C 11 : structures de données chaînées

Juliusz Chroboczek

5 décembre 2023

La taille d'une structure est la somme des tailles de ses champs ; il n'est donc pas possible d'inclure une structure dans elle-même. Par contre, les pointeurs ont une taille fixée, et il est donc possible d'inclure dans une structure un pointeur sur la structure elle-même, et donc de construire une « chaîne » de structures toutes identiques. Cette technique permet d'implémenter les structures de données *inductives* ou *récursives*, telles que les listes et les arbres.

1 Listes chaînées

Par définition, une liste est construite inductivement avec les règles suivantes :

- la liste vide Λ est une liste ;
- si v est une valeur et l est une liste, alors $v \cdot l$ est une liste.

Nous représenterons une liste vide par le pointeur `NULL`, et une liste non-vide par un pointeur sur une « cellule » :

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

Il est très facile d'ajouter une cellule en tête de liste :

```
struct cell *cons(int v, struct cell *l) {
    struct cell *c;
    c = malloc(sizeof(struct cell));
    if(c == NULL)
        return NULL;
    c->v = v;
    c->next = l;
    return c;
}
```

Il est tout aussi facile de supprimer la première cellule d'une liste :

```
struct cell *rest(struct cell *l) {
    struct cell *r = l->next;
    free(l);
    return r;
}
```

Par contre, d'autres opérations sont coûteuses. Par exemple, pour calculer la longueur d'une liste il faut la parcourir toute entière :

```
int list_length(struct cell *l) {
    struct cell *p = l;
    int n = 0;
    while(p != NULL) {
        n++;
        p = p->next;
    }
    return 0;
}
```

Efficacité des listes Les listes sont une structure de données très inefficace. Tout d'abord, elles consomment de la mémoire comme si c'était gratuit : chaque cellule contient un pointeur, qui occuppe 8 octets par élément sur un système à 64 bits. Ensuite, les cellules d'une liste sont allouées séparément, ce qui fait qu'elles n'ont aucune localité en mémoire : un parcours de liste accède à des adresses complètement aléatoires, ce qui empêche le matériel d'optimiser les accès. En pratique, les tableaux et les *slices* sont presque toujours plus efficaces que les listes.

À quoi les listes servent-elles, alors ? À introduire les arbres.

2 Arbres binaires

Un *arbre binaire* est une généralisation des listes au cas où chaque élément est chaîné à deux successeurs. Inductivement,

- l'arbre vide Λ est un arbre ;
- si v est une valeur et l et r sont deux arbres, alors (l, v, r) est un arbre.

Dans un arbre, on parle de *nœuds* qui sont chaînés à leurs deux *fils* — le *fils gauche* et le *fils droit*.

```
struct node {
    int v;
    struct node *left, *right;
}
```

À la différence des listes, qui ne sont que rarement utiles, les arbres permettent de construire des structures de données qui sont difficiles ou impossibles à construire avec des tableaux ; par exemple, presque toutes les bases de données sont construites à partir d'une structure arborescente, le *B-tree*. Nous verrons en TP les *arbres binaires de recherche* (ABR), une structure beaucoup plus simple qui est relativement efficace dans le cas moyen (mais pas dans le cas pire).