Examen, INFO505 – Programmation C II, Session 1

Durée : 1h30

Documents autorisés : tous documents du cours/td/tp, notes manuscrites (nb : pas de livres)

Les exercices sont indépendants. Le barème est indicatif, et dépasse volontairement 20. La durée est de 2h pour les étudiants bénéficiant d'un 1/3 temps.

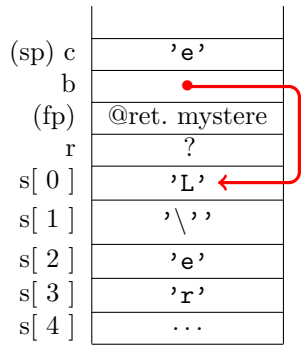
1 Connaissance du C et pile d'exécution (/5)

On vous donne le programme C suivant :

```c
#include <stdlib.h>
#include <stdio.h>
char* mystere( char* b, char c );

int main()
{
  char  s[] = "L'erreur est humaine";
  char* r   = mystere( s, 'e' );
  // (4)
  printf( "%s\n", r );
  free( r );
  return 0;
}
```

```c
char* mystere( char* b, char c )
{ // (1)
  int  n = 0;
  char* p = b;
  while ( *p != 0 )
    if ( *p++ != c ) n++;
  char* q = (char*) malloc( n + 1 );
  // (2) situation avant la boucle
  for ( p = q; *b != 0; b++ )
    if ( *b != c ) *p++ = *b;
  *p = 0;
  // (3) situation en fin de boucle
  return q;
}
```



1. (/2) Qu'affiche le programme et plus généralement que fait la fonction `mystere` ?
2. (/3) On affiche l'état de la pile d'exécution à l'étape (1). Afficher la pile d'exécution (et le tas d'exécution) de ce programme aux lignes (2), (3) et (4), comme précisé dans le programme.

(1) Le programme affiche : `L'rrur st humain`
Plus généralement, la fonction mystère alloue une chaîne de caractères qui est la copie de la chaîne `b` donnée en entrée moins tous les caractères `c` trouvés.

2 Elimination des données aberrantes (/3)

Ecrivez une fonction **void** filtre (**double*** debut, **double*** fin) qui prend un tableau de nombres en paramètres, et qui remplace chaque valeur par le médian des 3 valeurs formées par la valeur courante, celle d'avant et celle d'après. La première et la dernière valeur ne sont pas changées. On note que debut pointe au début du tableau, fin pointe juste après la dernière valeur valide, donc fin−debut est le nombre d'éléments.

```c
double v[] = { 3.0, 13.0, 16.0, 14.0, 8.0, 4.0, 7.0, 10.5, 12.0, 12.0 };
filtre( v, v+10 );
// v contient 3.0 13.0 14.0 14.0 8.0 7.0 7.0 10.5 12.0 12.0
```

```c
void filtre( double* debut, double* fin )
{
  if ( debut == fin ) return;
  double p = *debut++;
  while ( debut != fin ) {
    double* h = debut;
    double  c = *debut++;
    if ( debut == fin ) return;
```

```
      double n = *debut;
      if ( c > p && c > n )       *h = ( n < p ) ? p : n;
      else if ( c < p && c < n ) *h = ( n < p ) ? n : p;
      p = c; // update p wih *former* value
    }
  }
```

3 Ensembles par hachage ouvert (/13)

On propose une structure de données pour mémoriser un ensemble d'*Element*. L'idée des tables de hachage est de différencier la plupart des éléments par un code entier, appelé "haché" de l'élément. Ensuite, selon le haché d'un élément on le place dans une liste différente de la table de hachage. La table de hachage sera donc composée d'un tableau de N listes et un nouvel élément sera placé dans la liste qui correspond à son haché modulo N.

On a donc ces définitions quelque part :

```
typedef struct { int first; int second; } Element; // paire d'entiers
int Hache( Element e );  // retourne le haché de e: un entier qui dépend de e.
int Egal ( Element e1, Element e2 ); // vrai si e1 == e2

struct SCellule {
  Element val;            // L'élément stocké
  int     h;              // son haché (que l'on mémorise)
  struct SCellule* suiv; // un pointeur vers l'élément suivant
};
typedef struct SCellule Cellule;
typedef Cellule* Liste; // une liste est un pointeur vers le premier élément.
struct STable {
  int    N;              // nombre de listes dans la table.
  Liste* tab;            // tableau de listes d'éléments.
  int    nb;             // nombre d'éléments stockés dans la table
};
typedef struct STable Table;
```

Par exemple, supposons que la table de hachage contienne 10 listes différentes ($N = 10$), que nous modélisons un ensemble de paires d'entiers, et que la fonction de hachage soit tout simplement la somme des deux entiers de la paire, voilà comment les éléments suivants seraient stockés dans la table.

```
(11,5) (2,10) (17,13) (8,11) (3,0) (4,15) (6,4)

tab[ 0 ]= --> val=(17,13), h=30, suiv= --> val=(6,4), h=10 suiv= --> null
tab[ 1 ]= --> null
tab[ 2 ]= --> val=(2,10), h=12, suiv= --> null
tab[ 3 ]= --> val=(3,0), h=3, suiv= --> null
tab[ 4 ]= --> null
tab[ 5 ]= --> null
tab[ 6 ]= --> val=(11,5), h=16, suiv= --> null
tab[ 7 ]= --> null
tab[ 8 ]= --> null
tab[ 9 ]= --> val=(8,11), h=29, suiv= --> val=(4,15), h=9, suiv= --> null
```

On voit que le haché (pris modulo $N = 10$ ici) disperse les éléments dans les différentes listes. Ainsi, vérifier si un élément e appartient à la table est souvent rapide. On calcule son haché modulo N, i.e. $Hache(e)\%N$, ce qui nous donne le numéro de la liste où il faut chercher l'élément. Si les éléments sont bien dispersés, alors chaque liste a une longueur d'environ n/N, si n est le nombre d'éléments stockés. Du coup, si on choisit N à peu près de l'ordre de n, chaque liste est de longueur 1 environ, et donc la recherche est très rapide.

Concrètement, l'exemple précédent aurait été obtenu ainsi :

```
Table T;
Init( &T, 10 );
Element e[ 7 ] = { {11,5}, {2,10}, {17,13}, {8,11}, {3,0}, {4,15}, {6,4} };
for ( int i = 0; i < 7; ++i ) Insere( &T, e[ i ] );
for ( int i = 0; i < 7; ++i )
  if ( Appartient( &T, e[ i ] ) )
    printf( "Ok: (%d,%d) is in T.\n", e[ i ].first, e[ i ].second );
Element e1 = {18,4};
if ( ! Appartient( &T, e1 ) )
  printf( "Ok: (%d,%d) is not in T.\n", e1.first, e1.second );
```

Votre objectif est donc de créer les fonctions qui vont manipuler correctement une `Table`.

1. (/1,5) Ecrire la fonction `Init( Table* T, int N )` qui initialise cette structure comme si elle ne contenait aucun élément. N'oubliez pas d'utiliser l'allocation dynamique pour allouer le tableau de listes.

```
void Init( Table* T, int N )
{
  T->N   = N;
  T->tab = (Liste*) malloc( N*sizeof(Liste) );
  T->nb  = 0;
}
```

2. (/2,5) Ecrire la fonction `Appartient( Table* T, Element e )` qui teste si l'élément `e` appartient à l'ensemble `T`.

```
int Appartient( Table* T, Element e )
{
  int h          = Hache( e );
  int i          = h % T->N;
  Cellule* curr = T->tab[ i ];
  while ( curr != NULL )
    {
      if ( h == curr->h && Egal( e, curr->val ) ) return 1; // deja dedans
      curr = curr->suiv;
    }
  return 0;
}
```

3. (/2,5) Ecrire la fonction `Insere( Table* T, Element e )` qui réalise l'insertion dans la table `T`.

   NB : on peut utiliser la fonction précédente pour vérifier si l'élément appartient déjà à la table et on note aussi que l'ordre des éléments dans la liste n'a pas d'importance.

```
void Insere( Table* T, Element e )
{
  if ( Appartient( T, e ) ) return;
  int h          = Hache( e );
  int i          = h % T->N;
  Cellule* curr = T->tab[ i ];
  Cellule* cell = (Cellule*) malloc( sizeof( Cellule ) );
  cell->val     = e;
  cell->h       = h;
  cell->suiv    = curr;
  T->tab[ i ]   = cell;
  T->nb        += 1;
}
```

4. (/2) Ecrire la fonction `Supprime` qui enlève un élément et retourne le nombre d'éléments réellement supprimés (0 ou 1).

```
int Supprime( Table* T, Element e )
{
  int h          = Hache( e );
  int i          = h % T->N;
  Cellule* curr = T->tab[ i ];
  Cellule* prec = NULL;
  while ( curr != NULL )
    {
      if ( h == curr->h && Egal( e, curr->val ) )
        {
          Cellule* next = curr->suiv;
          free( curr );
          if ( prec != NULL ) prec->suiv = next;
          else T->tab[ i ] = next;
          return 1;
        }
      prec = curr;
      curr = curr->suiv;
    }
```

```
  return 0;
}
```

5. (/3) Lorsque le nombre d'éléments n se rapproche trop de N, on reconstruit un nouveau tableau de liste deux fois plus grand, puis on déplace les éléments dans ce nouveau tableau. Ecrivez donc la fonction `Agrandir( Table* T)` qui réalise ce travail.

```
void Agrandir( Table* T )
{
  Table   DT;
  Init( &DT, 2*T->N );
  for ( int n = 0; n < T->N; n++ )
    {
      Cellule* curr = T->tab[ n ];
      while ( curr != NULL )
        {
          Insere( &DT, curr->val );
          Cellule* next = curr->suiv;
          free( curr );
          curr = next;
        }
    }
  free( T->tab );
  T->tab = DT.tab;
  T->N   = DT.N;
}
```

6. (/1,5) On voudrait changer les fonctions de hachage et de tests d'égalité entre 2 élements. Comment mettre à jour la structure `Table` pour stocker une fonction de hachage et une fonction de test d'égalité ?

```
typedef int (*FctHachage)( Element e );
typedef int (*FctEgal)( Element e1, Element e2 );
struct STable {
  int    N;              // nombre de listes dans la table.
  Liste* tab;            // tableau de listes d'éléments.
  int    nb;             // nombre d'éléments stockés dans la table
  FctHachage Hache;
  FctEgal    Egal;
};
// On remplace les Egal par T->Egal et les Hache par T->Hache.
```