Durée 3 h Documents non autorisés

Sujet d'examen 1

Pratique du C

Décembre 2013

Introduction

Écrivez lisiblement et n'hésitez pas à commenter votre code en langage C. Vous ne pouvez utiliser que les fonctions C dont le prototype est donné dans l'énoncé et celles dont vous donnez la définition dans vos copies.

Les sections sont indépendantes ; lisez l'énoncé complet avant de commencer à le résoudre.

1 Quizz

1. Expliquez la différence entre les deux déclarations suivantes :

   ```
   int *a()   ;
   int (*a)() ;
   ```

   Correction. La déclaration :

   ```
   int * a () ;
   ```

   est celle d'une fonction ne prenant pas de paramètre et retournant un pointeur sur un entier.

   La déclaration :

   ```
   int (*a) () ;
   ```

   est celle d'un pointeur de fonction pointant sur des fonctions ne prenant pas de paramètre et retournant un entier.

2. On considère une chaîne de caractères représentant une expression arithmétique du type :

   `"(x+1)*(x+2)*(y+1)+2".`

   On désire donner la définition d'une fonction qui vérifie que l'expression est valide du simple point de vue du parenthésage. Il faut donc vérifier que le nombre de parenthèses ouvrantes est égale à celui des parenthèses fermantes et qu'il n'y a, à aucun endroit de la chaîne, plus de parenthèses fermantes qu'ouvrantes.

   Par exemple, `"(((x+1))"` est incorrecte car il y a 3 parenthèses ouvrantes et seulement 2 fermentes. L'expression `"(x+1))+((y+2)"` est incorrecte également car il y a 2 fermetures de parenthèses (après le `x+1`) alors qu'une seule est ouverte.

   Donnez la définition de la fonction de prototype `int estValide(char ch[MAX])` qui détermine si le parenthésage de `ch` est correct (on renvoie 1 si oui et 0 sinon).

   Notez qu'on ne vérifie que le parenthésage, la fonction renverra donc 1 pour les expressions du type `"(x++-3)*+y"` ou `"()()x"`.

2 Tri utilisant un tas binaire

L'objectif de l'exercice est d'implanter le tri d'un tableau d'entier en utilisant un *tas binaire*. (On souhaite trier le tableau afin d'avoir le plus petit élément au début et le plus grand à la fin).

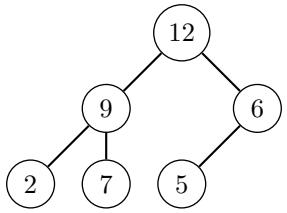
Tas binaire

Définition 1 *Un* tas binaire *est un arbre binaire complet — i.e. les nœuds de l'arbre peuvent être stockés de façon contiguë dans un tableau — ordonné en tas — les nœuds sont ordonnés par leurs clefs et les clefs des fils sont inférieures à celles des pères.*

Représentation d'un tas binaire par un tableau

Comme tout arbre binaire, un tas binaire peut être représenté dans un tableau unidimensionnel indicés à partir de 0 : le père d'un nœud en position i a pour enfants un fils gauche en position $2i + 1$ et un fils droit en position $2(i + 1)$.

Exemple. L'arbre



est codé par le tableau

12	9	6	2	7	5

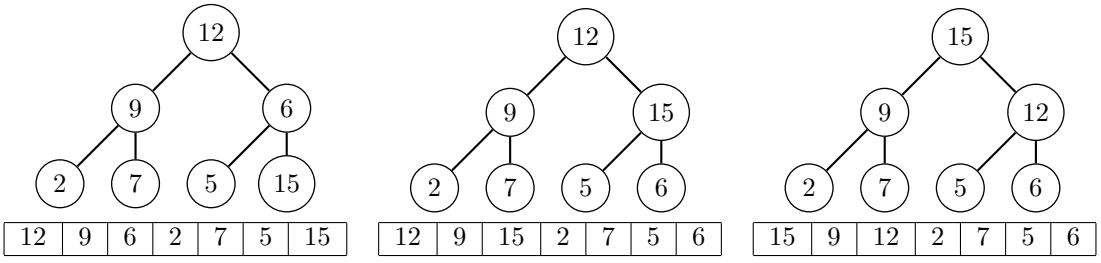
Les tas binaires sont utilisés pour implanter les files de priorités car ils permettent des insertions en temps logarithmiques et un accès direct au plus grand élément.

Insertion dans un tas binaire

L'insertion d'un élément dans un tas binaire se ramène à 2 type d'opérations :

1. l'insertion de l'élément dans première cellule vide du tableau codant l'arbre.
2. la *percolation* de cet élément depuis une feuille de l'arbre jusqu'à la racine (si nécessaire). Ces opérations consistent à échanger autant que nécessaire l'élément qui *percole* avec son père *courant* si ce dernier est plus petit que lui.

Dans l'exemple suivant, on ajoute l'élément 15 au tas en 3 étapes :

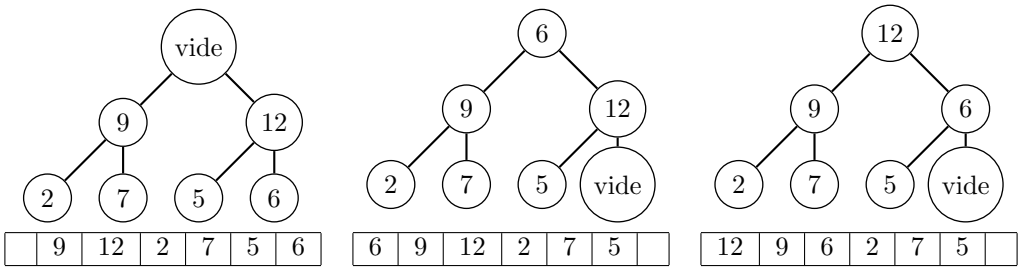


Retirer un élément d'un tas binaire

Pour retirer un élément d'un tas binaire on utilise 3 type d'opérations :

1. on retire la racine de l'arbre (i.e. le premier élément du tableau codant cet arbre) laissant ainsi la première cellule vide et deux sous-arbres.
2. on place la dernière feuille (i.e. le dernier élément du tableau codant l'arbre) à la racine (i.e. le premier élément du tableau codant l'arbre)
3. le *tamissage* de cet élément depuis la racine jusqu'à une feuille (si nécessaire) ; c'est en quelque sorte l'opération inverse de la percolation. Ces opérations consistent à échanger autant que nécessaire l'élément que l'on a placé dans la racine avec le plus grand fils *courant* qui lui est inférieur.

Dans l'exemple suivant, on retire l'élément 15 au tas que l'on a construit ci-dessus :



2.1 Présupposés

On suppose disposer de deux pointeurs

```
int *DebutTableau ;
int *FinTableau ;
```

le premier pointant au début d'un tableau d'entier et le second sur sa fin. On ne souhaite pas utiliser d'autre espace mémoire que ce tableau (notre tri est donc destructif). De plus, les tas binaires intermédiaires utilisent donc ce tableau comme espace de stockage.

Questions :

1. Donnez la définition de la fonction de prototype :

   ```
   void permuter(int *, int *) ;
   ```

   qui prend en argument 2 pointeurs d'entiers et qui permute les valeurs sur lesquelles ils pointent.

2. Donnez la définition de la fonction de prototype :

   ```
   void percolation(int *, int *) ;
   ```

   qui prend en premier paramètre un pointeur sur la cellule d'un tableau codant la racine du tas considéré et en second paramètre un pointeur sur la cellule où l'on suppose avoir déjà placé l'entier à insérer dans le tas (cette cellule correspond à la dernière feuille du tas que l'on obtient après percolation). Cette fonction implante les actions de percolation décrites ci-dessus.

3. Donnez la définition de la fonction de prototype :

   ```
   void ConstruireTas(int *, int *) ;
   ```

   qui prend en premier paramètre un pointeur sur la première cellule d'un tableau contenant les entiers à trier et en second paramètre un pointeur sur la dernière cellule de ce tableau. Cette fonction utilise la fonction `percolation` ci-dessus pour construire un tas binaire stocké au final dans ce tableau.

4. Donnez la définition de la fonction de prototype :

   ```
   void Trier(int *, int *) ;
   ```

   qui prend en paramètres un pointeur sur la première cellule d'un tableau contenant les entiers à trier et un pointeur sur la dernière cellule de ce tableau. Cette fonction
   – utilise la fonction `ConstruireTas` pour construire un tas binaire à partir des entiers en utilisant l'espace mémoire du tableau ;
   – place un pointeur `ptr` sur le dernier élément du tableau ;
   1) interverti le premier élément avec l'élément pointé ;
   2) utilise la fonction `tamissage` (cf. ci-dessous) pour reconstituer la structure de tas binaire du tableau considéré sans les éléments au-delà de `ptr` ;
   3) recule le pointeur `ptr` et recommence à l'étape 1 jusqu'à ce que le tableau soit trié.

5. Donnez la définition de la fonction de prototype :

   ```
   void tamissage(int *, int *) ;
   ```

   qui prend en premier paramètre un pointeur sur la cellule correspondant à la racine du tas que l'on veut construire — cette racine est supposée déjà contenir l'élément que l'on va tamiser — et en second paramètre un pointeur sur la première cellule du tableau d'entiers qui ne fait pas partie du codage du tas que l'on veut obtenir. Cette fonction implante les actions de tamissage décrites plus haut.

Pour donner un corrigé, nous pouvons coder les items suivants :

⟨ *** ⟩≡
  ⟨*directives au préprocesseur*⟩
  ⟨*fonctions d'entrée sortie*⟩
  ⟨*fonctions définies dans l'examen*⟩
  ⟨*fonction principale*⟩

Le nombre d'entiers à trier n est codé par une macro :

⟨*directives au préprocesseur*⟩≡
```
/* #define NBEntier 10 */
```

Afin de faciliter d'éventuels tests, nous allons donner la définition d'une fonction qui construit un tableau de n entiers et d'une fonction d'affichage de ce tableau :

⟨*fonctions d'entrée sortie*⟩≡
  ⟨*fonction d'entrée*⟩
  ⟨*fonction de sortie*⟩

Pour faire simple, on convient que le premier entier transmis correspond aux nombres d'entiers constituant l'ensemble à trier.

La fonction d'entrée va donc récupérer cet entier, allouer un espace mémoire permettant de stocker les entiers puis les récupérer :

⟨*fonction d'entrée*⟩≡
```
int *
loadintegers
(unsigned int n)
{
```
  ⟨*variables automatiques de loadintegers*⟩
  ⟨*allocation dynamique*⟩
  ⟨*chargement*⟩
  ⟨*valeur retournée par loadintegers*⟩
```
}
```

Le paramètre de notre fonction est par convention le nombre d'entiers que nous allons trier et on peut donc maintenant réserver l'espace mémoire correspondant sur le tas.

⟨*allocation dynamique*⟩≡
```
res = (int *) malloc(sizeof(int)*n) ;
```

Bien sur, il nous faut déclarer un pointeur à cet effet :

⟨*variables automatiques de loadintegers*⟩≡
```
int * res ;
```

et inclure le fichier d'entête donnant par exemple le prototype de la fonction d'allocation.

⟨*directives au préprocesseur*⟩+≡
```
#include <stdlib.h>
```

Reste à obtenir les entiers en questions. Pour faire simple, on les récupère depuis le fichier `/dev/random` qui nous fournit un générateur d'entiers aléatoire :

⟨*chargement*⟩≡
```
if(!(rand = fopen("/dev/random","r")))
   exit(1) ;
fread(res,sizeof(int),n,rand) ;
```

Ces opérations nécessitent la variable automatique :

⟨*variables automatiques de loadintegers*⟩+≡
```
FILE * rand ;
```

De plus, il nous faut inclure le fichier d'entête

⟨*directives au préprocesseur*⟩+≡
```
#include <stdio.h>
```

Pour finir, il nous reste à fermer le fichier ouvert et à retourner le pointeur sur la zone mémoire que nous venons de remplir :

⟨*valeur retournée par loadintegers*⟩≡
```
fclose(rand) ;
return res ;
```

La fonction d'affichage est bien plus simple :

⟨*fonction de sortie*⟩≡
```
void
printintegers
(int *ptr,int size)
{
 int i ;
 for(i=0;i<size;i++)
   printf("%d \n",ptr[i]) ;
 printf("\n\n");
}
```

Les fonctions demandées sont :

⟨*fonctions définies dans l'examen*⟩≡
⟨*permuter*⟩
⟨*percolation*⟩
⟨*ConstruireTas*⟩
⟨*tamissage*⟩
⟨*Trier*⟩

1. La fonction de permutation est un grand classique :

⟨*permuter*⟩≡
```
void
permuter
(int *a, int *b)
{
  int stck ;
  stck = *a ;
  *a = *b ;
  *b = stck ;
  return ;
}
```

2. Pour faire simple, on peut utiliser une implantation récursive :

⟨*percolation*⟩≡
```
void
percolation
(int *head, int *tail)
{
⟨variables automatiques de la fonction percolation⟩
⟨conditions d'arrêt de la fonction percolation⟩
⟨permutation⟩
⟨appel récursif⟩
}
```

Notre récursivité doit s'arrêter si l'élément courant est sur la racine du tas :

⟨*conditions d'arrêt de la fonction percolation*⟩≡
```
if (tail==head)
   return ;
```

Si ce n'est pas le cas, nous devons déterminer le père de l'élément courant et pour ce faire connaître l'indice de l'élément courant dans le tableau. On commence à calculer l'indice dans le tableau de l'élément courant et son père probable :

⟨*conditions d'arrêt de la fonction percolation*⟩+≡
```
pere = (tail-head)/2 ;
```

Suivant que cet indice est pair ou impair, le calcul de l'indice du père est légérement différent :

⟨*conditions d'arrêt de la fonction percolation*⟩+≡
```
if(pere && !((tail-head) % 2))
   pere -- ;
```

et pour ce faire définir la variable automatique `index`

⟨*variables automatiques de la fonction percolation*⟩≡
```
unsigned int pere ;
```

Avec cette définition, il ne faut plus rien faire si le père de l'élément courant est plus grand ou égal que cet élément :

⟨*conditions d'arrêt de la fonction percolation*⟩+≡
```
if(head[pere]>= *tail)
   return ;
```

Dans le cas contraire, il faut intervertir père et fils :

⟨*permutation*⟩≡
```
   permuter(head+pere,tail) ;
```

On peut maintenant positionner l'élément courant sur ce nouveau père et recommencer la percolation :

⟨*appel récursif*⟩≡
```
    percolation(head,head+pere) ;
```

3. Avec la fonction de percolation, on peut construire un tas à partir des entiers considérés.

⟨*ConstruireTas*⟩≡
```
void
ConstruireTas
(int *head, int *tail)
{
 int *tmp = head ;
 while(tmp<=tail)
  percolation(head,tmp++) ;
}
```

4. Le tamissage est l'opération inverse de la percolation et donc, on peut la coder de manière similaire. Pour changer, on va coder sans récursion :

⟨*tamissage*⟩≡
```
void
tamissage
(int * head, int * tail)
{
```
⟨*variables automatiques de la fonction de tamissage*⟩
⟨*corps de la fonction de tamissage*⟩
```
return ;
}
```

On va définir un entier pointant sur l'élément que l'on tamise :

⟨*variables automatiques de la fonction de tamissage*⟩≡
```
int current ;
```

Au début, l'élément est la racine :

⟨*corps de la fonction de tamissage*⟩≡
```
current = 0 ;
```

Notre tâche se résume à une boucle :

⟨*corps de la fonction de tamissage*⟩+≡
```
while(head+2*current+1<tail)
{
```
⟨*chercher avec qui permuter*⟩
⟨*permuter ou sortir*⟩
⟨*réactualiser la position de l'élément que l'on tamise*⟩
```
}
```

On cherche le maximun des clefs des nœuds fils supérieur au père :

⟨*chercher avec qui permuter*⟩≡

```
max=0 ;
if (head[2*current+1]>head[current])
   max = 2*current+1 ;
if( head+2*(current+1)<tail)
{
if (max)
{
   if(head[2*(current+1)]>head[2*current+1])
      max = 2*(current+1) ;
}
else
{
   if (head[2*(current+1)]>head[current])
      max = 2*(current+1) ;
}
}
```

L'implantation que nous donnons de cette opération nécessite la variable automatique :

⟨*variables automatiques de la fonction de tamissage*⟩+≡

```
int  max ;
```

Si le père est plus petit que ces fils, on arrête et sinon, le pointeur `max` pointant sur le fils à permuter avec le père, il nous suffit de faire une permutation :

⟨*permuter ou sortir*⟩≡

```
    if (!max)
      return ;
    else
    permuter(head+current,head+max) ;
```

Pour finir, il nous suffit maintenant de positionner `current` sur la nouvelle cellule contenant l'élément tamisé :

⟨*réactualiser la position de l'élément que l'on tamise*⟩≡

```
    current = max ;
```

5. Pour conclure, on peut implanter la fonction de tri en suivant les étapes indiqués dans l'énoncé :

⟨*Trier*⟩≡

```
void
Trier
(int *head,int *tail)
{
  int n ;
  ConstruireTas(head,tail) ;
  n = NBEntier ;
  while(tail>head)
  {
    permuter(head,tail) ;
    tamissage(head,tail) ;
    tail-- ;
  }
  return ;
}
```

⟨*fonction principale*⟩≡

```
int
main
(int argc, char **argv)
{
  int *head = loadintegers(NBEntier);
  printf("L'ensemble de depart\n");
  printintegers(head,NBEntier);
  Trier(head,head+NBEntier-1);
  printf("L'ensemble trie\n");
  printintegers(head,NBEntier);
  return 0;
}
```

3 Algorithme 196

On se donne le prétexte de l'algorithme 196 pour manipuler des listes doublement chaînées. Chaque cellule de ces dernières contient un chiffre décimal (compris entre 0 et 9), sa position dans le nombre considéré ainsi que deux pointeurs `next` et `previous` sur des cellules (pointant respectivement sur la cellule suivante et la cellule précédente).

On utilise le synomyme de type suivant :

```
typedef struct cell_m *liste;
```

Vous pouvez utiliser les fonctions `malloc`, `free` et `printf` de la librairie standard.

1. Donner le modèle de structure d'identificateur `cell_m` codant une cellule.

   Correction.

   ```
   typedef struct cell_m *liste ;
   struct cell_m
   {
           int val;
           int pos;
           liste previous;
           liste next ;
   } ;
   ```

2. Donner la définition d'une fonction de prototype :

   ```
   void inserer (liste *,int,int) ;
   ```

   qui permet d'allouer une cellule et de l'insérer dans la liste chaînée en position de tête (son pointeur `next` pointe sur la suite de la liste et son pointeur `previous` pointe sur `NULL`). Veillez à bien affecter les deux champs entiers de la liste et à vous assurer que la liste reste bien doublement chaînée.

   Correction.

   ```
   void
   inserer
   (liste *l,int pos,int val)
   {
      liste tmp ;
      tmp = (liste) malloc(sizeof(struct cell_m));
      tmp->previous = NULL ;
      tmp->next = *l ;
      tmp->pos = pos ;
      tmp->val = val ;
      if(tmp->next) tmp->next->previous = tmp ;
      *l = tmp ;
      return ;
   }
   ```

3. Donner la définition de la fonction de prototype

   ```
   liste int2liste(unsigned int);
   ```

   qui prend en entrée un entier non signé et qui retourne une liste dont les cellules contiennent les chiffres de l'entier ainsi que leurs positions. Par exemple, pour l'entier 2013 :
   – le premier chainon contient le chiffre 2 et la position 4 ;
   – le second chainon contient le chiffre 0 et la position 3 ;
   – le troisième chainon contient le chiffre 1 et la position 2 ;

– le dernier chainon contient le chiffre 3 et la position 1.
Le pointeur `next` du dernier chainon pointe vers `NULL`. Le pointeur `previous` du premier chainon pointe vers `NULL`. De plus, la fonction `int2liste` retourne la liste ainsi créée.

Correction.

```
liste
int2liste
(unsigned int nb)
{
        liste res,tmp ;
        int d,pos ;
        res = NULL ;
        pos = 1 ;

        while(nb)
        {
         d = nb%10 ;
         tmp = (liste) malloc(sizeof(struct cell_m));
         tmp->previous = NULL ;
         tmp->next = res ;
         tmp->pos = pos ;
         tmp->val = d ;
         pos++;
         if(tmp->next)
                tmp->next->previous = tmp ;
         res = tmp ;
         nb /=10 ;
        }

        return res ;
}
```

4. Donner la définition d'une fonction de prototype :

   `void printint(liste);`

   qui affiche l'entier représenté par la liste passée en paramètre.

   Correction.

```
void
printliste
(liste l)
{
        while(l)
        {
        printf("%d",l->val);
        l= l->next ;
        }
        return ;
}
```

5. Donner la définition d'une fonction de prototype :

   `void freeliste(liste *);`

   qui détruit une liste.

   Correction.

```
void
freeliste
(liste *l)
{
   if (*l==NULL)
        return ;
   freeliste(&((*l)->next)) ;
   free(*l);
   *l=NULL;
}
```

6. Donner la définition d'une fonction de prototype :

   `int palindrome(liste);`

   qui retourne 1 si l'écriture décimale de l'entier représenté par la liste passé en paramètre est un palindrome et 0 sinon.

   Un palindrome (substantif masculin), du grec πάλιν (en arrière) et δρόμος (course) est un texte ou un mot dont l'ordre des symboles (lettres, chiffres, etc.) reste le même qu'on le lise de gauche à droite ou de droite à gauche comme dans l'expression "Esope reste ici et se repose". Ainsi, en base 10, l'entier 121 est un palindrome.

   Correction.

```
int
palindrome
(liste l)
{
   liste tete ;
   if(l==NULL)
        return 1 ;
   tete = l ;
   /* on va se servir de l comme queue */
   while(l->next!=NULL)
     l = l->next ;

   while(1)
   {
        if(tete->val!=l->val)
            return 0;
        if(tete->pos<=l->pos)
            return 1 ;
        tete = tete->next ;
        l = l -> previous ;
   }

   return 1 ;
}
```

7. Donner la définition d'une fonction de prototype :

   `liste InverseEtAdditionne(liste) ;`

   qui prend en argument une liste représentant un entier non signé et retourne une liste représentant un entier non signé somme de ce nombre et de son écriture décimale inverse obtenu en inversant l'ordre des chiffres (par exemple $56 + 65 = 121$, $125 + 521 = 646$).

   Correction.

```
liste
InverseEtAdditionne
(liste l)
{
   liste res,tmp,tete ;
   int pos,retenue ;
   if (l==NULL)
      return NULL ;
   tete = l ;
   /*l va se mettre en queue */
  while(l->next!=NULL)
     l = l->next ;
  pos = 1 ; retenue = 0 ; res = NULL ;
  while(1)
  {
    tmp = (liste) malloc(sizeof(struct cell_m));
    tmp->previous = NULL ;
    tmp->next = res ;
    tmp->pos = pos++;
    tmp->val = tete->val + l->val + retenue ;
    if (tmp->val>9)
    {
       tmp->val %= 10 ;
       retenue = 1 ;
    }
    else retenue = 0 ;
   if (tmp->next)
      tmp->next->previous = tmp ;
    res = tmp ;
    tete = tete->next ;
    l = l->previous ;
    if(tete == NULL && l == NULL )
    {
       if (retenue!=0)
       {
        tmp = (liste) malloc(sizeof(struct cell_m));
        tmp->previous = NULL ;
        tmp->next = res ;
        tmp->pos = pos++;
        tmp->val = 1 ;
        tmp->next->previous = tmp ;
        res = tmp ;
        }
       break ;
    }
  }
  return res;
}
```

Remarques : on peut itérer la fonction `InverseEtAdditionne` sur des nombres. Tout nombre d'un (et de 2) chiffre(s) est *non-Lychrel* i.e. en itérant cette fonction sur ce chiffre il devient palindromique (son écriture décimale est un palindrome). 90% des nombres inférieurs à 10 000 sont non-Lychrel (au bout de 7 itérations le résultat est palindromique). Un nombre dont les

itérés ne sont jamais palindromiques est un nombre Lychrel ; (on n'en connait aucun — 196 est soupçonné d'être Lychrel).