Université Savoie Mont Blanc,
UFR SceM
INFO505, L3 Info / CMI Info
/ Math-Info

October 24, 2023

Problème 3, Tableaux associatifs

(Listes, pile d'exécution, table de hachage, pointeurs de fonction, généricité)

On cherche à représenter un dictionnaire, qui associe des valeurs à des mots. Ensuite, on veut pouvoir rapidement trouver un mot dans le dictionnaire, mais aussi ajouter ou enlever des mots. La structure choisie est la table de hachage, qui représente les collections d'éléments via un tableau de listes.

Par exemple, on peut se servir de telles structures pour compter le nombre de mots dans un texte. Si on prend le début du Petit Poucet de Charles Pergaud, le texte est:

```
Le petit poucet
Il était une fois un bûcheron et une bûcheronne qui avaient sept enfants, tous garçons;
l'aîné n'avait que dix ans, et le plus jeune n'en avait que sept.
On s'étonnera que le bûcheron ait eu tant d'enfants en si peu de temps ; mais c'est que
sa femme allait vite en besogne, et n'en avait pas moins de deux à la fois.
```

On place les mots (après les avoir repérés comme dans le problème 1, puis mis en minuscules) dans une table de hachage, composée de 16 listes. Cela donne:

```
[  0]
[  1]-->([de]=2, h=3671)-->([étonnera]=1, h=1c687041)
[  2]-->([eu]=1, h=3f02)-->([bûcheron]=2, h=b0e328b2)-->([poucet]=1, h=5759b792)-->([petit]=1, h=
[  3]-->([à]=1, h=ffffcc63)-->([c]=1, h=63)-->([tant]=1, h=255d7653)-->([s]=1, h=73)-->([en]=4, h
[  4]-->([deux]=1, h=26a9a034)-->([moins]=1, h=23708e84)-->([vite]=1, h=20973784)-->([femme]=1, h
[  5]-->([la]=1, h=3455)-->([besogne]=1, h=1a088d95)-->([qui]=1, h=4c2cc5)-->([il]=1, h=3a35)
[  6]-->([pas]=1, h=535d46)
[  7]-->([temps]=1, h=2416f9a7)
[  8]-->([peu]=1, h=54d1a8)
[  9]-->([que]=4, h=494849)-->([et]=3, h=3e79)-->([le]=3, h=3679)
[  a]-->([avaient]=1, h=18a63a9a)-->([était]=1, h=56b5139a)
[  b]-->([allait]=1, h=56afb39b)-->([enfants]=2, h=5b24360b)-->([bûcheronne]=1, h=b12c6ccb)-->([
[  c]-->([sa]=1, h=345c)-->([est]=1, h=541ffc)-->([ans]=1, h=53642c)-->([aîné]=1, h=395bd61c)-->
[  d]-->([on]=1, h=3b4d)-->([jeune]=1, h=23790d6d)-->([dix]=1, h=56ff1d)-->([garçons]=1, h=f1560
[  e]-->([ait]=1, h=541a9e)-->([plus]=1, h=2510c60e)-->([n]=3, h=6e)-->([une]=2, h=49448e)
[  f]-->([avait]=3, h=21d1ee3f)
```

Pour bien comprendre le principe de la table de hachage, les indices et les hachés sont ici affichés sous forme hexadécimale. On stocke les paires clé/valeur (ici les clés sont les mots et les valeurs sont des entiers représentant le nombre de fois où le mot apparaît) non dans une seule liste, mais dans plusieurs (beaucoup) de listes. On se donne ensuite un moyen de savoir dans quelle liste tombe chaque mot: la fonction de hachage. Celle-ci calcule (rapidement) un nombre à partir d'une clé. Pour que la table soit efficace, il faut que la fonction de hachage donne généralement des valeurs différentes pour une clé différente. Ici, nous avons pris cette fonction de hachage:

```
unsigned int h(char* s)
{
  static const unsigned int w[ 8 ] = { 1, 137, 47391, 5359101, 753, 193139197, 652389,
      349224 };
  unsigned int n = 0, i = 0;
  while ( *s != 0 )
    n += w[ i++ & 0x7 ] * (unsigned int) (*s++);
  return n;
}
```

Si w est le mot et N est le nombre de liste choisi pour la table, alors le mot w sera dans la liste $h(w) \mod N$. Dans l'exemple au-dessus, comme il y a 16 listes, on prend modulo 16 et on voit que les hachés d'une même ligne se terminent bien par le même chiffre hexadécimal que la ligne.

On observe aussi que les mots se dispersent naturellement dans la table et qu'ils ne s'accumulent pas dans une seule liste si la fonction de hachage est bien choisie. En pratique, on choisit souvent le nombre de listes N en fonction du nombre supposé n de paires clés/valeurs, et $N \approx 2n$.

Question 1. Liste simplement chaînée

Il faut d'abord une structure pour stocker les listes. On se donne le code suivant qui permet de stocker une liste d'entiers.

```
typedef int          Element;
typedef struct SList_Cell {
    Element            data;
    struct SList_Cell* next;
  }                  List_Cell;
typedef List_Cell*   Adr;
typedef Adr          List;

void     List_init     ( List* pL )         { *pL = NULL; }
Adr      List_first    ( List* pL )         { return *pL; }
Adr      List_next     ( List* pL, Adr a ) { return a->next; }
Element  List_value    ( List* pL, Adr a ) { return a->data; }
Element* List_adr_value( List* pL, Adr a ) { return &( a->data ); }
int      List_empty    ( List* pL, Adr a ) { return a == NULL; }
void     List_insert_first( List* pL, Element value ) {
  Adr pE   = (Adr) malloc( sizeof(List_Cell) );
  pE->data = value;
  pE->next = *pL;
  *pL      = pE;
}
void     List_insert_after( List* pL, Adr a, Element value ) {
  Adr pE   = (Adr) malloc( sizeof(List_Cell) );
  pE->data = value;
  pE->next = a->next;
  a->next  = pE;
}
```

Notez qu'on donne deux noms différents à 2 types identiques (List et Adr). Pourquoi ?

Sur le code suivant, tracez la pile et le tas d'exécution aux étapes (et dans les fonctions):

```
List L;
List_init( &L );                          // (1)
List_insert_first( &L, 12 );              // (2)
List_insert_after( &L, List_first( &L ) ); // (3)
```

Question 2. Eléments stockés dans les listes d'une table de hachage

Proposez la structure de données qui sera stockée dans les listes de la table de hachage.

```
typedef struct {
  ...
} Element;
```

Question 3. Table de hachage

Proposez maintenant la structure de données qui représentera la table de hachage. On veillera aussi à stocker le nombre d'éléments réellement présents dans la table.

```
typedef struct {
  ...
} HashMap;
```

Question 4. Initialisation de la table

Ecrivez maintenant la fonction qui initialise la table de hachage avec N listes différentes.

```
void  HashMap_init ( HashMap* pH, unsigned int N );
```

Question 5. Chercher un élément dans la table

On veut maintenant écrire une fonction qui recherche si une chaîne w est dans la table, si oui, retourne l'adresse de l'`Element`, sinon `NULL`.

```
Element*  HashMap_search ( HashMap* pH, char* w );
```

Question 6. Insérer une chaîne dans la table

Utilisez la fonction précédente pour maintenant faire une fonction qui ajoute un mot au dictionnaire ou augmente de 1 son nombre d'occurences si il est déjà dans le dictionnaire.

```
void  HashMap_update_word ( HashMap* pH, char* w );
```

Question 7. Généricité par pointeur de fonctions

Tout d'abord, on voit que notre structure de données est entièrement figée. Elle ne peut gérer que des clés qui sont des chaînes de caractères, des valeurs qui sont des entiers. De plus, la fonction de hachage est fixée.

Un premier moyen pour mettre un peu de souplesse dans notre structure de données est d'offrir à l'utilisateur le choix de sa fonction de hachage. Cela peut se faire par le biais des **pointeurs de fonction**. On peut définir un type `Fct` qui englobe toutes les fonctions de paramètres `(T1 t1, T2 t2, ... )` et qui retourne une valeur de type `R` ainsi:

```
typedef R (* Fct)(T1 t1, T2 t2, ... );
```

Ajoutez donc un champ dans la structure `HashMap` afin de stocker la fonction de hachage, et modifiez les fonctions `HashMap_*` pour l'utiliser.

NB: Les pointeurs de fonctions sont par exemple très utilisés pour coder les réactions (ou *callbacks*) dans les interfaces graphiques (ex: GTK).

Question 8. Généricité par pointeur `void*`

Malheureusement, dans l'état de nos structures liste et table de hachage, on ne peut modéliser que des dictionnaires ou tableaux associatifs mot vers entier. Si on veut faire des listes d'autres éléments, il faut tout recommencer alors même qu'une grosse partie du code se ressemblerait.

Une première solution pour rendre générique les valeurs stockées est d'utiliser les pointeurs `void*` comme élément d'une liste. Comme la valeur d'un pointeur est une adresse, celle-ci prend toujours la même taille indépendamment du type pointé. Mettez à jour les fonctions de gestion de liste vues ci-dessus avec cette approche.

```
typedef void*        Element; // type générique
typedef struct SList_Cell {
    Element            data;
    struct SList_Cell* next;
  }                  List_Cell;
typedef List_Cell*   Adr;
typedef Adr          List;

void     List_init     ( List* pL )        { *pL = NULL; }
Adr      List_first    ( List* pL )        { return *pL; }
Adr      List_next     ( List* pL, Adr a ) { return a->next; }
Element  List_value    ( List* pL, Adr a ) { return a->data; }
Element* List_adr_value( List* pL, Adr a ) { return &( a->data ); }
int      List_empty    ( List* pL, Adr a ) { return a == NULL; }
void     List_insert_first( List* pL, Element value, int size_byte );
void     List_insert_after( List* pL, Adr a, Element value, int size_byte );
```

Question 9. Inconvénients de l'approche par pointeur `void*`

Voyez-vous un inconvénient à cette approche par pointeur `void*` ? Par exemple, dessinez en mémoire une liste de 3 éléments.

Question 10. Généricité par macros préprocesseur

On peut utiliser le pré-processeur pour générer à la demande des types et fonctions génériques. On utilise le fait qu'une macro peut être paramétrée et que l'on peut concaténer les paramètres au texte avec l'opérateur ##. Ainsi:

```
#define PROTOTYPE_ADD( T ) \
   T add_##T( T t1, T t2 );

#define BODY_ADD( T ) \
   T add_##T( T t1, T t2 ) { \
     return t1+t2; \
   }

// Déclare des fonctions d'addition sur les int et double
PROTOTYPE_ADD( int )
PROTOTYPE_ADD( double )
// Définit des fonctions d'addition sur les int et double
BODY_ADD( int )
BODY_ADD( double )
```

Par exemple, pour définir une liste générique (les éléments sont de type T, le nom du type liste est U et le nom du type adresse d'un élément est A):

```
#define TYPEDEF_LIST_OF( T, U, A )            \
  typedef struct SList_Cell_##T {             \
    T                       data;             \
    struct SList_Cell_##T * next;             \
  } List_Cell_##T;                            \
  typedef List_Cell_##T * A;                  \
  typedef A               U;
```

A quoi ressemblerait alors les macros de déclaration et définition des fonctions ?

```
#define PROTOTYPE_LIST_OF( T, U, A )    \
  ...
#define BODY_LIST_OF( T, U, A )         \
  ...
```

On s'en servirait ainsi:

```
TYPEDEF_LIST_OF( int, ListI, AdrI )
PROTOTYPE_LIST_OF( int, ListI, AdrI )
BODY_LIST_OF( int, ListI, AdrI )

{
  ListI L;
  ListI_init( &L );              // L = ()
  ListI_insert_first( &L, 10 ); // L = (10)
  ListI_insert_first( &L, 5 );  // L = (5, 10)
  ListI_insert_first( &L, 12 ); // L = (12, 5, 10)
  ListI_end( &L );               // L = ()
}
```