# Programmation – TD1 : Listes et graphes en C

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

29/30 septembre 2004

## 1 Listes (simplement) chaînées

### 1.1 Représentation

On représente une liste chaînée d'entiers par un ensemble de cellules. Chaque cellule est constituée de deux informations : l'entier qu'elle contient, et un pointeur qui permet de trouver la prochaine cellule de la chaîne. Ce pointeur est mis à `NULL` si la cellule est la dernière de la chaîne.

On définit donc le type C suivant :

```
struct cell {
  int value;
  struct cell *next;
};
```

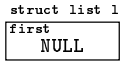
Pour donner une liste, il suffit donc d'indiquer où se trouve la première cellule :

```
struct list {
  struct cell *first;
};
```

Donnons ici deux exemples :

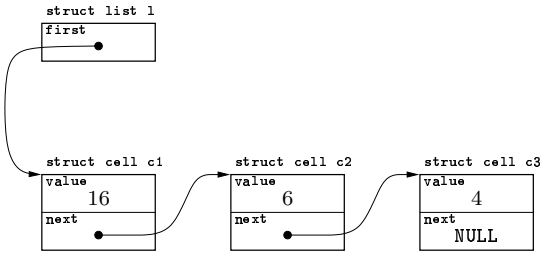
– la liste vide ne contient aucune cellule, il suffit donc de dire que `first` ne pointe sur rien :

```
struct list l;
l.first = NULL;
```




– la liste $[16, 6, 4]$ peut par exemple être obtenue comme suit :

```
struct cell c1, c2, c3;
struct list l;

c1.value = 16; c1.next = &c2;
c2.value = 6;  c2.next = &c3;
c3.value = 4;  c3.next = NULL;
l.first = &c1;
```




### 1.2 *Exemple* : la fonction `push`

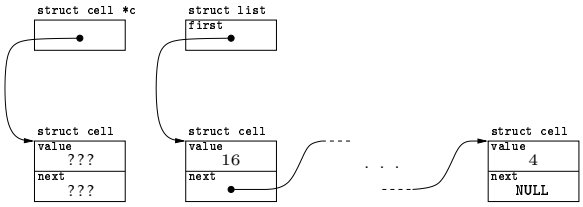
Pour illustrer l'utilisation des listes chaînées, voici l'exemple de la fonction `push`, qui insère un élément en tête de la liste :

```
void push(struct list *l, int value)
{
  struct cell *c;
  c = malloc(sizeof(struct cell));

  c->value = value;
  c->next = l->first;

  l->first = c;
}
```

Cette fonction opère de la manière suivante :

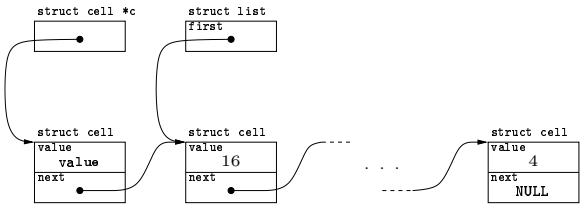
1. `struct cell *c;`
   `c = malloc(sizeof(struct cell));`

   On alloue un bloc mémoire de la taille d'une cellule, et on met l'adresse de ce bloc dans `c`.



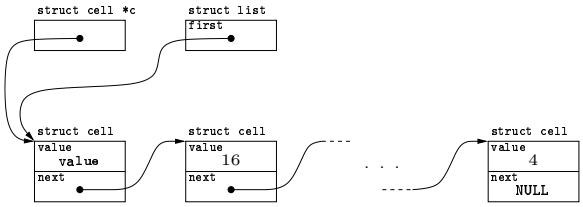

2. `c->value = value;`
   `c->next = l->first;`

   On initialise les champs de la structure pointée par `c` :
   `c->value` prend la valeur `value` ;
   `c->next` pointe vers le premier élément de la liste.




3. `l->first = c;`

   On fait pointer `l->first` sur la nouvelle cellule (qui devient alors tête de la liste).




Vous avez tout compris ? Sinon appelez le TDman à la rescousse !

### 1.3 Utilisation de `ddd`

`ddd` (pour *Data Display Debugger*) est une interface graphique venant se placer au dessus de `gdb`, et permet de visualiser très simplement des structures de données telles que celles manipulées dans cet exercice.

Tout d'abord, allez sur la page `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/` et téléchargez-y le fichier `list.c`. Ce fichier contient les définitions de `struct cell`, `struct list` et de la fonction `push`, ainsi qu'un `main` rudimentaire qui crée une liste vide `l` et y insère les éléments $4$, $6$ et $16$ pour obtenir la liste $[16, 6, 4]$.

Compilez ce programme avec l'option `-g` (option de debug) :

```
gcc list.c -Wall -g -o list
```

Lancez ensuite `ddd list`. Une fenêtre va s'ouvrir, séparée en deux horizontalement : dans la moitié supérieure se trouve le code du programme, et le bas est une interface en ligne de commande avec `gdb`.

Cliquez sur `main`, puis sur le bouton Break dans la barre d'outils. Cela va insérer automatiquement un point d'arrêt (*breakpoint*) au début de la fonction `main`. Ensuite, cliquez sur Run : le programme s'exécute alors jusqu'au breakpoint. Une flèche verte indique à quel endroit du code vous vous trouvez.

Normalement, cette flèche verte doit pointer sur la déclaration de la variable `l`. Faites donc Next pour exécuter cette instruction.

La structure `l` vient donc d'être créée. Pour regarder ce qu'elle contient, cliquez dessus, puis cliquez sur le bouton Display dans la barre d'outils. Une représentation graphique de `l` apparaît alors. On peut voir que son champ `first` contient la valeur `0x0`, ce qui représente le pointeur `NULL`.

En cliquant sur le champ `first` puis sur Display, vous pouvez *déréférencer* le pointeur, c'est-à-dire afficher ce vers quoi il pointe. Ici, `ddd` affiche (Disabled) car le pointeur `NULL` ne pointe vers rien. Cliquez alors sur cette boîte et fermez la en cliquant sur Undisplay.
*Attention* : il est important de toujours bien fermer les boîtes (Disabled) car `ddd` ne gère pas très bien les mises à jour du graphe. Il peut parfois être même préférable de recommencer complètement l'affichage de la structure.

Ensuite, faites Next pour exécuter le premier appel à `push`. `ddd` surligne alors le champ `first` de `l` pour vous signaler que sa valeur vient de changer. Affichez alors son contenu comme précédemment (clic sur `first`, puis Display). `l.first` pointe donc bien sur une structure de type `struct cell` ayant son champ `value` à 4 et son champ `next` à `NULL`.

Continuez à exécuter le programme pas à pas, et observez bien les changements successifs de la liste `l`.

### 1.4 Fonctions de base

1. Écrivez une fonction `void print(struct list *l)` qui affiche les entiers d'une liste, par exemple sous la forme `[16, 6, 4]`.
2. Écrivez une fonction `int pop(struct list *l)` qui supprime la première cellule d'une liste, tout en renvoyant sa valeur. Si jamais la liste est vide avant l'appel à `pop`, la fonction affiche un message d'erreur, puis quitte par un appel à `abort()`.
   *Indication* : votre fonction doit donc :
   – copier le pointeur `l->first` dans une variable locale (`c` par exemple) ;
   – copier la valeur de `c->value` dans une variable locale ;
   – faire pointer `l->first` vers `c->next` (cette cellule devient alors la tête de la liste) ;
   – libérer la cellule pointée par `c` avec l'instruction `free(c);` ;
   – retourner la copie de `c->value` ;

Un élève a écrit le code suivant :

```
int x;
struct list m;
struct list l;

/* ... */
```

```
/* PLEIN DE CODE INCOMPRÉHENSIBLE */
/* ...  */

m.first = NULL;
while (l.first != NULL) {
  x = pop(&l);
  push(&m, x);
}
```

**3.** Que fait ce bout de code ?

Dans la vie, créer et détruire des tas de cellules n'est pas forcément une bonne chose, surtout quand on peut faire autrement.[1] L'exemple précédent libère toutes les cellules de la liste pour ensuite les recréer, ce qu'on veut éviter. Vous n'avez donc pas le droit dans la question suivante d'utiliser une fonction qui crée ou libère de la mémoire (`malloc`, `free`...) ni d'utiliser des fonction qui appellent celles-ci (`push`, `pop`...).

**4.** Écrivez une fonction `void reverse(struct list *l)` tel qu'après l'appel à cette fonction, la liste ait changée de sens. Ainsi la liste `[16,6,4]` devient `[4,6,16]`.

### 1.5 Un tri rapide

Là encore, essayez d'éviter d'utiliser des fonctions gérant la mémoire.

**5.** Faites `void split(int x, struct list *l, struct list *l1, struct list *l2)` telle qu'après l'appel à `split(x, l, l1, l2)`, la liste `l1` contienne tous les éléments inférieurs ou égaux à `x`, et la liste `l2` tous les éléments qui lui sont supérieurs. Après l'appel de cette fonction, la liste `l` est donc devenue la liste vide.
*Remarque* : on ne demande pas que les éléments de `l1` et de `l2` soient dans le même ordre que celui dans lequel ils étaient dans `l`. Par exemple une fonction `split` qui sur l'entrée `x = 5` et `l = [2,7,4,5,3,8,9]` renvoie `l1 = [4,2,3,5]` et `l2 = [7,9,8]` est valide.

**6.** Faites la fonction `void append(struct list *l1, struct list *l2, struct list *l)` telle qu'après l'appel à cette fonction, la liste `l` soit la concaténation des listes `l1` et `l2`. On ne précise pas ce que deviennent les deux autres listes.

Aidés de ces deux fonctions, vous devriez être capables d'écrire un tri rapide. On rappelle le fonctionnement de ce tri :

- choisir et retirer un élément $x$ de la liste $l$ au hasard (par exemple le premier) ;
- couper la liste en deux listes contenant respectivement les éléments plus petits et plus grands que $x$ ;
- trier récursivement les deux listes, puis recoller sans oublier de rajouter $x$.

**7.** Écrivez une fonction `void quick_sort(struct list *l)` qui trie la liste `l`.

## 2 Listes doublement chaînées

### 2.1 Représentation

Cette fois-ci, chaque cellule contient deux pointeurs : un pointeur vers la cellule qui la précède dans la liste, et un pointeur vers celle qui la suit. Une liste est alors la donnée de deux

[1] On parle d'informatique, pas de biologie

cellules : la première et la dernière.

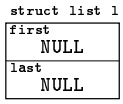
On en vient donc à définir les types suivants :

```
struct cell{
  int value;
  struct cell *prev, *next;
};

struct list {
  struct cell *first, *last;
};
```

Reprenons les deux exemples précédents.

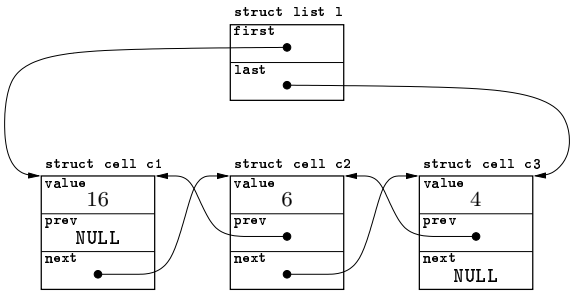
– la liste vide ne contient aucune cellule, il suffit donc de dire que `first` et `last` ne pointent sur rien :

```
struct list l;
l.first = NULL; l.last = NULL;
```




– enfin, la liste $[16, 6, 4]$ peut par exemple être obtenue comme suit :

```
struct cell c1, c2, c3;
struct list l;

c1.value = 16; c1.prev = NULL; c1.next = &c2;
c2.value = 6;  c2.prev = &c1;  c2.next = &c3;
c3.value = 4;  c3.prev = &c2;  c3.next = NULL;
l.first = &c1; l.last = &c3;
```




## 2.2 *Exemple* : la fonction `push_front`

De même que pour les listes simplement chaînées, voici la fonction `push_front`, qui insère un élément en tête de la liste :

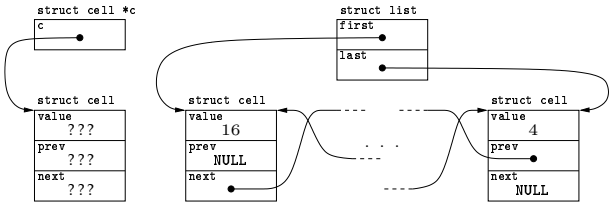
```
void push_front(struct list *l, int value)
{
  struct cell *c;

  c = malloc(sizeof(struct cell));
  c->value = value;
  c->next = l->first;
  c->prev = NULL;
```

```
  if (l->first != NULL)
    l->first->prev = c;

  l->first = c;
  if (l->last == NULL)
    l->last = c;
}
```

Cette fonction opère de la manière suivante :

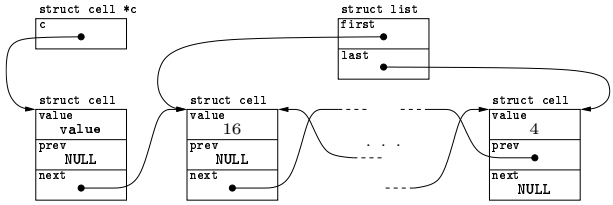
1. `c = malloc(sizeof(struct cell));`

   On alloue un bloc mémoire de la taille d'une cellule, et on met l'adresse de ce bloc dans `c`.



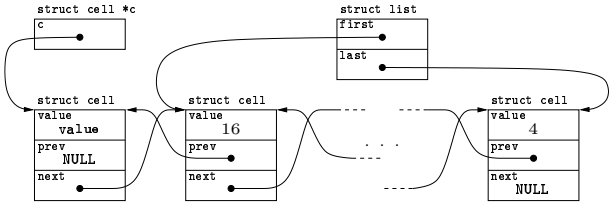

2. `c->value = value;`
   `c->next = l->first;`
   `c->prev = NULL;`

   On initialise les champs de la structure pointée par `c` :
   `c->value` prend la valeur `value` ;
   `c->next` pointe vers le premier élément de la liste ;
   `c->prev` prend la valeur `NULL` car il n'y a aucun élément précédant `c` ;



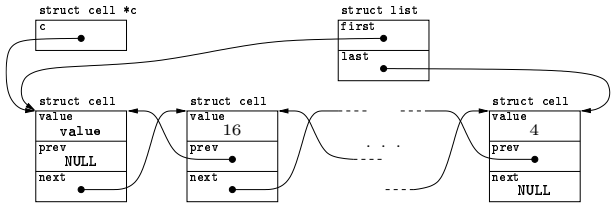

3. `if (l->first != NULL)`
   `  l->first->prev = c;`

   Si la liste n'est pas vide, on fait pointer le champ `prev` du premier élément de la liste vers la nouvelle cellule.




4. `l->first = c;`
   `if (l->last == NULL)`
   `  l->last = c;`

   On fait pointer `l->first` sur la nouvelle cellule (qui devient alors tête de la liste). Si la liste était vide, il faut aussi faire pointer `l->last` sur cette cellule.




## 2.3 Fonctions de base

1. Écrivez une fonction `void print(struct list *l)` qui affiche les entiers d'une liste.
2. Sur le modèle de `push_front`, faites `void push_back(struct list *l, int value)`, qui ajoute une cellule à la fin de la liste.
3. Écrivez `int pop_front(struct list *l)` et `int pop_back(struct list *l)`, qui effacent respectivement la première et la dernière cellule, tout en renvoyant leur valeur (attention en cas de liste vide).
4. Écrivez une fonction `void reverse(struct list *l)` qui change le sens d'une liste.

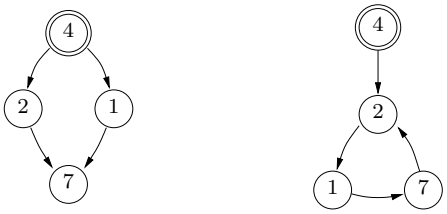
**5.** Écrivez `void append(struct list *l1, struct list *l2, struct list *l)` qui concatène deux listes. Vérifiez qu'elle est bien plus simple que pour le cas des listes simplement chaînées.

# 3 *Bonus* : Graphes binaires acycliques orientés (DABG)

## 3.1 Représentation

Un tel graphe est constitué d'un ensemble de nœuds, chacun ayant (au plus) deux successeurs. Un graphe est acyclique s'il n'existe aucun cycle orienté (il n'existe aucun sommet à partir duquel, en choisissant des successeurs, on puisse revenir sur celui-ci). On y distingue un sommet privilégié, appelé racine, tel qu'on puisse atteindre tout sommet en partant de celui-ci.

Voilà un exemple de graphe acyclique et un contre-exemple :

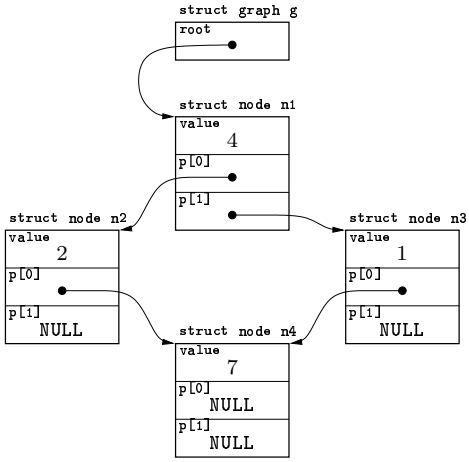



En C, un tel graphe est représenté naturellement de la façon suivante :

```
struct node {
  int value;
  struct node *p[2];
};

struct graph {
  struct node *root;
};
```

Ainsi, le premier graphe ci-haut est représenté de la façon suivante :

```
struct graph g;
struct node n1, n2, n3, n4;

n1.value = 4; n1.p[0] = &n2;  n1.p[1] = &n3;
n2.value = 2; n2.p[0] = &n4;  n2.p[1] = NULL;
n3.value = 1; n3.p[0] = &n4;  n3.p[1] = NULL;
n4.value = 7; n4.p[0] = NULL; n4.p[1] = NULL;
g.root = &n1;
```




## 3.2 Questions

**1.** Quel est le rapport avec les questions précédentes ?

2. Écrivez une fonction d'affichage qui se contente d'afficher chaque nœud.
   *Attention* : chaque nœud ne devra être affiché qu'une seule fois. Si vous vous y prenez mal, vous risquez, pour l'exemple précédent, d'afficher la valeur 7 deux fois. On vous propose deux méthodes pour y remédier :
   – utiliser une structure auxiliaire de votre choix ;
   – enrichir un peu la structure existante. Par exemple, vous pouvez ajouter un champ à chaque nœud qui vous permet de savoir si vous l'avez déjà affiché ou non :

```
struct node {
  int value;
  struct node *p[2];
  int visited;
};
```

3. *Difficile* : écrivez une fonction qui libère toute la mémoire allouée par un DABG. Là encore, une structure de données auxiliaire peut s'avérer nécessaire.