Examen 2010

Durée : 2H

Documents autorisés: supports et notes manuscrites de cours et de TD/TME
– Barême indicatif –

Cette énoncé n'a jamais été proposée en l'état en examen. Il s'agit du rassemblement de plusieurs exercices donnés les années précédant 2009-2010 en Caml et traduits ici en langage C

Note importante : Les questions peuvent toutes être traitées indépendamment les unes des autres, au besoin en s'appuyant sur l'existence des fonctions des questions précédentes.

Exercice 1 (5 points) – Liste inverse

On considère une liste donnée L d'entiers codée par une liste simplement chaînée. Le but de cet exercice est de créer une liste *inverse*, c'est-à-dire une liste dont les éléments sont dans le sens inverse des éléments de L.

Q 1.1 Rappeler la déclaration du type `List` d'une liste simplement chaînée d'entiers et donner une fonction `insere_tete` qui permet d'insérer un nombre entier en tête d'une liste.

```c
typedef struct elnt{
  int i;
  struct elnt * suiv;
}Elnt;

typedef Elnt* List;

void insere_tete(List *L, int i){
  Elnt * nouv=(Elnt*) malloc(sizeof(Elnt*));

  nouv->i=i;
  nouv->suiv=*L;
  *L=nouv;
}
```

Q 1.2 Donner une fonction itérative `inverse_ite` qui prend en paramètre deux listes $L1$ et $L2$ de façon à ce qu'après exécution, $L2$ soit l'inverse de $L1$

```c
void inverse_ite(List L1, List *L2){
  Elnt * cour;
  cour=L1;
  while (cour!=NULL){
    insere_tete(L2,cour->i);
    cour=cour->suiv;
  }
}
```

Q 1.3 Même question mais pour une fonction récursive `inverse_rec`.

```
void inverse_rec(List L1, List *L2){

  if (L1!=NULL){
    insere_tete(L2,L1->i);
    inverse_rec(L1->suiv,L2);
  }
}
```

Q 1.4 Donner (sans preuve) la complexité des deux fonctions itératives et récursives des questions précédentes en fonction du nombre n d'élements de la liste $L1$. Indiquer l'occupation mémoire des deux fonctions (sans compter la taille de $L1$ et $L2$). Dans quel cas, peut-on réduire cette occupation mémoire ?

Ces deux fonctions sont en $\Theta(n)$. `inverse_ite` utilise uniquement les pointeurs $L1$, $L2$ et *cour*. La fonction `inverse_ite` utilise par contre 2 pointeurs $L1$ et $L2$ par appel, soit $2n$ pointeurs qui sont maintenus en activité tout au long de l'exécution. En fait, il s'agit d'une fonction où l'appel est en dernière ligne. Il est donc possible d'utiliser un compilateur reconnaissant la récursivité terminale qui détruit la mémoire conservée inutilement : ainsi en permanence, il n'existerait que 2 pointeurs en mémoire. Les deux fonctions sont alors équivalentes.

Exercice 2 (3 points) – Mise en oeuvre du cours

Parmi les structures de données que vous avez rencontrées en cours, laquelle vous paraît la plus appropriée pour les programmes ci-dessous. Vous justifierez bien votre réponse.

Q 2.1 Un système d'exploitation X gère l'exécution d'autres programmes. Au moment où l'on veut exécuter un programme, les attributs concernant ce dernier sont stockés dans la structure de données. Les temps d'exécution des programmes sont assez petits et on exécute souvent des programmes. À la fin de l'exécution d'un programme, les attributs de ce dernier sont détruits. Ceci implique que l'état de la structure de données change très souvent. Les attributs peuvent être consultés par l'utilisateur mais c'est une opération assez rarement réalisée.

De tout ce que l'on a vu jusqu'à maintenant, les listes sont les plus adaptées car elles permettent des insertions et suppressions rapides. Elles ont l'inconvénient d'avoir un temps d'accès aux éléments un peu conséquent, mais puisque l'accès aux éléments est rare, ce n'est pas dirimant.

Q 2.2 Un logiciel qui effectue des calculs matriciels. Les matrices sont déclarées au début du programme. Les matrices sont non creuses et de petites tailles. Le programme doit s'exécuter le plus rapidement possible.

Les matrices manipulées sont définies une fois pour toutes en début de programme. Donc utiliser une liste n'offrirait que peu d'intérêt puisqu'on n'a pas besoin de faire des ajouts/suppression. En revanche, utiliser des tableaux est une bonne solution car cela permet d'accéder immédiatement aux éléments, et donc d'obtenir un programme rapide.

Q 2.3 Idem mais les matrices sont creuses et sont de grandes tailles.

Si les matrices sont de grandes tailles, il peut être impossible de les stocker en extension sous forme de tableau. Il peut être plus astucieux de ne stocker que les termes non nuls, et donc d'utiliser un enregistrement ou un triplet (i,j,val) indiquant que la valeur de la ième ligne, jème colonne est val. Les collections de triplets ou enregistrements peuvent être elles-mêmes stockées dans des tableaux ou dans des listes.

Exercice 3 (4 points) – Arbres ternaires

Une liste chaînée est une structure de données permettant de stocker une collection d'éléments, et dans laquelle on peut accéder directement au premier élément (42 sur la figure 1.a), puis de cet élément, on accède au suivant (43), et ainsi de suite. De la même manière, les arbres ternaires sont des structures de données permettant de stocker des collections d'éléments. Elles sont représentables ¡¡ logiquement ¿¿ par des arbres renversés comme le montre la figure 1.b : l'élément en haut de la figure (23), qu'on appelle la racine, est accessible directement. Les éléments suivants —ceux juste en dessous en suivant les flèches partant de 23— sont accessibles à partir de la racine, etc. Ainsi, pour accéder à 15, faut-il partir de 23, aller à l'élément 30, puis à l'élément 10, et enfin arriver à l'élément 15.

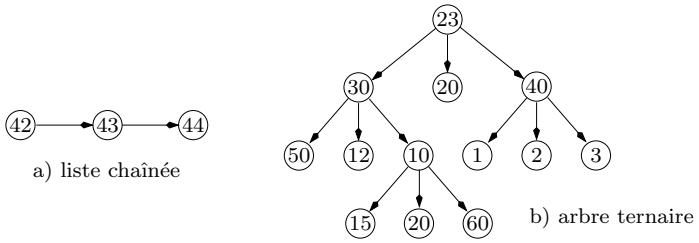
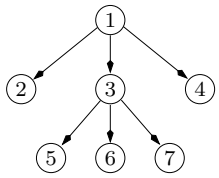


FIG. 1 – Une liste chaînée (à gauche) et un arbre ternaire (à droite)

Q 3.1 Donnez une définition d'un type pour représenter les arbres ternaires ainsi qu'une fonction allouant et retournant un pointeur sur un noeud de l'arbre.

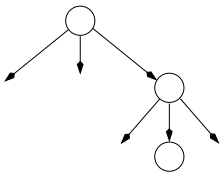
```
typedef struct noeud{
  int elt;
  struct noeud *f1;
  struct noeud *f2;
  struct noeud *f3;
}Noeud;

typedef Noeud * ArbreTernaire;

Noeud * alloue_noeud(int i){

 Noeud * nd=(ArbreTernaire) malloc(sizeof(Noeud));
 nd->elt=i;
 nd->f1=NULL;
 nd->f2=NULL;
 nd->f3=NULL;;
}
```

Q 3.2 En utilisant le type de la question précédente, définissez l'arbre ci-dessous dans une fonction main :




```c
int main(){
  ArbreTernaire T;
  T= alloue_noeud(1);
  T->f1=alloue_noeud(2);
  T->f1=alloue_noeud(3);
  T->f3=alloue_noeud(4);
  T->f2->f1=alloue_noeud(5);
  T->f2->f2=alloue_noeud(6);
  T->f2->f3=alloue_noeud(7);
  return 0;
}
```

Q 3.3 Écrivez une fonction qui renvoie vrai si et seulement si l'arbre passé en paramètre a exactement la forme suivante :



```c
int test_structure(ArbreTernaire T){

  if (T!=NULL){
    if ((T->f1==NULL)&&(T->f2==NULL)&&(T->f3!=NULL)){
      if ((T->f3->f1==NULL)&&(T->f3->f2!=NULL)&&(T->f3->f3==NULL)){
   if ((T->f3->f2->f1==NULL)&&(T->f3->f2->f2==NULL)&&(T->f3->f2->f3==NULL)){
      return 1==1;
    }
       }
     }
  }
  return 1==2;
}
```

Exercice 4 (5 points) – Transfert de structures

Soit le type :

```c
typedef struct {
   float x;
   int y;
}enreg;
```

Q 4.1 Définissez le type `List` d'une liste simplement chaînée de pointeurs sur des éléments `enreg`.

```c
typedef struct cellule{
   enreg * elnt;
   struct cellule * suiv;
}Cellule;

typedef Cellule* List;
```

Q 4.2 Écrivez une fonction qui, à partir d'une liste de pointeurs sur `enreg`, renvoie par, valeur-retour de la fonction, un tableau contenant les mêmes éléments pointés par la liste (sans donc copier les éléménts en double en mémoire). Ne pas utiliser la commande C `realloc`.

Une manière de résoudre le problème consiste à calculer la longueur de la liste, à créer un tableau de cette longueur, puis à remplir ce tableau, autrement dit :

```c
enreg** transfert(List L){
   enreg ** T;
   int i=0;
   Cellule * cour=L;

   while (cour!=NULL){
     i++;
     cour=cour->suiv;
   }

   T=(enreg**) malloc(i*sizeof(Cellule*));

   cour=L;
   while (cour!=NULL){
     T[i]=cour->elnt;
     cour=cour->suiv;
   }

   return T;
}
```

Exercice 5 (8 points) – Modélisation de l'allocation

On considère la mémoire d'un ordinateur comme un tableau M de 1000000 d'octets (ou de caractères si vous préférez), comme le montre la figure 2 (les nombres au dessus du tableau représentent les indices des cases mémoire ou octets). Lorsqu'un programme s'exécute, il utilise une portion de la mémoire pour ses besoins personnels. Celle-ci n'est alors accessible que de lui et pas des autres programmes[1]. Pour acquérir une portion de mémoire contiguë dans M, c'est-à-dire l'ensemble des octets de M entre des indices i et j, en C, on utilise la fonction `malloc` à laquelle on passe en paramètre le nombre d'octets dont on a besoin, et qui retourne l'indice i de M à partir duquel commence la portion de mémoire contiguë acquise. i est donc en fait l'adresse de la zone mémoire allouée. Par exemple, si `malloc(6)` renvoie la valeur 3, alors la zone mémoire acquise est celle des octets d'indices 3 à 8 (la partie grisée sur la gauche de la figure 2).
Au démarrage de l'ordinateur, aucune zone n'est acquise, autrement dit, la mémoire libre va des indices 0 à 999999. Lorsqu'une zone est acquise par un programme, elle ne peut plus l'être par un autre. Pour

[1] C'est bien entendu une version simplifiée de la réalité.

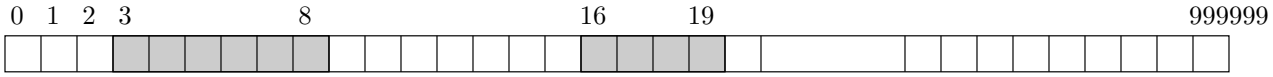


FIG. 2 – Représentation de la mémoire d'un ordinateur.

éviter cela, `malloc` conserve l'information de début et de fin des zones acquises. Pour cela, on utilisera ici des enregistrements de type :

```
1 typedef struct zone {
2    int debut;
3    int fin;
4    struct zone * suiv;
5 } Zone;
```

et on supposera que l'on dispose dans la fonction `main` d'un programme une liste triée (simplement chaînée) `Lzone` contenant les informations sur les zones acquises. La liste est triée sur les indices `deb` de début de zone. Sur l'exemple de la figure 2, on aurait donc une liste contenant le couple (3,8) chaîné au couple (16,19). Ces zones étant acquises, on ne pourra plus affecter une nouvelle zone à un programme contenant par exemple l'octet d'indice 8.

Q 5.1 Écrivez une fonction `int libre(int n, int deb, Zone* z)` qui renvoie vrai si et seulement si il existe au moins `n` octets entre la case d'indice `deb` et le début de la zone pointée par `z`. Par exemple, sur la figure ci-dessus, `libre(7,9,z1)` où `z1` est un pointeur sur la zone (16,19) renverra `vrai` alors que `libre(7,10,z1)` renverra `faux`.

Q 5.2 Écrivez une fonction `reserve(int n, Zone** Lzone)` telle que, si `Lzone` contient une liste des zones mémoire allouées, alors `(reserve(n, Lzone)` essaye de trouver un emplacement contigu de `n` octets libres. La fonction renvoie alors l'emplacement où les `n` octets ont été alloués et met à jour la variable `Lzone`. Dans le cas contraire, elle renvoie `-1`. Par exemple, `reserve(4,0, Lzone)` pourrait renvoyer 9 (car il y a 4 octets de libre à partir de la case 9) et modifie la liste chaînée `Lzone` en (3,8), (9,12), (16,19).

> Si la liste est non vide, il faut déterminer si l'on a suffisamment de place pour réserver n octets entre 0 et le début de la liste. Le cas échéant, on réserve ces n octets à partir de 0. Sinon, on continue notre recherche d'un espace libre après la fin de la première zone. Lorsque l'on a parcouru toute la liste, soit on peut caser n octets entre 0 et la fin de la mémoire, auquel cas on réserve un emplacement, soit on n'a pu trouver de place dans la mémoire et on renvoie -1.

Q 5.3 Lorsqu'une zone a été acquise, on peut la rendre à nouveau libre grâce à la fonction `int free(int deb)` qui recherche dans la liste `Lzone` la zone débutant précisément à l'indice `deb`. Si elle trouve cette zone, elle la supprime de la liste, sinon elle retourne -1. Écrire une définition de la fonction `free`.