

Listes chaînées (0)

Adrien Reboisson

15th December 2005

1 Rappels

1.1 Les listes chaînées

Une liste chaînée est une structure qui permet de contenir des informations, au même titre qu'un tableau. Elle est constituée d'un ensemble d'éléments (appelés *maillons*) chaînés entre eux; c'est à dire où chaque maillon, en plus de contenir des informations utiles, contient un champ qui désigne son successeur.

Une liste chaînée n'est pas un type du langage, c'est juste un concept, une manière de ranger en mémoire des structures (**struct**) qui utilise la technique décrite dans le paragraphe précédent. On parle alors de *structures autoréférentielles*, puisqu'on a une structure de données qui fait référence à une autre structure de même type. Il y a plusieurs variantes selon le nombre et la nature des références entretenues par la structure (ici, on s'intéresse à des listes chaînées simples, mais il existe aussi par exemple des listes dites *doublement chaînées* qui possèdent un lien vers leur successeur, mais aussi un lien vers leur prédécesseur).

Chaque maillon peut être considéré de manière indépendante. Par conséquent, les opérations d'ajout ou de suppression d'éléments sont parfois plus pratiques qu'avec d'autres structures de données. Par exemple, alors que l'insertion d'un élément au milieu d'un tableau nécessite le décalage de la moitié des éléments qu'il contient, cette opération se résume ici à l'allocation d'un nouveau maillon et la modification d'un prédécesseur et d'un successeur. Par contre, puisque les éléments sont indépendants ils ont été alloués séparément et ne sont donc pas rangés en mémoire de manière contiguë... Donc, contrairement au tableau, il n'y a pas moyen d'accéder à l'élément n sans avoir parcouru ses $(n - 1)$ prédécesseurs.

Puisque une liste chaînée peut contenir un nombre indéfini d'éléments et que ceux-ci doivent être accessibles de manière globale dans le programme, chaque maillon doit être alloué de manière dynamique avec **malloc()**. Il faut par conséquent dans toutes les opérations de manipulation de la liste faire attention à ne jamais perdre le lien sur un maillon de la liste avant de l'avoir libéré avec **free()**. Sinon, bonjour les fuites mémoire !

Les liens sont naturellement réalisés avec des *pointeurs* ce qui est compatible avec la manière d'allouer les maillons (**malloc** renvoyant une adresse...). Le

dernier élément de la liste se voit attribuer **NULL** comme adresse de successeur, puisque c'est la seule constante qui peut désigner une adresse pointant un objet invalide.

Voici un exemple de liste chaînée qui sert dans le système d'exploitation virtuel **Zindoz**. Il s'agit d'une liste mémorisant les utilisateurs définis sur la machine ainsi que leurs propriétés :

```
typedef struct _zin_user_t
{
    /* Informations utiles stockées */
    zin_userid_t userid; /* ID de l'utilisateur */
    char name[32]; /* Nom de l'utilisateur */
    char passwd[32]; /* Mot de passe de l'utilisateur */
    char shell[128]; /* Shell de l'utilisateur */
    zin_groupid_t gid; /* Groupe parent de l'utilisateur */
    /* Pointeur vers l'élément suivant */
    struct _zin_user_t *next;
} zin_user_t;
```

Si 8 utilisateurs sont enregistrés sur la machine, chaque utilisateur sera décrit par une structure **zin_user_t** où le champ **next** de l'élément n désignera l'élément $n + 1$. Pour les raisons décrites précédemment, le champ **next** du 8^e utilisateur contiendra la valeur **NULL**.

On a plusieurs moyens de désigner une liste chaînées. Les deux plus courants se choisissent en fonction de son degré d'amour des pointeurs... Soit on manipule directement la tête de la liste (qui n'est qu'une structure comme une autre), soit *un pointeur vers la tête de la liste*. Ce n'est pas par plaisir qu'on utilise un pointeur, c'est pour gérer le cas de la liste vide. Dans la première solution, on ne peut pas réellement représenter une liste vide puisque on utilise la tête pour représenter la liste, et que la tête compte déjà pour un élément. Il fait alors soit ajouter une variable pour dire "J'utilise cet élément pour représenter la liste, mais la liste est vide !" ou simplement ne jamais tenir compte des valeurs de la tête et uniquement utiliser le champ pointeur, qui désigne alors le réel premier élément de la liste. Dans ce cas, une liste vide sera représenté par un maillon dont le champ désignant son successeur sera initialisé à **NULL**. Pour ces raisons, on nomme ce genre de construction des *listes chaînées à tête fictive* puisque l'élément de tête ne sert qu'à éviter de manipuler directement le pointeur de tête.

Quand on n'est grand, on n'a pas peur des pointeurs, et on utilise comme tête de liste un pointeur sur son premier élément. Ainsi, une liste chaînée vide est désignée directement par un pointeur initialisé à **NULL**.

Conceptuellement, on a la construction suivante :

```
typedef struct _maillon_t
{
    /* Données */
```

```

...
/* Pointeur sur le successeur */
struct _maillon_t* suivant;
} maillon_t, *tete_liste_t;

```

Ici, un maillon est représenté par le type `maillon_t` (ou la structure `struct _maillon_t`) et la tête de la liste par le type `tete_liste_t` (puisque la tête est un pointeur vers le premier élément), ou bien directement par le type `maillon_t*`.

On pourra faire l’analogie avec un train de marchandises, où un maillon est un wagon. Pour supprimer un wagon d’un train, on n’a qu’à le faire exploser et raccorder son prédécesseur et son successeur. Pour ajouter un wagon, il faut séparer son futur prédécesseur de son futur successeur, déposer le wagon avec une jolie grue sur la voie, puis raccorder le nouveau venu avec ses voisins. Pour manipuler le train, on se sert uniquement du wagon de tête (même si le wagon de tête est ferroviairement une locomotive, mais après tout ce n’est qu’un cas particulier de wagon...), et pour aller au wagon n alors qu’on est dans la locomotive, il faut croiser ses $n - 1$ prédécesseurs...

1.2 Opérations courantes

Quelques rappels sur les opérations élémentaires que l’on peut effectuer sur une liste chaînée.

1.2.1 Allocations des maillons

Sans surprise, l’allocation des maillons se réalise avec `malloc()` :

```

/* Allocation du maillon */
maillon_t* m = (maillon_t*) malloc(sizeof(maillon_t));
if (m != NULL)
{
    /* L'allocation a réussi. On peut initialiser les champs de la structure. */
    m->s1 = b1;
    m->suivant = suivant;
    ...
} else /* Pas assez de mémoire pour allouer le maillon */
{
    /* On quitte le programme ! */
    perror("malloc");
    exit(errno);
}

```

Lorsqu’on initialise un maillon, si le champ pointant vers son successeur n’est pas immédiatement affecté, il vaudra mieux pour des raisons évidentes de sécurité les initialiser à `NULL` pour que ceux-ci désignent bien des objets invalides et qu’on puisse se “rappeller” qu’il faudra un jour les affecter.

1.2.2 Parcours de la liste

Beaucoup de procédures de traitement sur des listes chaînées peuvent être écrites soit de manière récursive, soit de manière itérative. Bien que le premier cas soit souvent plus élégant, il est aussi plus coûteux en temps et en consommation mémoire puisqu'il faut à chaque appel de la fonction empiler un contexte sur la pile.

Pour exemple, voilà deux fonctions explorant et affichant le champ **texte** des maillons d'une liste chaînée de manière récursive. La première parcourt la liste de la tête vers la queue en affichant le champ, la seconde parcourt la liste de la tête vers la queue jusqu'à l'atteindre, puis affiche les valeurs en "remontant" ce qui permet d'obtenir un affichage inverse données :

```
void affiche_liste_endroit(maillon_t* m)
{
    if (m) /* Si le maillon existe */
    {
        /* Afficher son champ "texte" */
        printf("Valeur: %s\n", m->texte);
        /* Recommencer la même chose sur son successeur */
        affiche_liste_endroit(m->suivant);
    }
}

void affiche_liste_envers(maillon_t* m)
{
    if (m) /* Si le maillon existe */
    {
        /* Recommencer la même chose sur son successeur
           jusqu'à, en fait, atteindre la queue de la liste */
        afficher_liste_envers(m->suivant);
        /* Afficher son champ "texte" */
        printf("Valeur: %s\n", m->texte);
    }
}
```

Un exemple de parcours itératif :

```
void affiche_liste_iteratif(maillon_t* m)
{
    while (m)
    {
        /* Afficher son champ "texte" */
        printf("Valeur: %s\n", m->texte);
        m = m->suivant; /* On passe au maillon suivant */
    }
}
```

Certains pourront s'exclamer : “*Quoi ?!*” en modifiant le successeur de **m** à chaque tour de boucle, on perd le chaînage de la liste puisqu’à la fin de la fonction **m** vaut **NULL** et que donc par conséquent le pointeur de liste chaînée ne pointe plus du tout sur la tête”. Ce raisonnement est faux. **m** contient une copie de l’adresse pointée par la tête de la liste, puisque les variables paramètres contiennent toujours des copies des variables qui leur sont associées. C’est donc une copie du pointeur désignant la tête de liste que l’on modifie à chaque tour de boucle sur la ligne **m = m->suivant**. Néanmoins, il n’est pas interdit d’utiliser une variable temporaire pour des raisons de clareté :

```
void affiche_liste_iteratif(maillon_t* m)
{
    maillon_t* temp = m;
    while (temp)
    {
        /* Afficher son champ “texte” */
        printf(“Valeur: %s\n”, temp->texte);
        temp = temp->suivant; /* On passe au maillon suivant */
    }
}
```

1.2.3 Suppression de la liste

Pour supprimer toute la liste, on doit désallouer chaque maillon de manière individuelle avec **free()**. Que le parcours soit réalisé de manière itératif ou récursif, il faut éviter de tomber dans un piège : si on supprime le maillon *n* sans mémoriser son successeur (le maillon *n + 1*), on ne saura pas quel maillon continuer à supprimer une fois que l’élément *n* sera libéré de la mémoire.

```
void liberer_liste_rekursif(maillon_t* m)
{
    /* Le maillon existe ? */
    if (m)
    {
        /* On continue l’exploration de manière
           à aller “au bout” de la liste */
        liberer_liste_rekursif(m->suivant);
        /* Quand on est arrivé ici, les maillons
           successeurs de m ont été supprimés.
           On peut libérer le maillon m */
        free(m);
    }
}
```

Ici, on se déplace jusqu’à que le maillon n’ait plus de successeur, autrement dit, qu’on soit arrivé en queue de liste. A chaque rappel de **liberer_liste_rekursif()** un contexte a été empilé, dans lequel **m** désigne le maillon courant au moment

de l'appel : le dernier contexte empilé contient ainsi un pointeur vers le maillon de queue de liste. Lorsque le premier `liberer_liste_recuratif()` retourne, la queue est libérée, la fonction quittée, le contexte précédent restauré. A ce moment, l'exécution de la fonction appellante reprend à l'endroit où elle s'était arrêtée, juste avant la libération (`free`). `m` contient alors la valeur stockée dans le contexte précédent restauré, c'est à dire l'adresse de l'avant dernier maillon. Celui-ci est détruit, la fonction retourne, etc. : il s'en suit donc une cascade de libérations jusqu'à que la tête soit alors finalement supprimée.

Une procédure équivalente écrite de manière itérative doit utiliser un pointeur temporaire pour stocker le successeur de l'élément à supprimer, car celui-ci libéré, cette information est évidemment perdue :

```
void liberer_liste_iteratif(maillon_t* m)
{
    maillon_t* tmp;
    while (m)
    {
        /* On mémorise le successeur de l'élément courant */
        tmp = m->suivant;
        /* On supprime l'élément courant */
        free(m);
        /* On passe au successeur de l'élément supprimé */
        m = tmp;
    }
}
```

1.2.4 Accès à l'élément k de la liste

Il n'est pas très difficile d'accéder à l'élément numéro k d'une liste chaînée. Il suffit de se déplacer dans la liste en maintenant un compteur et de renvoyer l'élément lorsque le compteur atteint l'indice recherché :

```
maillon_t* element_k(maillon_t* liste, unsigned int k)
{
    /* Compteur initialisé à 1 */
    int cpt = 1;
    /* Tant que le compteur est inférieur à k et que
       le maillon existe, on avance */
    while (cpt < k && liste != NULL)
    {
        liste = liste->suivant;
        cpt++;
    }
    return liste;
}
```

Si l'élément k n'existe pas, il est clair que **NULL** est renvoyé. L'écriture de ce code pourrait au passage être réduite en utilisant directement k comme variable compteur et en la décrémentant pendant la boucle (**while** ($k > 1$)...).

Le même algorithme peut être écrit de manière récursif de manière beaucoup plus élégante :

```
maillon_t* element_k_recuratif(maillon_t* liste, unsigned int k)
{
    /* Élément inexistant ? On retourne NULL. */
    /* Sinon, si k=1, on a atteint l'élément qu'on cherche. */
    if (liste == NULL || k == 1)
        return liste;
    else /* Sinon, on recommence sur le reste de la liste... */
        return element_k_recuratif(liste->suivant, k-1);
}
```

1.2.5 Insertion en tête de liste

L'insertion d'un maillon en tête de liste est simple : il suffit de définir la liste chaînée existante comme successeur du maillon à chaîner. Si la liste chaînée existante est vide, il n'y a pas de problème puisque on associe **NULL** comme successeur du maillon à chaîner, ce qui correspond à dire qu'il est le seul de la liste.

La fonction suivante chaîne le maillon m en tête de la liste $liste$ et renvoie la liste modifiée :

```
maillon_t* chainer_tete(maillon_t* liste, maillon_t* m)
{
    assert(m != NULL); /* m doit exister */
    m->suivant = liste;
    return m;
}
```

Ici, puisque m désigne la nouvelle tête, il suffit de le renvoyer pour qu'on puisse manipuler la liste. Une autre manière de procéder équivalente aurait été de définir la fonction comme **void** (ne renvoyant rien), mais qui aurait modifié directement le pointeur $liste$. On aurait dû alors passer par un double pointeur (**maillon****)... mais pourquoi faire compliqué quand on peut faire simple ?...

1.2.6 Suppression en tête de liste

Supprimer un élément en tête de liste, c'est définir comme tête de liste son successeur. Il faut simplement veiller à préserver l'adresse de la tête afin de pouvoir ensuite libérer la mémoire disponible :

```
maillon_t* supprimer_tete(maillon_t* liste)
```

```

{
    maillon_t* tmp;
    assert(liste != NULL); /* La liste doit au moins cont. 1 élément */
    tmp = liste->suivant; /* Mémoriser le successeur pour le retourner */
    free(liste); /* On a supprimé la tête */
    return tmp; /* On retourne la nouvelle tête */
}

```

1.2.7 Insertion en fin de liste

Si on possède un pointeur p sur le dernier élément de la liste, l'insertion d'un maillon m comme queue de liste se résume à définir m comme successeur de p , en vérifiant bien que le successeur de m a bien été défini à **NULL** (si ce champ n'a pas été initialisé auparavant, la fin de liste ne sera jamais détectée et on pourra se heurter à d'inexplicables *Segmentation Fault...*).

Si ce pointeur n'est pas possédé, il faudra utiliser une boucle de parcours pour obtenir un pointeur sur la queue :

```

void chainer_queue(maillon_t* liste, maillon_t* m)
{
    if (liste) /* La liste est elle vide ? */
    {
        maillon_t* tmp = liste;
        /* Non: on la parcourt pour arriver au dernier élément */
        while (tmp->suivant)
            tmp = tmp->suivant;
        /* "liste" contient à ce stade un pointeur sur la queue */
        m->suivant = NULL; /* m est désormais la queue... */
        tmp->suivant = m; /* ...de la nouvelle liste */
        return liste; /* On renvoie la tête de liste, tête non modifiée */
    } else {
        /* Oui: alors désormais m représente la liste.
           On vérifie bien que son successeur est initialisé
           à NULL, et on renvoie son adresse */
        m->suivant = NULL;
        return m;
    }
}

```

Le code ci-dessus est un exemple itératif (il existe bien évidemment des versions récursives de cet algorithme). Il traite deux cas : soit la liste est vide, et alors le maillon à chaîner représente à lui seule la nouvelle liste, soit la liste n'est pas vide et on doit la parcourir pour obtenir un pointeur vers sa queue afin de lui chaîner le maillon et enfin de renvoyer la tête de la liste. Cette fois-ci on a du utiliser impérativement une variable temporaire **tmp** pour naviguer dans la liste. Le cas contraire, on aurait modifié **liste** et le renvoi de la tête de liste

n'aurait été possible (puisqu'à ce moment là, **liste** aurait pointé sur la queue de la liste...).

1.2.8 Insertion dans la liste (position quelconque)

Si m est un maillon à insérer, il aura forcément dans la liste un prédécesseur l et un successeur n (cas particulier: si m est la tête de la liste alors $l = NULL$ et si m est la queue de la liste alors $n = NULL$). L'opération de chaînage constitue alors à :

- Définir n comme successeur de m : **m->suivant = m**,
- Si l existe, définir m comme successeur de l : **if (l) l->suivant =m**.

La difficulté est évidemment d'obtenir le prédécesseur de l'élément, ce qui à la différence du successeur ne peut s'obtenir dans ce type de liste qu'en effectuant un parcours dans lequel on a une variable qui a un "temps de retard" initialisée à **NULL** mémorisant toujours le dernier prédécesseur rencontré.

1.2.9 Suppression dans la liste (position quelconque)

La démarche est similaire à celle utilisée précédemment. Si m est un maillon à supprimer, il a alors forcément un prédécesseur l et un successeur n , **NULL** autorisés dans le cas de la tête et/ou de la queue de la liste. Alors, l'opération de suppression constitue en la simple définition de n comme successeur de l , si l existe. Comme tout cas de suppression, on veillera à garder un pointeur temporaire sur l'élément à supprimer afin de pouvoir appeler **free()** une fois le chaînage entre l et n effectué.

2 Présentation du problème

2.1 Types de données utilisés

On considère encore une fois notre système d'exploitation virtuel, **Zindo**. Dans **Zindo**, le contenu des répertoires est stocké sous la forme d'une liste chaînée, ou chaque maillon représente un élément du dossier. La structure descriptive est relativement simple : elle contient un nombre identifiant le fichier de manière unique (pour retrouver à partir de ce nombre les données qui y sont attachés), le nom du fichier et quelques attributs. On suppose de plus qu'un dossier ne contient pas autre chose que des fichiers. Voici sa représentation en C :

```
/* Quelques types personnalisés pour la beauté de la structure... */
typedef struct unsigned int zin_entryid;
typedef struct unsigned int zin_date;
typedef struct unsigned int zin_entryattribs;
```

```

/* La structure représentant un élément dans un dossier */
typedef struct _zin_direntry_t
{
    zin_entryid id; /* ID du fichier */
    char name[64]; /* Nom du fichier */
    zin_date moddate; /* Date de dernière modification */
    zin_entryattrs attr; /* Attributs divers */
    struct _zin_direntry* next; /* Pointeur vers l'élément suivant */
} zin_direntry_t;

```

On s'intéresse aux opérations basiques que l'on pourrait effectuer sur cette structure.

2.2 Fonctionnalités à implémenter

Attention, ne pas se laisser abuser par le nom apparemment compliqué des fonctions et des structures. Il ne s'agit que d'opérations *élémentaires* sur les listes chaînées, inutile de chercher midi à quatorze heures :-)

1. On suppose qu'une structure utilisée dans un autre module possède une variable de type `zin_direntry_t*` pour représenter le contenu d'un répertoire. Si le répertoire représenté est vide, que peut-on dire sur le champ cité précédemment ?
2. Ecrire la fonction `zin_direntry_t* zin_initdir(void)` qui renvoie le descripteur d'un dossier vide.
3. Ecrire la fonction `zin_direntry_t zin_allocentry(zin_entryid id, char* name, zin_date dt, zin_entryattrs attr, zin_direntry_t* next)` qui alloue un nouvel élément à partir d'un ID de fichier, d'un nom de fichier, d'une date de modification, d'attributs d'éléments et d'un pointeur vers un successeur, et qui renvoie un pointeur vers l'élément créé.
4. Ecrire la fonction `void zin_releaseentry(zin_direntry_t* t)` qui libère un descripteur alloué précédemment avec `zin_allocentry()`.
5. Ecrire la fonction `zin_direntry_t* zin_addentry(zin_direntry_t* list, char* name, zin_date dt, zin_entryattrs attr, zin_direntry_t* next)` qui ajoute un fichier dans le répertoire décrit par `list` et qui renvoie la liste modifiée. L'élément est ajouté en tête de liste. Conseil: si on a écrit `zin_allocentry`, c'est peut-être pour la réutiliser ici...
6. Ecrire la fonction `void zin_printdir(zin_direntry_t* t, FILE* f)` qui affiche la liste des fichiers contenus dans le dossier décrit par `t` sur le flux `f` sous forme récursive puis sous forme itérative.
7. Ecrire la fonction `void zin_removedir(zin_direntry_t* t)` qui supprime le contenu entier d'un dossier, sous forme récursive puis sous forme itérative.

8. On veut maintenant que la liste soit maintenue triée lors de l'insertion par rapport aux noms des fichiers (champ **name**). Réécrire la fonction **zin_addentry()** sous forme itérative puis récursive en utilisant **strcmp** pour comparer les chaînes (inclure `<string.h>` si besoin).
9. On veut pouvoir supprimer un fichier d'un nom donné dans un répertoire. Ecrire la fonction **zin_removeentry(zin_dirent_t* t, char filename, char* success)** qui cherche dans la liste si un fichier **filename** existe, et si oui, le supprime (itératif ou récursif). Si le fichier a été trouvé, **success** doit être positionné à 1, à 0 sinon.