");
return 0;
}
```



Recherche d'un élément

```
/* interface appartient   type : liste*T -> booleen
arg l, v
post true si v est dans la liste, false sinon
*/
```

partir de la tête de la liste
comparer les données de chacun des éléments à celui recherché

```
booleen appartient (liste l, T v){
*telement p=l;
while (p!=NULL){
      if (p->data==)
      return vrai;
      else p->suivant=p;
}
return faux;
}
```

Supprimer un élément

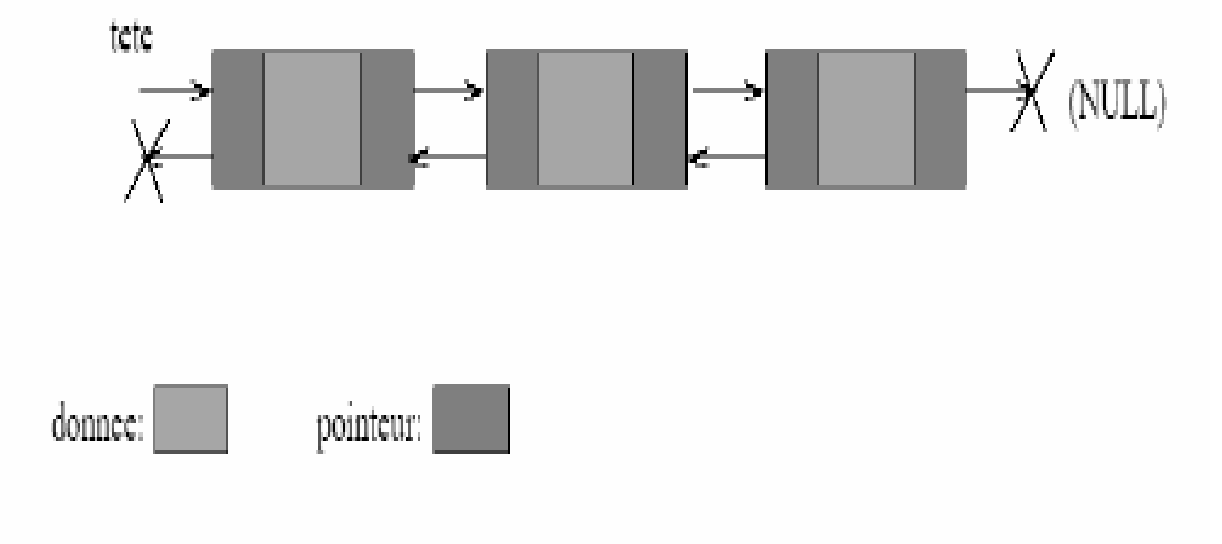
/* interface supprimer
type : liste*T -> liste
arg l, v
post retourne la liste après avoir supprimé la première occurrence de v
*/

Remarque : le résultat est l sauf si le premier élément de la liste représentée par l contient v

A vous de jouer ....

D'autres listes chaînées

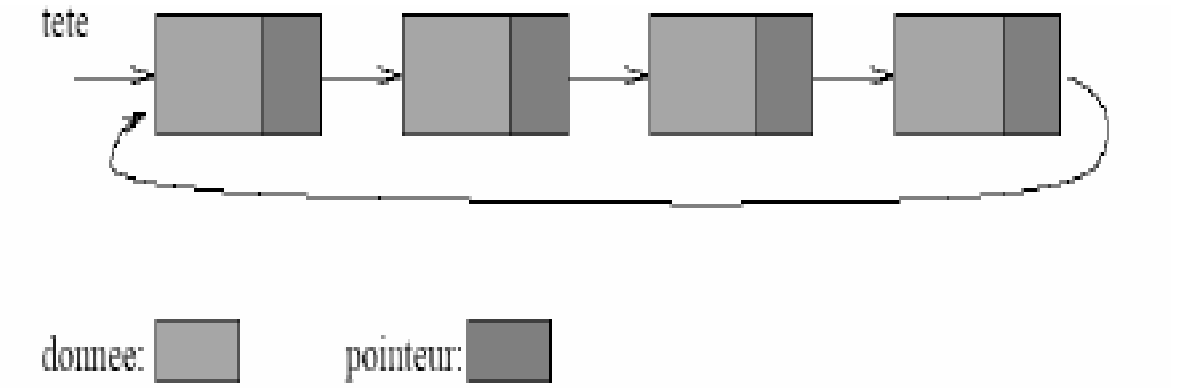
Listes doublement chaînées



Tout élément a un suivant (NULL si dernier) et prédécesseur (NULL si premier)

Les listes doublement chaînées peuvent être parcourues dans les 2 sens

Listes circulaires



Le dernier élément pointe sur le premier : tout élément a un suivant