Informatique S3 **L2 CUPGE 2018–2019**

Contrôle continu

Durée : 1h30. Aucun document n'est autorisé.

Note de l'auteur : les solutions proposées sont complètes (de mon point de vue) et sont écrites pour être compréhensibles avec les rappels de cours qui vont avec. Il n'est pas nécessaire d'écrire toutes les étapes pour avoir les points, ni même d'avoir le bon résultat : juste savoir ou expliquer comment il aurait fallu faire est récompensé.

I Exercice : *Codage (4 points)*

Note de l'auteur : cet exercice était un peu trop long par rapport aux points (seulement 4). Pour l'année prochaine (au cas où cette correction circulerait), il y aurait probablement plus de points sur cette partie, avec le même type de questions.

On rappelle qu'un nombre :

$$x = (-1)^s \cdot 2^{e-127} \cdot (1 + \frac{M}{2^{23}})$$

est codé sous la forme :

(a) (2 pts) Donnez le code hexadécimal du signe s, de l'exposant e, et de la mantisse M, ainsi que le code entier sur 32 bits, du nombre $-10,5$.

On a :

$$\begin{aligned} -10,5 &= (-1)^1 \cdot 2^3 \cdot (1 + \frac{2,5}{8}) \\ &= (-1)^1 \cdot 2^{130-127} \cdot (1 + \frac{5}{16}) \\ &= (-1)^1 \cdot 2^{130-127} \cdot (1 + \frac{5}{2^4}) \\ &= (-1)^1 \cdot 2^{130-127} \cdot (1 + \frac{5 \cdot 2^{19}}{2^{23}}) \end{aligned}$$

Donc :

$$s = 1 \qquad e = 130 \qquad M = 5 \cdot 2^{19}$$

En base 16 (hexadécimal) : Donc :

$$s = 1 \qquad e = 128 + 2 = 82 \qquad M = 5 \cdot 8 \cdot 2^{16} = 280000$$

Pour écrire en binaire le code entier, on écrit tout en base 2 :

$$\overbrace{1}^{s}\ \overbrace{1000\ 0010}^{e}\ \overbrace{010\ 1000\ 0000\ 0000\ 0000\ 0000}^{M \text{ sur } 23 \text{ bits}}$$

$$\underbrace{1}_{1}\ \underbrace{1000}_{8}\ \underbrace{0010}_{2}\ \underbrace{010}_{2}\ \underbrace{1000}_{8}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}$$

et on regroupe par groupe de 4, qu'on traduit en hexadécimal :

$$\underbrace{1100}_{12=C}\ \underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{1000}_{8}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}$$

Donc le code entier en hexadécimal est $C1280000$.

(b) (2 pts) On utilise la méthode par multiplications successives (en base 16).

Pour avoir 10 chiffres en base 2, il en faut 3 en base 16 (et on aura une estimation trop précise ayant 12 chiffres en base 2) :

$$\begin{aligned} 0,23 \cdot 16 &= \boxed{3},68 \\ 0,68 \cdot 16 &= \boxed{10},88 \\ 0,88 \cdot 16 &= \boxed{14},08 \end{aligned}$$

Donc :

$$\begin{aligned} 1,23 &= (1,3AE)_{16} \\ &= (1,0011\,1010\,1110)_2 \\ &= \boxed{(1,0011\,1010\,11)_2} \qquad \text{avec 10 chiffres significatifs} \end{aligned}$$

II Exercice : *Test de compréhension (4 points)*

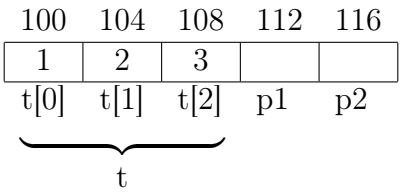
> Comme cela n'a pas été beaucoup vu en TP, on rappelle que si x est une valeur, alors :
>
> `( t ) x`
>
> est la traduction de cette valeur dans le type `t`.

On considère la fonction `main` suivante :

```
int main ( int argc , char * argv[] )
{
  int t[3] = {1,2,3} ;
  int * p1 = ( int * ) ( t + 1 ) ;
  int * p2 = ( int * ) ( & t + 1 ) - 1 ;
  * ( t + 2 ) = p1 [ 1 ] - t [ 0 ] ;
  p2[-2] = p1[1] ;
  printf ( "%d␣%d␣%d\n" , t[0] , t[1] , t[2] ) ;
  return 0 ;
}
```

(a) (2 points) Expliquez, en vous aidant si possible de schémas, quelles sont les valeurs, en tant qu'entier, de p_1 et p_2 en fonction de la valeur de t en tant qu'entier. Pour simplifier, on suppose que `sizeof ( int ) = 4`.

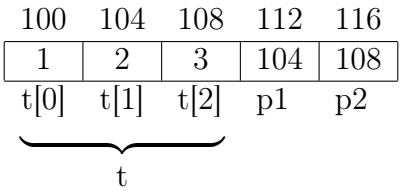
Il est nécessaire (et suffisant) de faire un schéma de la mémoire pour s'y retrouver ! Au début, on a :



Pour les additions et soustractions sur les adresses en tant qu'entier, si x est l'adresse d'une valeur de type `t`, alors en tant qu'entier, $x+1$ vaut `x + sizeof ( t )`.

1. `t` est l'adresse de `t[0]`, une valeur de type `int`, donc `t+1` vaut en tant qu'entier `t + sizeof ( int )`, soit `104` sur le schéma (c'est défini ainsi pour que `t+1` soit l'adresse de `t[1]`) ;
2. `& t` est l'adresse de `t`, une valeur qui est un tableau de 3 entiers, donc `& t+1` vaut en tant qu'entier `t + 3 * sizeof ( int )`, soit `112` sur le schéma. On traduit ensuite cette valeur en tant qu'adresse d'un entier, et on enlève 1, donc en tant qu'entier on obtient l'adresse `112 - sizeof ( int )=108`.

Donc après l'initialisation, on a :

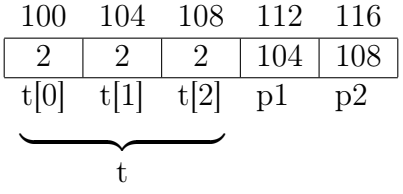


On passe ensuite à l'évaluation des 2 instructions :

— `t[2] = * ( p1 + 1 ) - t[0]`. D'après ce qui précède, `p1+1` est l'adresse de `t[2]`, donc on pourrait écrire de manière plus simple : `t[2] = t[2] - t[0]=2` ;

— `p2[-2] = t[2]`. D'après ce qui précède, `p2-2` est l'adresse d'une case contenant un entier 2 cases contenant des entiers avant `t[2]`, donc on pourrait écrire de manière plus simple : `t[0] = t[2] =2` ;

Donc avant le `printf`, le contenu de la mémoire est :



Le programme affiche `2 2 2`.

Note de l'auteur : On a fait ce genre de schéma en cours et en TP. Il est **très important** de bien les comprendre. On a dans cet exercice tout ce qui est difficile dans la programmation en C, le reste est simple.

III Exercice : *Somme des éléments d'un tableau (4 points)*

Note de l'auteur : après 2 exercices relativement difficiles, un exercice normalement simple. J'écris ces lignes avant d'avoir corriger les copies du CC1… Il y a en particulier un guidage maximum sur ce qu'il faut écrire, ce ne sera pas toujours le cas !

On veut écrire une fonction `somme_partielle` qui prend en entrée l'adresse de la première case d'une tableau d'entiers, le nombre n de cases de ce tableau, un entier i qui devrait être

l'indice d'une case dans le tableau, et l'adresse d'une variable (de type `int`) dans laquelle il faudra stocker :

$$\Sigma_{j=0}^{i} t[j]$$

c'est-à-dire la somme des éléments du tableau de l'indice 0 à l'indice i (inclus). Cette fonction renvoie 0 si le calcul est possible, et 1 sinon.

(a) (1 point) Donnez la déclaration de cette fonction.

```
int somme_partielle ( int * t , int n , int i , int * resultat ) ;
```

(b) (0,5 points) Décrivez avec une expression C les cas d'erreur. Il y a une erreur si i n'est pas l'adresse d'une case dans le tableau :

```
( i < 0 ) || ( i >= n )
```

(c) (2,5 points) Écrivez la fonction.

```
int somme_partielle ( int * t , int n , int i , int * resultat )
{
  int j , somme ;
  if ( ( i < 0 ) || ( i >= n ) )
    return 1 ;
  for ( j = 0 , somme = 0 ; j <= i ; j++ )
    somme = somme + t[j] ;
  * resultat = somme ;
  return 0 ;
}
```

IV Exercice : *Tas (8 points)*

Note de l'auteur : Il y a beaucoup plus à lire et comprendre qu'à écrire dans cet exercice ! Ça arrive souvent quand il faut écrire un programme.

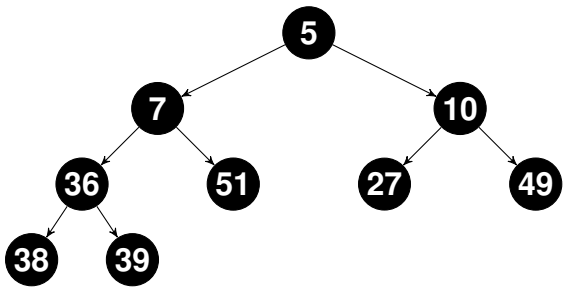
Préliminaires. Un *arbre* est soit un arbre vide, soit est un nœud (la *racine* de cet arbre) qui est le père de deux *fils*, son fils droit et son fils gauche, qui sont eux-mêmes des arbres.

Les *tas* sont des tableaux qui codent certains arbres de la manière suivante :

- la valeur de la racine de l'arbre est dans la case 0 ;
- si la valeur d'un nœud est stockée dans la case i, alors la valeur de la racine de son fils gauche est stockée dans la case $2 \times i + 1$, et la valeur de la racine de son fils droit est stockée dans la case $2 \times i + 2$;
- la valeur d'un nœud est toujours plus petite que la valeur de ses fils.

Pour savoir quand s'arrêtent les valeurs du tas (les cases blanches de la Figure 1), on a besoin, en plus du tableau, du nombre d'éléments dans le tas. Pour décrire des tas en C, on utilise les déclarations suivantes :

```
struct tas_s {
    int nb_elts; /* nombre d'éléments dans le tas */
    int * val; /* tableau des valeurs */
};
typedef struct tas_s * tas;
```

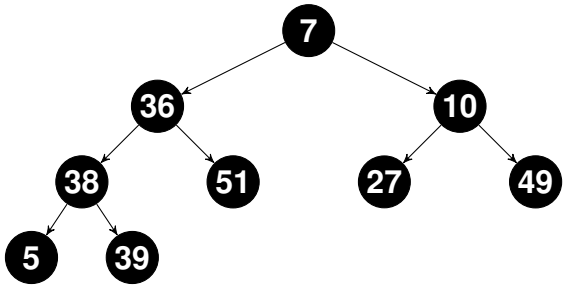


(a) Un tas, dont la racine a pour valeur 5, son fils gauche a une racine de valeur 7 et son fils droit a pour racine un nœud dont la valeur est 10. Les arbres vides ne sont pas représentés.

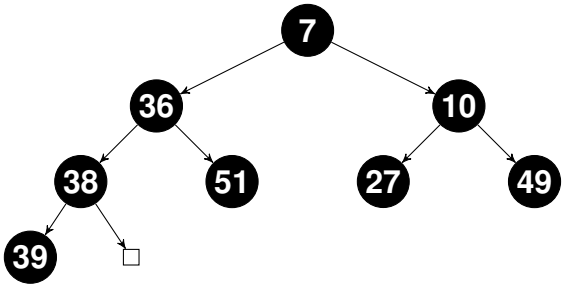
5	7	10	36	51	27	49	38	39			
0		2			5						

(b) Le tableau correspondant à ce tas : la valeur de la racine est dans la case 0, celle de son fils droit dans la case $2 = 2 \times 0 + 2$, et celle du fils gauche de ce fils droit est dans la case $5 = 2 \times 2 + 1$.

FIGURE 1 – Tas et leur représentation par un tableau



(a) Le tas de la Figure 1a après l'étape 2 de la suppression.



(b) Le tas de la Figure 1a après l'étape 5 de la suppression. Le nœud vide représente la décrémentation du nombre d'éléments. 39 est plus grand que 38, donc l'étape 6 ne fait rien

FIGURE 2 – Tas aux différentes étapes de la suppression.

(a) (0.5 + 0.5 + 1 = 2 pts) Écrire trois fonctions fils_gauche, fils_droit, et pere qui, en fonction d'un indice i, calculent l'indice correspondant respectivement au fils gauche, au fils droit, et au père du nœud dont la valeur est stockée dans la case i. *Par exemple, pere(5)=2, et fils_gauche(2)=5.*

Il faut tester des fonctions simples qui vont bien (on ne demande pas de justifier !) :

```
int fils_gauche ( int i )
{
  return 2 * i + 1 ;
}
int fils_droit ( int i )
{
  return 2 * i + 2 ;
}
int pere ( int i )
{
  // on utilise le fait que / est la division entiere
  return ( i - 1 ) / 2 ;
}
```

L'algorithme permettant d'insérer un élément dans un tas est donné informellement dans la Figure 3a.

1. On note i l'indice courant. Au départ, l'indice courant est l'indice de la première case vide dans le tableau (dans la Figure 1b, i vaut au départ 9) ;
2. mettre l'élément à ajouter dans la case d'indice i ;
3. Tant que :
   — l'indice courant est strictement positif ;
   — et la valeur du père de l'indice courant est plus grande que la valeur de l'indice courant

   échanger ces deux valeurs, et positionner l'indice courant sur son père ;
4. rendre le tas ;

(a) Algorithme d'insertion d'un élément.

1. On note i l'indice courant. Au départ, l'indice courant vaut 0 ;
2. Tant que le fils droit de l'indice courant est dans le tableau :
   — échanger la valeur du nœud courant avec celle du plus petit de ses fils ;
   — l'indice courant devient celui du fils avec lequel l'échange a été fait.
3. Si l'élément courant a un fils gauche, on échange ces deux éléments, et l'élément courant devient l'indice du fils gauche ;
4. on échange ensuite la valeur de l'élément courant et celle du dernier élément du tas ;
5. on décrémente de 1 le nombre d'éléments dans le tas ;
6. on remonte, comme pour l'insertion, l'élément courant tant qu'il est plus petit que son père.

(b) Algorithme de suppression du plus petit élément.

FIGURE 3 – Algorithmes sur les tas

(b) (1 pt) Justifiez (informellement) que l'insertion d'un élément dans un tas permet d'obtenir un tas, *i.e.*, que dans le tas obtenu, tout élément est plus petit que ses fils.

Il suffit de suggérer quelque chose qui ressemble à :
— Avant l'insertion, chaque fils est plus grand que son père ;
— Après l'insertion, chaque fils sauf celui inséré est plus grand que son père ;
— Quand le nœud inséré est strictement plus petit que son père, on les échange. La seule exception possible devient le père. Le nouveau père est strictement plus petit que ses deux fils.
— on remonte ainsi jusqu'à ce qu'il n'y ait plus d'exception (pas de père/racine, ou fils plus grand que son père

Le but n'était pas de faire une preuve, mais d'obliger à essayer de comprendre pourquoi l'algorithme marche.

(c) (1 pt) De quels arguments (signification de la variable et son type) est-ce qu'une fonction qui prend en entrée un tas aura besoin ?

Question faite pour guider.
— de l'adresse t de la première case du tableau, de type `int *` ;
— du nombre n d'éléments dans le tas (de type `int`).

(d) (2 pts) Écrire la fonction d'insertion d'un élément en C. On suppose qu'il y a strictement moins d'éléments dans le tas que le nombre maximal qu'il ne peut en contenir.

On a besoin du nombre n d'éléments dans le tas, de la valeur à ajouter, et de l'adresse t de

la première case du tas.

```
void insertion ( int * t , int i , int valeur )
{
  int tmp ;
  t[i] = valeur ;
  while ( ( i > 0 ) && ( t[i] < t[ pere(i) ] ) )
  {
    tmp = t[ pere(i) ] ;
    t[ pere(i) ] = t[i] ;
    t[i] = tmp ;
  }
}
```

(e) (2 pts) Écrire la fonction de suppression du plus petit élément en C en se basant sur l'algorithme de la figure 3b. On suppose qu'il y a au moins un élément dans le tas.

note : il serait exceptionnel que quelqu'un ait tout fait, y compris cette question. Un 20, ça se mérite ! Pour en revenir à l'exercice, on a besoin de l'adresse de la première case du tableau et de son n d'éléménts.

```
void suppression ( int * t , int n )
{
  int i ;
  int tmp ;
  i = 0 ;
  while ( fils_droit ( i ) < n ) // et donc fils_gauche(i) < n !
  {
    if ( t[ fils_droit ( i ) ] < t[ fils_gauche ( i ) ] )
    {
      tmp = t[fils_droit ( i )] ;
      t[i] = t[fils_droit ( i )] ;
      t[fils_droit ( i )] = tmp ;
      i = fils_droit ( i ) ;
    }
    else
    {
      tmp = t[fils_gauche ( i )] ;
      t[i] = t[fils_gauche ( i )] ;
      t[fils_gauche ( i )] = tmp ;
      i = fils_gauche ( i ) ;
    }
  }
  if ( fils_gauche ( i ) < n )
  {
    tmp = t[fils_gauche ( i )] ;
    t[i] = t[fils_gauche ( i )] ;
    t[fils_gauche ( i )] = tmp ;
    i = fils_gauche ( i ) ;
  }
  t[i] = t[n-1] ;
  while ( ( 0 < i ) && ( t[i] < t[pere(i)] ) )
```

```
    {
      tmp = t[pere(i)] ;
      t[pere(i)] = t[i] ;
      t[i] = tmp ;
      i = pere(i) ;
    }
}
```

Post-scriptum : on peut simplifier les 2 derniers exercices avec la fonction suivante :

```
void echange ( int * a , int * b )
{
  int tmp ;
  tmp = *a ;
  *a = *b ;
  *b = tmp ;
}
```

et utiliser, par exemple :

```
    echange ( t + i , t + pere ( i ) ) ;
```