Chapitre 10

Listes chaînées

10.1 Introduction

Comme nous l'avons vu dans un chapitre précédent, les tableaux permettent de stocker des variables de même type et d'y accéder de façon très simple. Ils ont cependant des inconvénients : ils réservent un grand espace mémoire contingu et ne sont pas adaptés si vous avez un nombre de variables à stocker qui varie avec une amplitude importante au cours de l'exécution du programme.

Pour remédier à ces inconvénients, on utilise des listes chaînées. Leur implémentation est un peu complexe aussi beaucoup de langages proposent une librairie pour les utiliser. Ce chapitre explique comment les listes chaînées fonctionnent.

10.2 Les listes

10.2.1 Présentation et définitions

Les listes sont un objet mathématique récursif dont une définition est la suivante :
Une **liste** est
- soit la liste vide, notée $\emptyset$
- soit un couple formé d'un élément et d'une liste

Exemple 15
$(2, (3, \emptyset))$ *est la liste des entiers 2 et 3.*
3 est un entier, donc $(3, \emptyset)$ *est une liste d'entiers.*
2 est un entier il s'ensuit que $(2, (3, \emptyset))$ *est une liste d'entiers.*

On munit cet objet de quatre fonctions :
- `estVide` qui prend en argument une liste et retourne `vrai` si la liste est vide, `faux` sinon

– `premier` qui prend en argument une liste *non vide* et retourne l'élément du couple que compose cette liste
– `fin` qui prend en argument une liste *non vide* et retourne la liste du couple que compose cette liste
– `ajout` qui prend en argument un élément et une liste et retourne la liste constituée par cet élément et cette liste.

Exemple 16
`premier`$((2,(3,\emptyset))) = 2$
`fin`$((2,(3,\emptyset))) = (3,\emptyset)$
`ajoute`$(4,(2,(3,\emptyset))) = (4,(2,(3,\emptyset)))$

Ces quatre fonctions permettent de créer des listes et de commencer à les utiliser.

10.2.2 Premières utilisations

Etant donnée la définition récursive des listes, il est naturel d'utiliser des fonctions récursives pour les manipuler. Voici quelques exemples.

Pour connaître la taille d'une liste (son nombre d'éléments), on définit la fonction `taille` suivante :

```
int taille (liste l){
if (estVide(l))
  return 0;
else
  return 1 + taille(fin(l))
}
```

Exemple 17
Le calcul de `taille`$((2,(3,\emptyset)))$ *est le suivant :*

$$\begin{aligned}
\texttt{taille}((2,(3,\emptyset))) &= 1 + \texttt{taille}(\texttt{fin}((2,(3,\emptyset)))) \\
&= 1 + \texttt{taille}((3,\emptyset)) \\
&= 1 + 1 + \texttt{taille}(\texttt{fin}((3,\emptyset))) \\
&= 2 + \texttt{taille}(\emptyset) \\
&= 2 + 0 \\
&= 2
\end{aligned}$$

De la même façon on peut définir une fonction qui fait la somme des éléments de la liste :

```
int somme (liste l){
if (estVide(l))
  return 0;
else
  return debut(l) + somme(fin(l))
}
```

10.3 Présentation des listes chaînées

10.3.1 Définition

Les listes chaînées sont une implémentation des listes. Voici leur définition :

```
struct maille{
  int valeur;
  struct maille *suivant;
};

typedef struct maille Maillon;
typedef Maillon * Liste;
```

L'idée est la suivante :
- un `Maillon` est contitué de deux parties, la `valeur` qu'il stocke et l'adresse du `Maillon` suivant.
- une `Liste` pointe sur le premier `Maillon`

Exemple 18
Ainsi la liste de l'exemple 15 va être stocké en mémoire de la façon suivante :

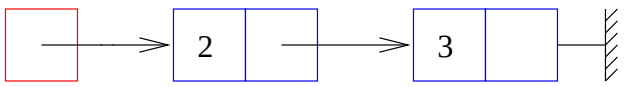


FIG. 10.1 – implémentation de la liste $(2, (3, \emptyset))$.

La liste est représentée par le carré rouge, les maillons par les rectangles bleus. La liste pointe sur le maillon contenant le 2 car c'est le premier élément de la liste. La liste vide est modélisée par `NULL`.

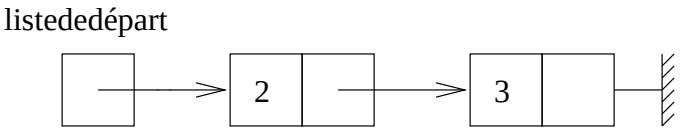
10.3.2 Les quatres fonctions de base

Voici les implémentations des quatre fonctions de base.

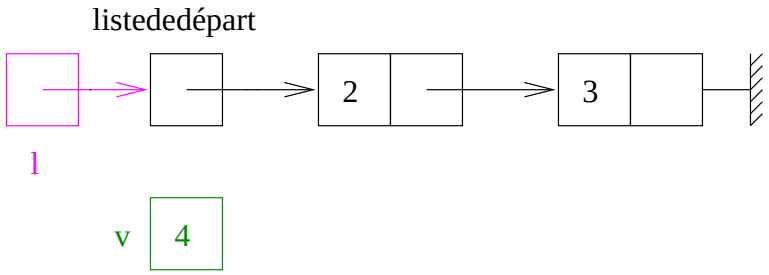
```
int estVide (Liste l){
  return(l==NULL);
}

int premier (Liste l){
  return(l->valeur);
}

Liste fin (Liste l){
  return (l -> suivant);
}

void ajout (Liste * l, int v){
  Maillon * e;
  e = new (Maillon);
  e -> valeur = v;
  e -> suivant = *l;
  *l = e;
}
```

Les trois premières ne posent pas de difficultés, nous allons voir un peu plus en détail la fonction `ajout`.

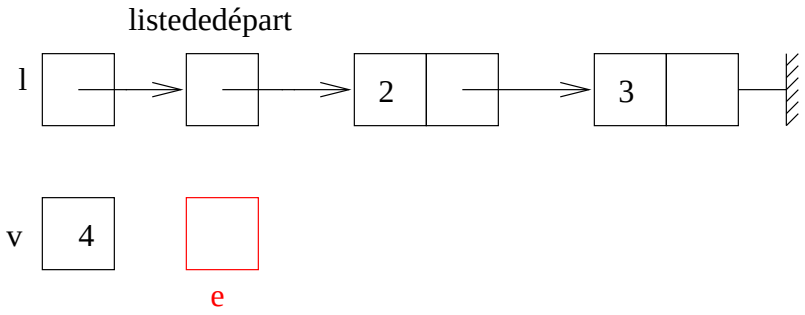
Exemple 19
Je reprends la liste de l'exemple 18 :



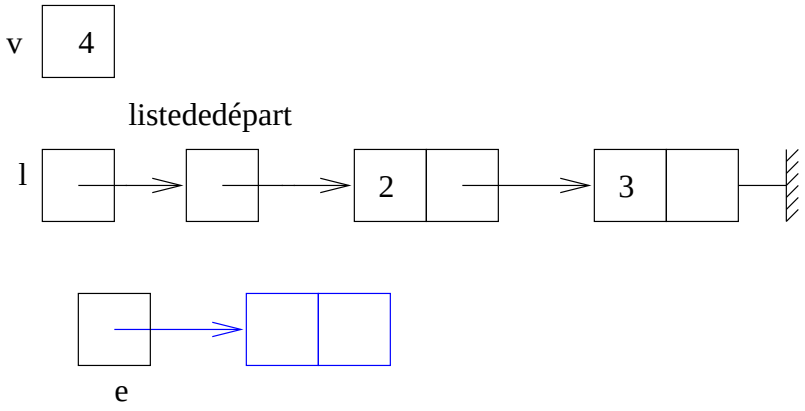
Sur l'appel `ajout(listedépart, 4)` *on va avoir l'évolution de la mémoire suivante :*
On commence par donner à `l` *l'adresse de la liste de départ et à v la valeur 4.*



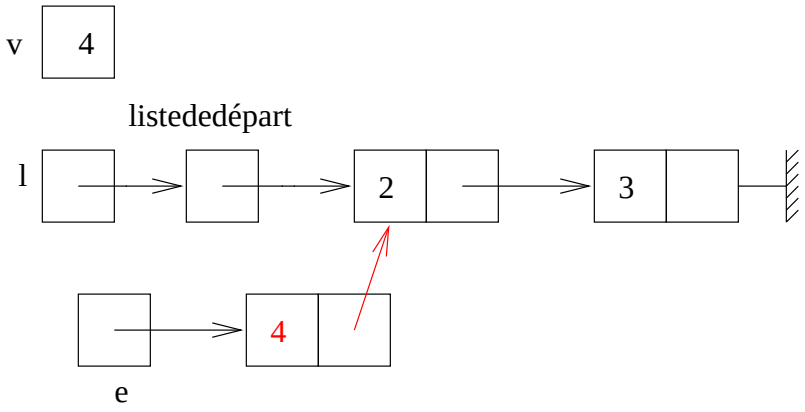
Ensuite on réserve de la mémoire pour le pointeur sur un maillon `e` *(*`Maillon * e ;`*)*



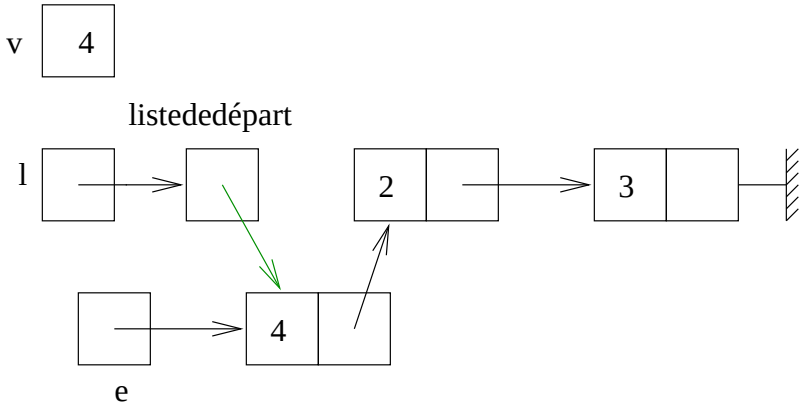
Puis on alloue de la mémoire pour un maillon dont on stocke l'adresse dans `e` *(*`e = new (Maillon) ;`*)*



On donne alors comme valeur a la première case du maillon `v` *et à la seconde la valeur de la case pointée par* `l` *autrement dit la valeur de* `listededépart`*. (*`e -> valeur = v ; e -> suivant = *l ;`*)*

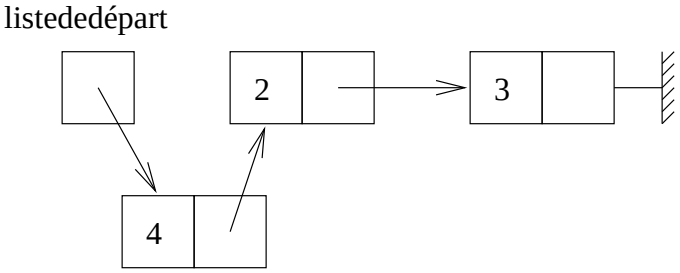


Enfin on donne a la case pointée par `l` *autrement dit* `listededépart` *la valeur de* `e`. *(*`*l = e ;`* *)*



Le lien entre `listededépart` *et le maillon contennat 2 a donc disparu.*

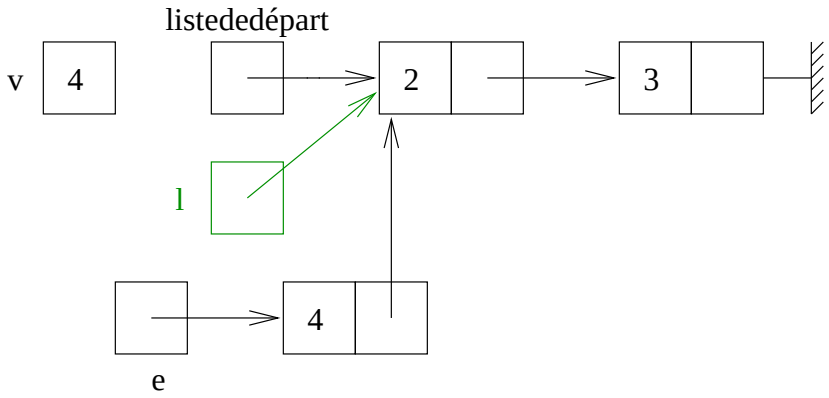
Quand on quitte la fonction, on libére la mémoire occupée par `l`, `v` *et* `e`. *Il ne reste plus que :*



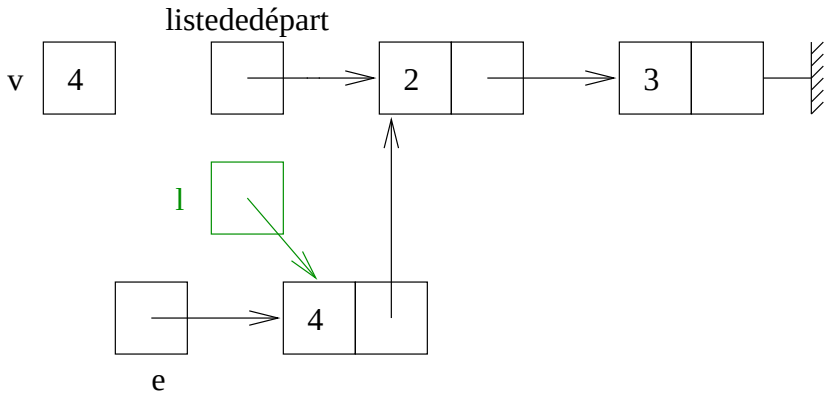
On constate que 4 a bien été ajouté à la liste de départ.

Quelques remarques au sujet de la fonction `ajout` :

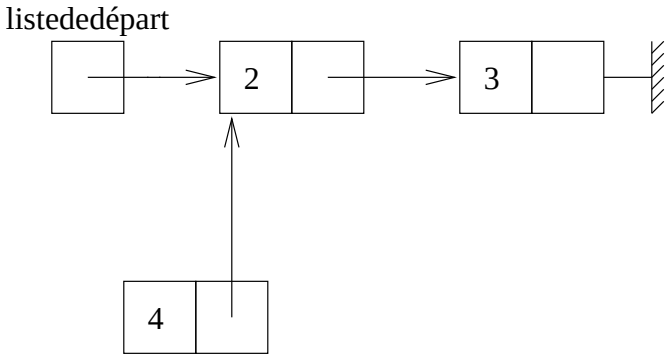
– il est nécessaire de passer en argument de la fonction `ajout` l'adresse de la liste de départ sinon on est dans la situation suivante :



On ne peut alors lier le nouveau maillon qu'à `l`



Quand on sort de la fonction le rajout n'a pas été fait.



– Il est nécessaire d'allouer de la mémoire pour le nouveau maillon qui ne soit pas libéré quand on sort de la fonction d'où le passage par un pointeur sur un maillon et l'usage de `new`.

On peut alors facilement écrire une fonction pour afficher une liste :

```
void afficherec(Liste l){
  if (estVide(l))
    cout << endl;
  else{
    cout << premier(l) << " ";
    afficherec(fin(l));
  }
}
```

Testez le programme suivant :

Programme 67

```
#include <iostream>
using namespace std;

struct maille{
  int valeur;
  struct maille *suivant;
};

typedef struct maille Maillon;
typedef Maillon * Liste;

void ajout (Liste * l, int v){
  Maillon * e;
  e = new (Maillon);
  e -> valeur = v;
  e -> suivant = *l;
  *l = e;
}

int estVide (Liste l){
  return(l==NULL);
}

int premier (Liste l){
  return(l->valeur);
}
```

```
Liste fin (Liste l){
  return (l -> suivant);
}

void afficherec(Liste l){
  if (estVide(l))
    cout << endl;
  else{
    cout << premier(l) << " ";
    afficherec(fin(l));
  }
}

main(){
  Liste l;
  l=NULL;
  ajout(&l,2);
  ajout(&l,5);
  ajout(&l,5);
  ajout(&l,3);
  afficherec(l);
}
```

Il crée une liste contenant les valeurs 2, 5, 5 et 3.

Les valeurs sont ajoutées en tête de la liste. La valeur immédiatement accessible est la dernière ajoutée à la liste.

10.3.3 Fonctions récursives contre fonctions itératives

Le plus naturel lorsque l'on travaille avec des fonctions est d'écrire des fonctions récursives.

Exemple 20
Voici une fonction qui indique si la valeur `x` *est présente dans la liste* `l` *:*

```
int presentrec (Liste l, int x){
  if (estVide(l))
    return(0);
  else
    if (x == l->valeur)
      return(1);
    else
```

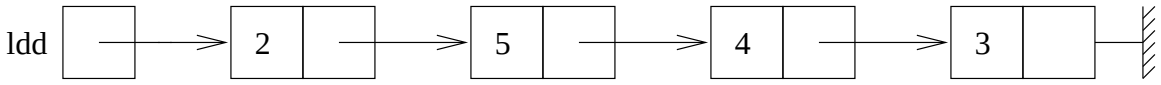
```
    return(presentrec(fin(l),x));
}
```

Cependant il est également possible de les écrire itératives mais on utilise alors des pointeurs.

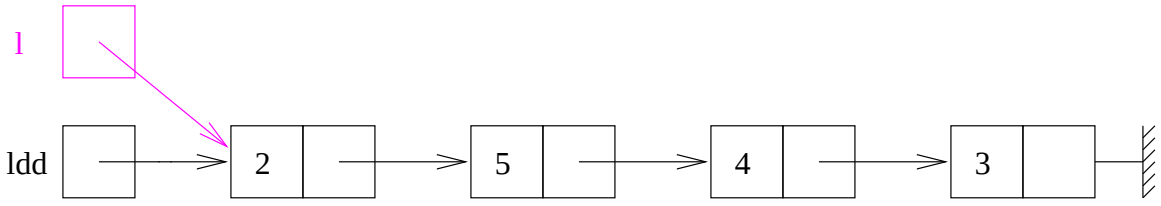
Exemple 21
La fonction suivante indique si la valeur `x` *est présente dans la liste* `l` *comme celle de l'exemple 20 sans utiliser d'appel récursif :*

```
int presentit (Liste l, int x){
  int b;
  b = 0;
  while (! (estVide(l) | b)){
    b =  (x == l -> valeur);
    l = l -> suivant;
  }
  return(b);
}
```

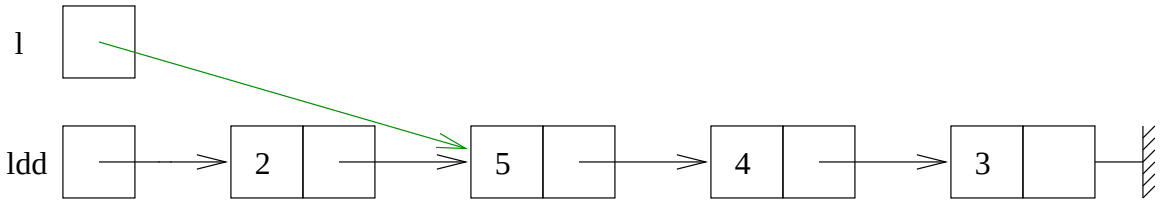
Nous allons voir l'évolution de la mémoire lors de l'appel de `presentit` *sur la liste* `ldd` *si dessous et la valeur 4.*



Pour ne pas surcharger la figure je ne représente que `ldd` *et* `l`.
A l'appel de la fonction, on est dans la situation suivante :

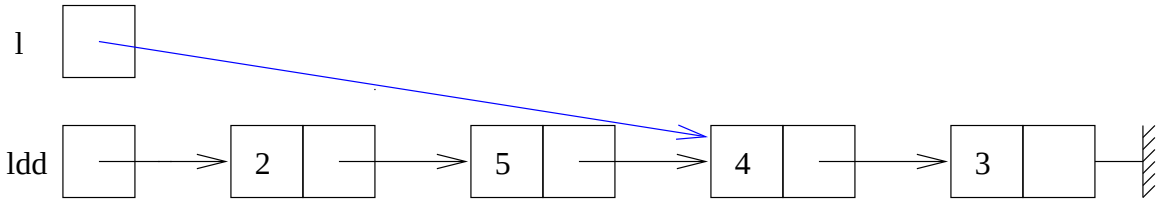


Le premier test du `while` *est vrai, donc on rentre dans la boucle.* `b` *reprend la valeur* `faux` *car* `l -> valeur` *vaut 2.* `l = l -> suivant` *amène la modification suivante :*



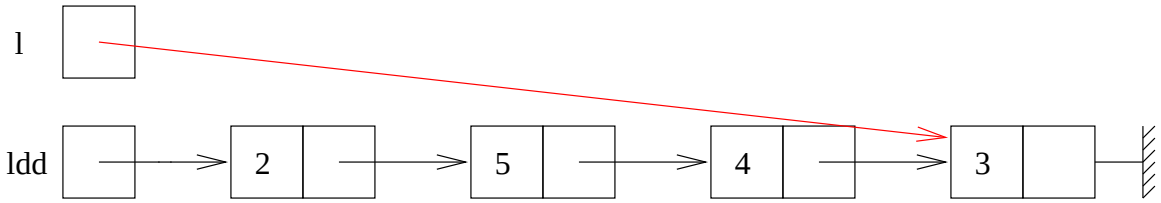
`l` *pointe désormais sur le second maillon de la liste.*

Le second test du `while` *est vrai, donc on rentre dans la boucle.* `b` *reprend la valeur* `faux` *car* `l -> valeur` *vaut 5.* `l = l -> suivant` *amène la modification suivante :*



`l` *pointe désormais sur le troisième maillon de la liste.*

Le second test du `while` *est vrai, donc on rentre dans la boucle.* `b` *prend la valeur* `vrai` *car* `l -> valeur` *vaut 4.* `l = l -> suivant` *amène la modification suivante :*



`l` *pointe désormais sur le quatrième maillon de la liste.*

On sort alors de la boucle et on retourne `vrai`

Exemple 22

voici une fonction qui permet de sommer itérativement les éléments d'une liste :

```
int sommeit (Liste l){
  int s;
  s = 0;
  while (! estVide(l)){
```

```
    s += l -> valeur;
    l = l -> suivant;
  }
  return(s);
}
```

Tant que l'on ne modifie pas la structure de la liste, on peut utiliser simplement les fonctions récursives. Par contre quand on veut modifier la structure de la liste, il faut passer par les pointeurs, c'est notamment le cas quand on veut supprimer un élément.

10.3.4 Suppression d'un élément

C'est certainement le point le plus délicat des listes chaînées : supprimer un élément. Voici deux fonctions :

- la première supprime une seule fois `x`, celui qui est le plus tôt dans la liste

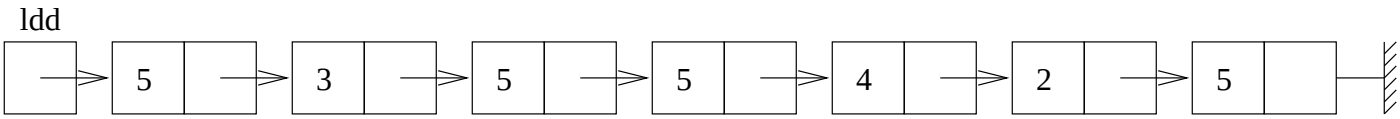
```
void supprimerun (Liste *l, int x){
  Liste aux;
  aux = *l;
  while(! estVide(aux) ){
    if (aux -> valeur == x){
      *l = aux -> suivant;
      delete aux;
      aux = NULL;
    }
    else{
      l = &(aux -> suivant);
      aux = aux -> suivant;
    }
  }
}
```

- la première supprime tous les `x` présents dans la liste

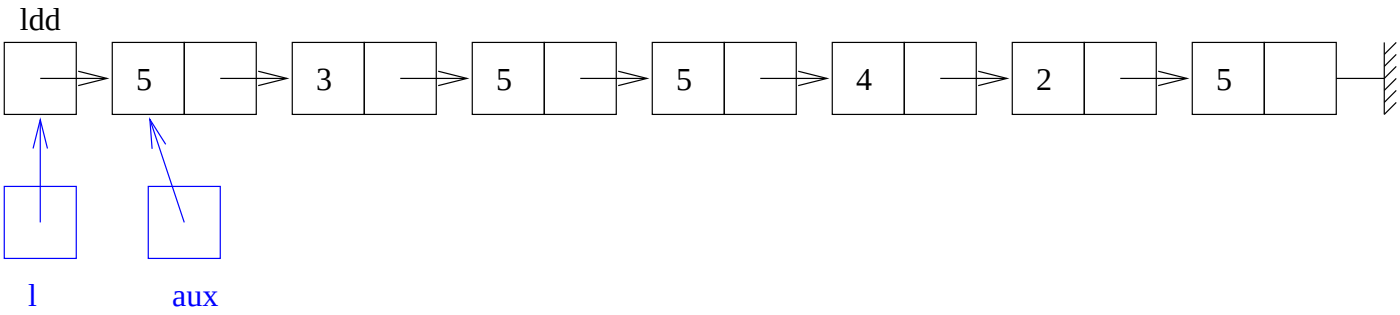
```
void supprimertous (Liste *l, int x){
  Liste aux;
  aux = *l;
  while(! estVide(aux) ){
    if (aux -> valeur == x){
      *l = aux -> suivant;
      delete aux;
      aux = *l;
    }
    else{
      l = &(aux -> suivant);
```

```
            aux = aux -> suivant;
          }
        }
      }
```

Exemple 23
Nous allons voir l'évolution de la mémoire pour la liste `ldd` *suivante :*

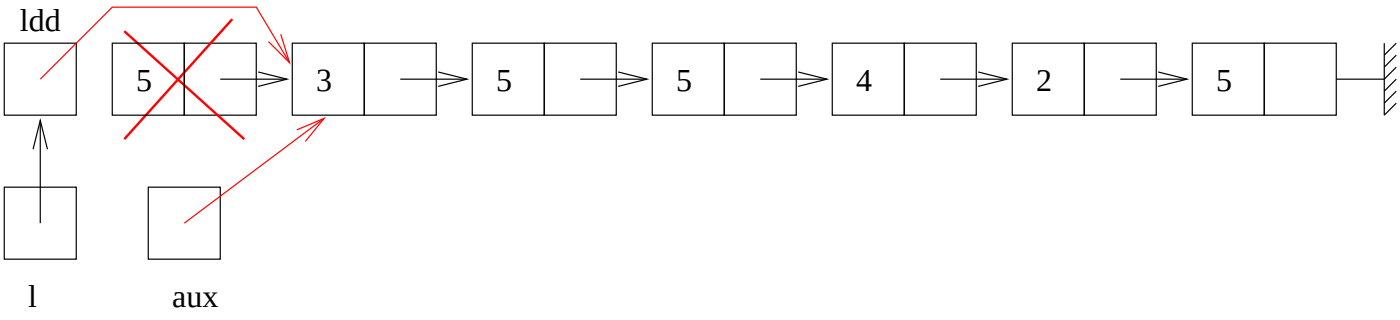


lors de l'appel de `supprimertous` *pour la valeur 5.*
Apres l'appel de la fonction et l'affectation de `aux`*, on est dans la situation suivante :*

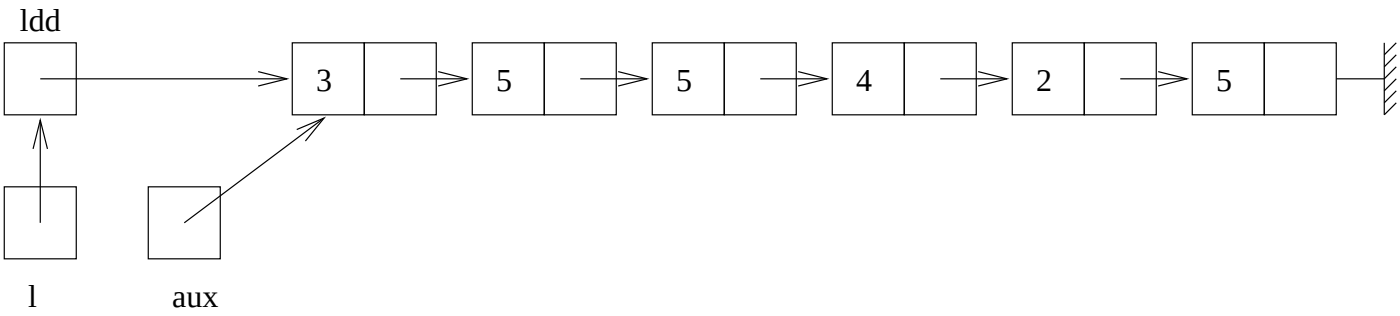


La valeur du test du while est `vrai` *ainsi que celle du* `if` *donc la case surlaquelle pointe* `l` *autrement dit* `ldd` *prend la valeur de* `aux -> suivant` *autrement dit pointe sur le deuxième maillon.*

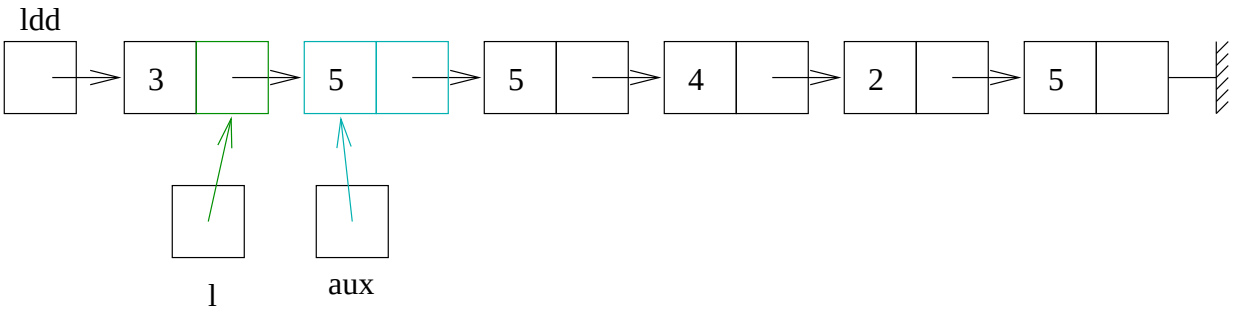
On libére la mémoire contenant le premier maillon et `aux` *pointe désormais sur le second maillon.*



On a donc



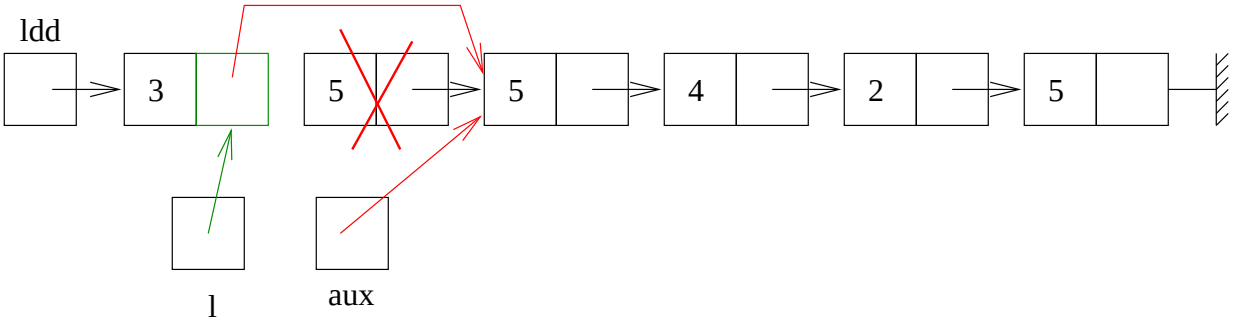
La valeur du second test du while est toujours `vrai` *mais celle du* `if` *est* `faux` *donc* `l` *pointe désormais sur la deuxième case du maillon contenant 3 et* `aux` *pointe sur le maillon suivant.*



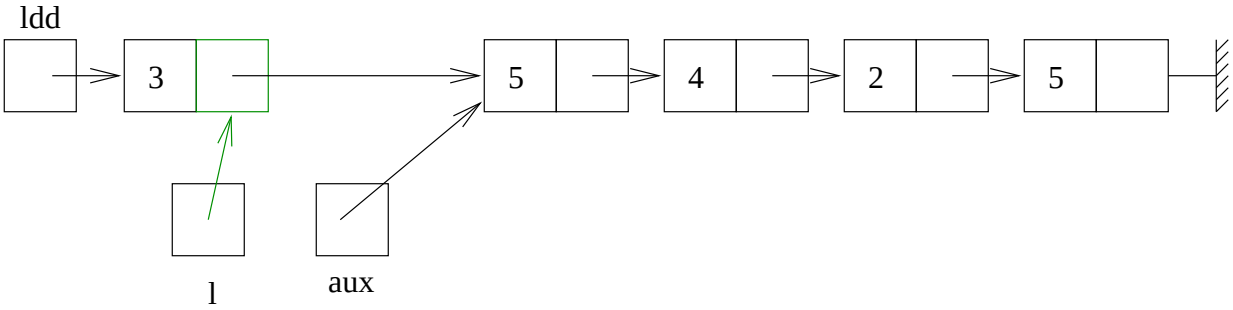
On a donc `l` *qui pointe sur une case alors que* `aux` *pointe sur tout un maillon.*

La valeur du troisième test du while est `vrai` *ainsi que celle du* `if` *donc la case sur laquelle pointe* `l` *prend la valeur de* `aux -> suivant` *autrement dit pointe sur le troisième maillon.*

On libére la mémoire contenant le deuxième maillon et `aux` *pointe désormais sur le troisième maillon.*

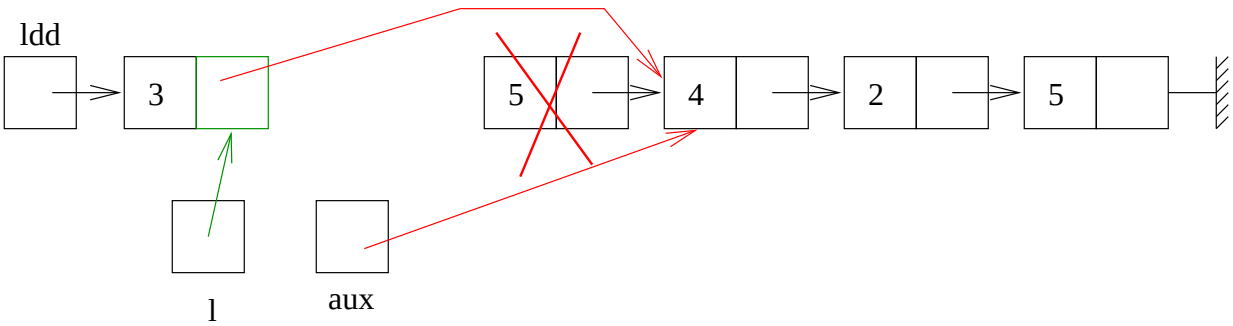


On a donc

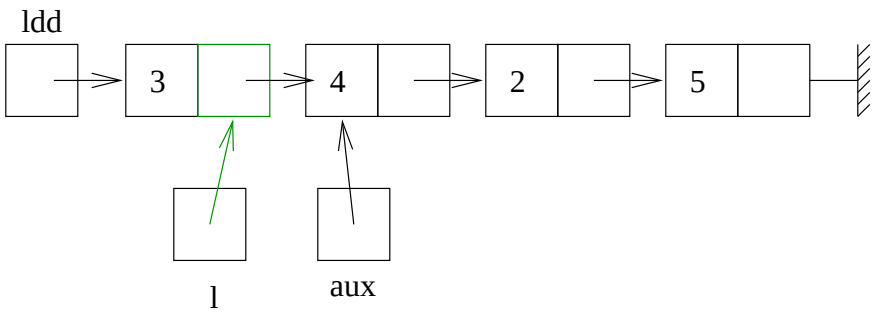


La valeur du quatrième test du while est `vrai` *ainsi que celle du* `if` *donc la case sur laquelle pointe* `l` *prend la valeur de* `aux -> suivant` *autrement dit pointe sur le troisième maillon.*

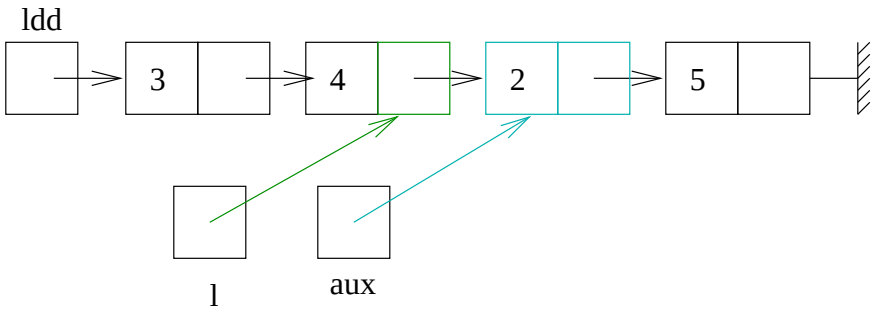
On libére la mémoire contenant le deuxième maillon et `aux` *pointe désormais sur le troisième maillon.*



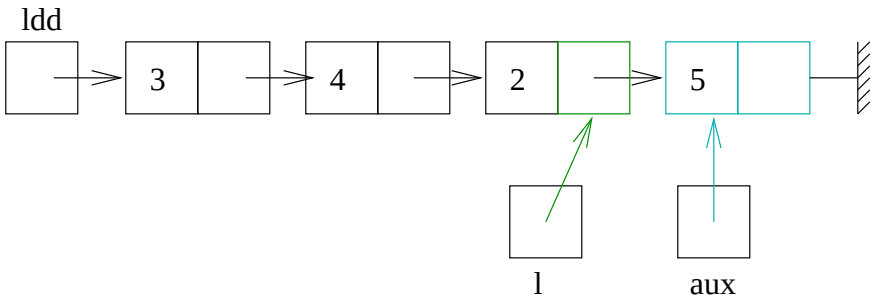
On a donc



La valeur du cinquième test du while est toujours `vrai` *mais celle du* `if` *est* `faux` *donc* `l` *pointe désormais sur la deuxième case du maillon contenant 4 et* `aux` *pointe sur le maillon suivant.*



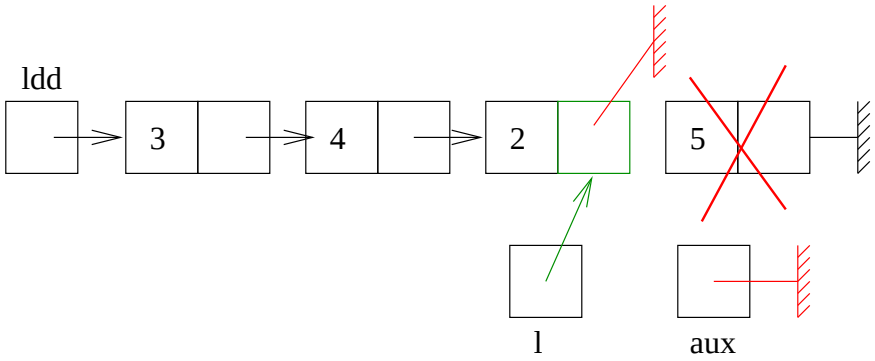
La valeur du sixième test du while est toujours `vrai` *mais celle du* `if` *est* `faux` *donc* `l` *pointe désormais sur la deuxième case du maillon contenant 2 et* `aux` *pointe sur le maillon suivant.*



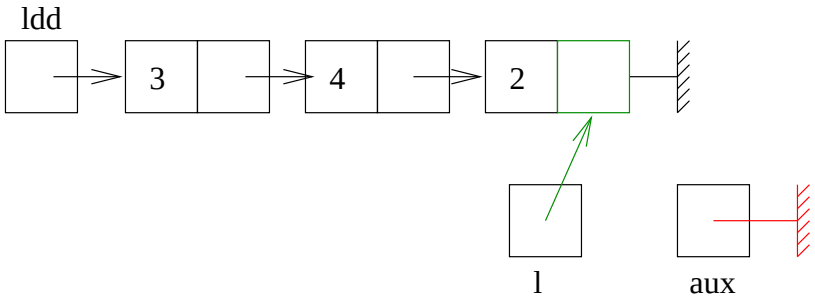
La valeur du septième test du while est `vrai` *ainsi que celle du* `if` *donc la case*

sur laquelle pointe `l` *prend la valeur de* `aux -> suivant` *autrement dit pointe sur NULL.*

On libére la mémoire contenant le quatrième maillon et `aux` *pointe désormais sur NULL.*



On a donc



On remarque que dans les fonctions de suppression, `l` permet de stocker la case qui pointe sur le maillon à supprimer. Il est en effet **impossible de la retrouver à partir du maillon**. `aux` permet de parcourir la liste.