TD 8 : Listes

Semaine du 19 mars 2007

1 Tables de hachage

Nous allons implémenter une technique qui permet de réaliser les mêmes opérations que sur des tableaux dont les indices de case appartiennent à un ensemble très grand et ceci de manière bien plus efficace qu'en utilisant un énorme tableau. Dans la suite de cet exercice, l'ensemble des chaînes de caractères nous servira d'indice de case. Pour cela, on va d'abord calculer à partir de la chaîne qui nous sert d'indice (appelée « clé »), un « haché » qui sera le véritable indice de la case dans un tableau (qui, lui, est de taille raisonnable). Évidemment, la fonction qui calcule le haché ne peut être injective (plusieurs clés différentes peuvent produire le même haché), et on utilisera non pas un tableau donnant directement les valeurs associées aux clés mais un tableau de listes de couples (clé, valeur) où chaque liste du tableau correspond à un haché différent. Si la fonction qui calcule le haché est bien construite et que le tableau est suffisamment grand, les listes associées aux mêmes hachés seront courtes et les opérations d'insertion, de recherche et de suppression d'un élément seront rapides.

On travaille avec les types suivants :

```
typedef struct liste_s {
    char *cle;
    int valeur;
    struct liste_s *suivant;
} *liste_t;

typedef struct table_de_hachage_s {
    int taille;
    liste_t *tableau;
} *table_de_hachage_t;
```

La figure 1 montre une table de hachage représentant un répertoire téléphonique (à un nom correspond un numéro de téléphone).

Exercice 1 Écrire une fonction table_de_hachage_t cree_table_de_hachage(**int** taille) qui crée une table de hachage vide avec un tableau de la taille indiquée (mais qui ne contient que des listes vides puisqu'il n'y a pas encore d'élément dans la table de hachage).

Exercice 2 Écrire une fonction **void** detruit_table_de_hachage(table_de_hachage_t table) qui libère la mémoire occupée par une table de hachage donnée en argument.

Exercice 3 Écrire une fonction **int** hachage(table_de_hachage_t table, **char** *cle) qui calcule le haché d'une clé, c'est-à-dire l'indice du tableau de listes de la table de hachage que l'on doit utiliser pour placer les valeurs associées à la clé donnée en argument : pour cet exercice, le haché sera simplement la somme des valeurs associées aux caractères de la clé modulo la taille du tableau.

Exercice 4 Écrire une fonction **void** insere(table_de_hachage_t table, **char** *cle, **int** valeur) qui insère dans la table de hachage la valeur associée à la clé donnée en argument.

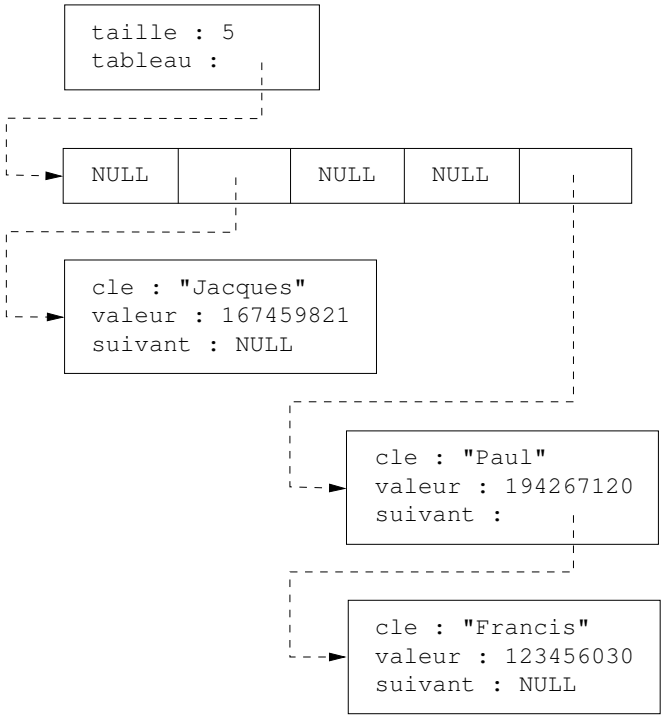

FIG. 1 – Une table de hachage ("Paul" et "Francis" ont le même haché)

Exercice 5 Écrire une fonction **int** recherche(table_de_hachage_t table, **char** *cle, **int** *valeur) qui cherche si une entrée de la table a pour clé celle indiquée en argument, et le cas échéant renvoie la valeur correspondante via le pointeur donné en argument. Cette fonction renvoie un entier qui indique si la recherche a abouti.

Exercice 6 Écrire une fonction **void** supprime(table_de_hachage_t table, **char** *cle) qui efface l'éventuelle entrée de la table ayant pour clé celle donnée en argument (et on suppose qu'il y a au plus une telle entrée dans la table).

Exercice 7 — *Bonus* Écrire une fonction **void** affiche_numero(**int** numero) qui affiche un numéro de téléphone donné sous la forme d'un entier : ainsi si numero = 564302120, la fonction devra afficher `05.64.30.21.20` (Pourquoi doit-on omettre le 0 du préfixe téléphonique ?). Écrire dans la fonction main() une séquence d'opérations qui crée une table de hachage représentant un répertoire téléphonique, insére plusieurs entrées dans la table (noms et numéros), effectue une recherche puis supprime l'élément trouvé, fait à nouveau la même recherche et enfin détruit la table.

2 Files

Une file est une structure de donnée dans laquelle les premiers éléments ajoutés sont les premiers à être récupérés. Nous allons implémenter une file à l'aide d'une liste simplement chaînée : on maintient non seulement un pointeur vers le début de la liste qui contient les éléments de la file, mais aussi un pointeur vers la fin. Pour enfiler un élément, on l'ajoute à la fin de la liste, tandis que pour défiler un élément, on retire l'élément du début de la liste.

On utilise le type suivant, où liste_t est le type de liste défini à l'exercice précédent :

```
typedef struct file_s {
  liste_t debut;
  liste_t fin;
} *file_t;
```

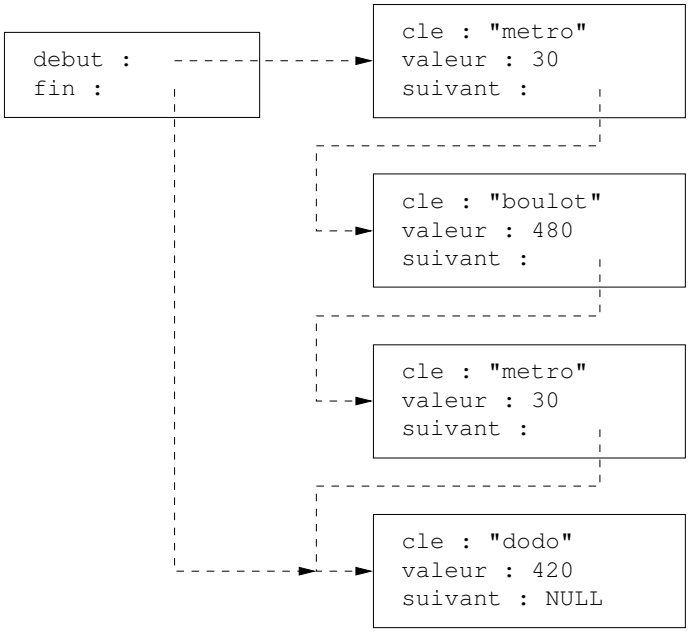

FIG. 2 – Une file

La figure 2 montre un exemple de file contenant une liste de tâches à effectuer (nom de la tâche et durée). Une file vide aura ses deux champs debut et fin à NULL.

Exercice 8 Écrire les fonctions

```
file_t cree_file(void);
int est_vide(file_t file);
void detruit_file(file_t file);
```

qui, respectivement,
- alloue la mémoire nécessaire à une file et l'initialise pour représenter la file vide,
- indique si une file donnée en argument est vide,
- libère la mémoire occupée par une file donnée en argument.

Exercice 9 Écrire une fonction **void** enfile(file_t file , **char** *cle, **int** valeur) qui enfile la chaîne de caractère cle et l'entier valeur à la fin de la file donnée en argument.

Exercice 10 Écrire une fonction **void** defile(file_t file , **char** **cle, **int** *valeur) qui retire de la file donnée en argument son premier élément et renvoie les champs correspondants via les pointeurs cle et valeur.

Exercice 11 — *Bonus* Réécrire une fonction **void** detruit_file2(file_t file) qui libère la mémoire occupée par une file donnée en argument mais sans parcourir explicitement la liste chaînée de la file. Écrire dans la fonction main() une séquence d'opérations qui crée une file de tâches, y insére plusieurs tâches (nom de la tâche à réaliser et durée), affiche les premiers éléments en les défilant et enfin détruit la file.