TD6
Listes doublement chaînées – Tables de hachage

Langage C (LC4)

Semaine du 8 mars 2010

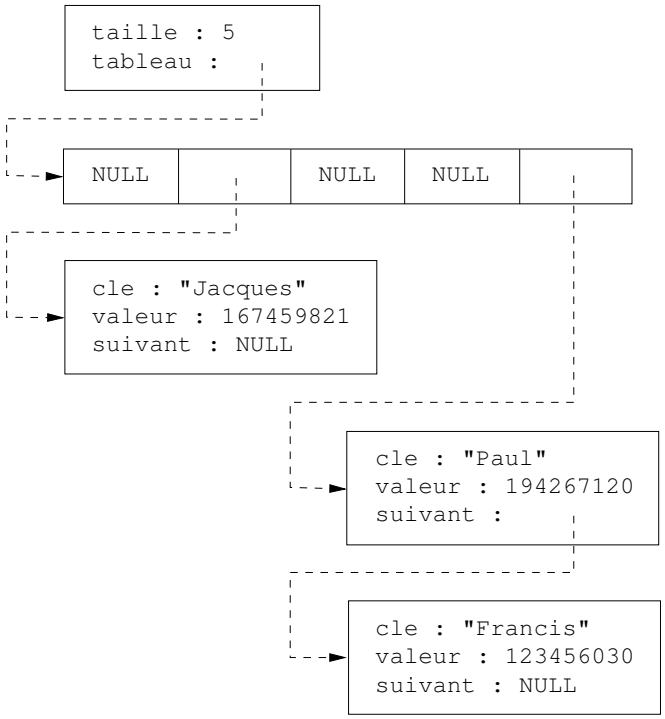
1 Tables de hachage

En programmation, on est souvent amené à utiliser des listes d'associations. Une liste d'association associe des valeurs à des indices. Par exemple, si on implémente un annuaire téléphonique, on veut pouvoir retrouver rapidement le numéro de téléphone (la valeur) correspondant à un nom donné (l'indice). Si les indices sont des entiers, on peut utiliser des tableaux pour implémenter nos liste d'association. Le gros avantages est alors que l'accès à une valeur se fait en temps constant. Par contre, non seulement on est limité à avoir des indices entier, mais en plus, si les indices sont pris dans un ensemble très grand, on est obliger d'allouer un énorme tableau (songer à un annuaire inversé). Les tables de hachages sont une structure de donnée associant des valeurs à des indices sans les inconvénients des tableaux, mais permettant tout de même un accès en temps constant en moyenne.
Dans la suite de cet exercice, on utilisera l'ensemble des chaînes de caractères comme 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 efficaces.
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 suivante montre une table de hachage représentant un répertoire téléphonique (à un nom correspond un numéro de téléphone). `"Paul"` et `"Francis"` ont le même haché.



Question 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).

Question 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.

Question 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.

Question 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.

Question 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.

Question 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 seul telle entrée dans la table).

2 Listes doublement chaînées

On veut implémenter des listes circulaires, où chaque élément pointe vers son prédécesseur et son successeur. On peut utiliser le type suivant :

```
typedef struct cellule {
   struct cellule * precedent;
   struct cellule * suivant;
   int contenu;
} * liste_circulaire;
```

2.1 Éléments alloués à la demande

Question 7. Écrire une fonction `void insere_cellule(liste_circulaire l, struct cellule * c)` qui insère la cellule `c` (supposée non `NULL`) entre `l` et `l.suivant` si `l` n'est pas `NULL`, et relie `c` à elle-même sinon.
En déduire une fonction `liste_circulaire insere(liste_circulaire l,int x)` qui insère l'entier `x` entre entre `l` et `l.suivant` si `l` n'est pas `NULL`, et crée une liste contenant juste `x` dans le cas contraire. Elle doit renvoyer en résultat un pointeur vers le nœud de la liste créé pour contenir `x`.

Question 8. Écrire une fonction `liste_circulaire supprime_cellule(liste_circulaire l)` qui supprime de la liste l'élément pointé par `l` de la liste circulaire à laquelle il appartient, mais sans le libérer de la mémoire,. Elle doit renvoyer `NULL` si la liste est vide après l'effacement de `l`, et `l.precedent` sinon.
En déduire une fonction `liste_circulaire supprime(liste_circulaire l)` qui, en plus de supprimer l'élément, le libère de la mémoire.

Question 9. Écrire une fonction `int compte(liste_circulaire l)` qui compte le nombre d'éléments dans la liste pointée par `l`.

Question 10. Écrire une fonction `void inverse(liste_circulaire l)` qui inverse l'ordre des éléments de la liste `l` (sans toucher aux champs `contenu`).

Question 11. Écrire une fonction `liste_circulaire fusionne(liste_circulaire premier, liste_circulaire deuxieme` qui fusionne les deux listes circulaires de telle sorte que le dernier élément de la première liste ait comme successeur le premier élément de la seconde. Si l'une des deux listes est vide, renvoie l'autre.

2.2 Espace alloué à l'avance (*bonus*)

Le problème, c'est que lors d'une série d'insertions et suppressions (par exemple pour la gestion d'une file d'attente), chaque opération nécessite une allocation ou libération de mémoire. Pour éviter de répéter trop souvent ces opérations coûteuses, on se propose d'allouer un espace à l'avance pour y stocker une liste chaînée à `n` éléments. Pour cela, on va définir le type suivant :

```
typedef struct liste_preallouee {
  liste_circulaire liste;
  liste_circulaire cellules_libres;
} * liste;
```

qui dans le champ `liste` contient la liste chaînée proprement dite, et dans le champ `cellules_libres` contient des cellules libres à utiliser lors de l'ajout d'un élément.

Question 12. Écrire une fonction `void enchaine_cellules(int n, struct cellule * t)` qui, étant donné un tableau `t` de `n` cellules, les relie en cercle. On n'utilisera pas d'accès indicé (tel que `t[i]`), mais uniquement des incréments de pointeurs (`t++`, etc.)
Attention : `t` est un tableau de `struct` : ses cases sont des cellules et non des pointeurs vers cellules. En fait, `t` pointe vers la première cellule.
En déduire une fonction `liste_circulaire cercle(int n)` qui crée et renvoie un tableau de `n` cellules enchaînées en cercle, sans initialiser la valeur de ces cellules.

Question 13. Écrire une fonction `liste cree_liste(int n)` qui crée et renvoie une liste préallouée de `n` cellules, dont la liste effective sera initialement vide, et toutes les `n` cellules sont libres.

Question 14. Écrire une fonction `int insere2(liste li, int x)` qui ajoute un élément à la liste en prélevant une cellule libre dans le champ `cellules_libres`, et renvoie 1 en cas de succès, et 0 si l'élément ne peut pas être inséré faute de cellule libre.
On pourra réutiliser les fonctions `insere_cellule` et `supprime_cellule`.

Question 15. Écrire une fonction `void supprime2(liste li)` qui supprime la première cellule de la liste `li` (celle pointée par `li->liste`) et la reclasse parmi les cellules libres.

Question 16. Comment libérer l'espace total pris par une telle liste préallouée ? Peut-on directement utiliser `free` sur le champ `cellules_libres` ? Si non, rajouter un ou plusieurs champs à la `struct liste_preallouee` et proposer une solution.

Question 17. On veut écrire une fonction `void etend(liste l, int n)` étendant la liste `l` en y ajoutant `n` cellules libres supplémentaires et en les ajoutant à `cellules_libres`. Cependant, on ne veut pas les allouer une à une, mais en bloc.
La fonction standard `void * realloc(void * p, int taille)` permet de "redimensionner" l'espace mémoire pointé par `p`. C'est-à-dire que si on a `p = malloc(taille_p)`, alors les données comprises entre 0 et le minimum de `taille` et `taille_p` sont conservées. `realloc` renvoie `p` sauf si la nouvelle taille est "trop grande", auquel cas `realloc` :
– alloue un nouvel emplacement mémoire
– y recopie **telles quelles** les données pointées par `p`
– libère l'ancien emplacement pointé par `p`
– et renvoie un pointeur vers le nouvel emplacement
Peut-on réutiliser `realloc` sur le tableau de cellules initialement alloué ? Si oui, ajouter un champ à la `struct liste_preallouee` pour ce tableau, puis proposer une fonction `etend`. Si non, justifier la réponse.

Question 18. Dans cette question, on va étendre la liste de `n` cellules supplémentaires en allouant un nouveau tableau de `n` cellules indépendant de l'existant. La `struct liste_preallouee` aura donc cette fois-ci un champ `liste_circulaire * toutes_listes` qui sera un tableau dont chaque case sera **un pointeur** vers une liste circulaire allouée lors d'une extension (ou initialement pour la première case).
On disposera également d'un nouveau champ `int nombre_listes` qui sera la taille du tableau `toutes_listes`.
Peut-on utiliser `realloc` ici ? Si oui, sur quel pointeur ? Proposer une fonction `etend`, ainsi qu'une fonction `libere` pour libérer tout l'espace.