1 Manipulation de listes chaînées en C

Une liste chaînée permet de gérer un collection ordonnée de données de même type, de manière plus souple mais également plus complexe qu'avec un simple tableau. Le nouveau type "*liste d'entiers*" se définit de la manière suivante :

```
struct cellule
{
  int val;
  struct cellule *suiv;
};

typedef struct cellule *Liste;
```

La difficulté majeure, lorsque l'on manipule des listes chaînées, consiste à maintenir un chaînage cohérent, c'est-à-dire :

- de toujours posséder un pointeur vers la première cellule de la liste,
- de maintenir un bon chaînage, i.e. que la valeur du pointeur `suiv` soit bien l'adresse de la cellule suivante,
- de s'assurer que le dernier pointeur de la liste (la valeur du champ `suiv` de la dernière cellule) vaut bien la valeur spéciale `NULL`, indiquant la fin de la liste.

Une seconde difficulté est de bien gérer la mémoire. L'allocation est faite au moyen de l'instruction `malloc` qui, malgré une syntaxe difficile, s'utilise simplement de la manière suivante : afin de réserver un bloc de mémoire de 17 octets, il suffit d'utiliser l'instruction `malloc(17)`. Cette instruction retourne l'adresse du bloc réservé, c'est-à-dire un pointeur vers cette zone de mémoire.

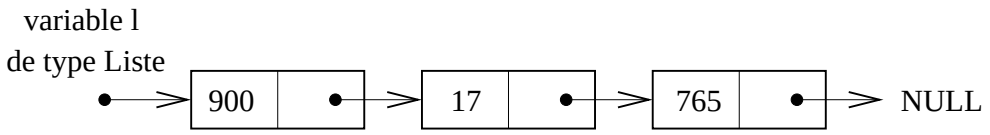

FIGURE 1 – Représentation graphique d'une liste d'entiers

Afin de réserver l'espace mémoire nécessaire pour stocker une cellule de liste chaînée, il faut connaître la taille, en octets, d'une telle cellule. Ceci peut se faire laborieusement *à la main* mais peut également être obtenu beaucoup plus simplement avec un appel à la fonction `sizeof` ; `sizeof(struct cellule)` retourne le nombre d'octets utilisés par une cellule. Cette approche est d'autant plus conseillée que la taille nécessaire, pour stocker un `int` par exemple, peut varier d'un système à l'autre.

Enfin, le pointeur retourné par la fonction `malloc` n'a pas de type, ce qui *dérange* le compilateur `C`. Afin de lui donner le bon type, ici `Liste`,

i.e. pointeur de cellule, il faut utiliser une opération dite de "*cast*". Cette opération permet de transformer une donnée d'un certain type en un autre. Ici, cette transformation est purement formelle et ne modifie pas l'adresse du pointeur ; elle est juste utilisée pour "*rassurer*" le compilateur. Afin d'effectuer ce *cast*, on indique entre parenthèses le nouveau type devant la donnée à trans-typer, ici le résultat de la fonction `malloc`. On obtient donc finalement l'instruction suivante permettant de réserver une nouvelle cellule et de stocker son adresse dans une variable `nouv` de type `Liste` :

```
nouv = (Liste) malloc(sizeof(struct cellule));
```

Exercice 1 *Écrire une fonction dont l'en-tête est*

```
Liste cons(int v, Liste l)
```

et qui retourne un pointeur vers une liste obtenue en plaçant l'entier `v` *en tête de la liste* `l`*.*

Exercice 2 *Écrire une fonction dont l'en-tête est*

```
void print_liste(Liste l)
```

qui affiche le contenu d'une liste d'entiers à l'écran.

Exercice 3 *Écrire une fonction dont l'en-tête est*

```
Liste insere(int v, Liste l)
```

qui insère une valeur `v` *à sa place dans une liste* `l` *supposée triée par ordre croissant et qui retourne la liste ainsi obtenue comme résultat de la fonction. On ne demande pas à ce que cette fonction libère la mémoire devenue inutile et on conseille fortement d'utiliser une approche récursive.*

Exercice 4 *Écrire une fonction dont l'en-tête est*

```
Liste tri_insertion(Liste l)
```

qui trie la liste `l` *et retourne la liste ainsi obtenue. On utilisera la fonction d'insertion précédemment programmée et l'algorithme de tri par insertion.*

2 Comment trier mieux qu'en $\Theta(n \log n)$?

Nous avons démontré en cours qu'il n'était pas possible de trouver un algorithme de tri nécessitant en moyenne moins de $\Theta(n \log n)$ opérations de comparaison entre éléments. Ce résultat s'applique cependant à des données très générales et il est possible de faire mieux si l'on ajoute des hypothèses supplémentaires sur la distribution statistique des données à trier.

L'idée du tri dit "*par paquets*" est très simple ; on commence par définir un tableau de k listes vides, les "*paquets*". Ensuite, on parcourt l'ensemble

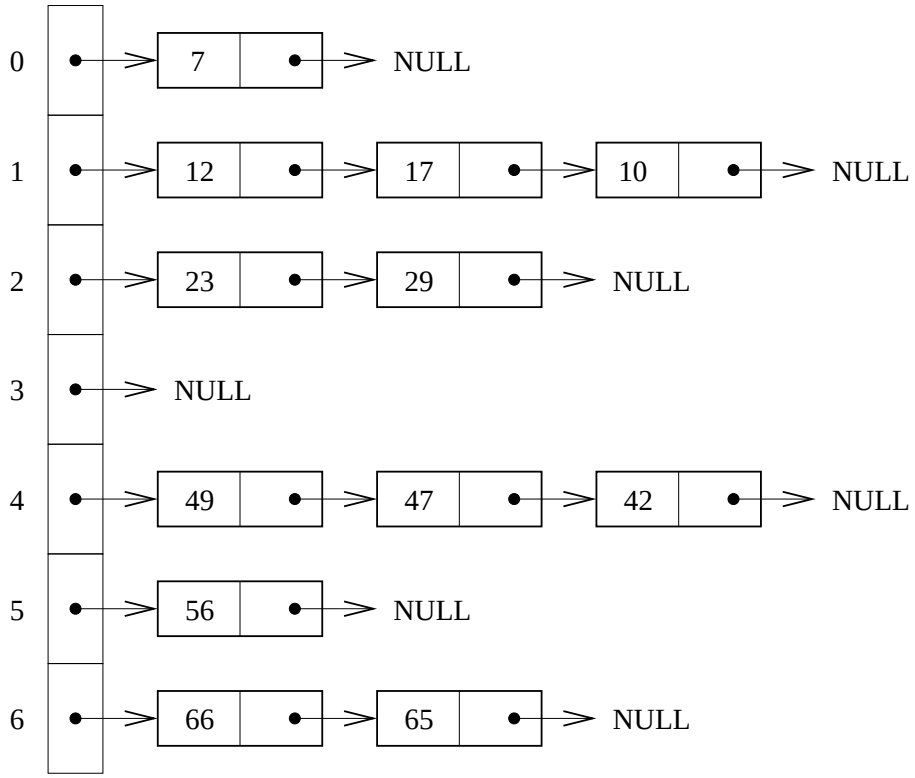

FIGURE 2 – Répartition en $k = 7$ paquets de $n = 12$ entiers compris entre 0 et $N = 70$

des n éléments à trier et on les range dans les différentes listes selon un règle garantissant un remplissage équitable des différents paquets et de manière à ce que les paquets soient "mutuellement triés". Par exemple, si l'on sait que les entiers à trier sont uniformément répartis entre 0 et N, et qu'il y a k paquets, on place un entier x dans la liste numéro $\left\lfloor \frac{x \times k}{N} \right\rfloor$.

Une fois les éléments ainsi répartis, on trie chaque liste avec un algorithme simple, tel que le tri par insertion. Le résultat final s'obtient ensuite en mettant bout à bout ces k listes triées.

Paradoxalement, une approche aussi simple permet d'obtenir un tri dont la complexité est linéaire, i.e. en $\Theta(n)$, et qui est donc meilleur que la borne démontrée de manière générale ! Ceci provient du fait que l'on a ajouté une hypothèse sur la distribution statistique des données à trier. Afin de s'en convaincre, il suffit d'observer que le nombre moyen d'éléments par paquet est n/k. Si le tri de m éléments nécessite de l'ordre de αm^2 opérations élémentaires, la complexité du tri par paquets est donc

$$\sum_{i=1}^{k} \alpha \left(\frac{n}{k}\right)^2 = \frac{k \times \alpha n^2}{k^2} = \frac{\alpha n^2}{k}$$

Si k est de l'ordre de n on obtient donc un nombre moyen d'opérations de l'ordre de $\alpha n = \Theta(n)$. Ce raisonnement peut de plus être rendu exact en

considérant l'influence des variations autour de la moyenne n/k (en utilisant la *variance* du nombre d'éléments par paquet).

Exercice 5 *Écrire une fonction dont l'en-tête est*

```
Liste tri_paquet(int k, Liste l)
```

qui trie la liste `l` *et retourne la liste ainsi obtenue au moyen d'un tri par paquets utilisant* k *paquets.*

3 Libération de la mémoire devenue inutile (pour aller plus loin)

Nous avons vu comment réserver des blocs de mémoire avec la fonction `malloc`. Tant que le programme ne s'arrête pas, tout bloc ainsi réservé reste alloué. Par conséquent, s'il devient inutile, l'espace mémoire qu'il occupe est gaspillé. Prenons l'exemple du programme suivant :

```
int *pointeur;

pointeur=(int *)malloc(10000);
pointeur=NULL;
```

Ce programme réserve 10000 octets et place l'adresse de ce bloc mémoire dans la variable `pointeur`. Cette dernière est le seul lien permettant d'accéder au bloc ainsi réservé. L'instruction suivante (`pointeur=NULL`) perd ce lien en modifiant le contenu de la variable `pointeur`. Le bloc réservé continu cependant d'occuper de la mémoire sans pour autant être utilisable puisque l'on ne sait plus où il se trouve !

Certains langages modernes, tel que `Java`, savent détecter ces situations et libèrent automatiquement les blocs devenus inutiles. En `C`, ce n'est pas le cas et cette libération doit être faite *à la main*, au moyen de la commande `free`. Cette dernière reçoit en argument l'adresse d'un bloc réservé avec `malloc` et le libère. Il est important de noter qu'il est inutile de préciser la taille du bloc à libérer, le système mémorisant la taille des blocs alloués.

Exercice 6 *Écrire une fonction libérant l'ensemble des cellules occupées par une liste chaînée.*

Exercice 7 *Reprendre la fonction d'insertion de manière à libérer tout bloc de mémoire devenu inutile.*

Exercice 8 *Reprendre la fonction de tri par paquet de manière à libérer tout bloc de mémoire devenu inutile.*

Une manière très simple de savoir si un programme gaspille de la mémoire consiste à insérer les fonctions visées dans une boucle infinie et à faire tourner dans une autre fenêtre UNIX le programme `top`. Ce dernier indique, pour chaque processus, de nombreuses informations, dont la mémoire occupée. Si certains blocs ne sont pas libérés, la mémoire utilisée par le programme croît alors très rapidement.