Durée 3 h Documents autorisés

Sujet d'examen 1

Pratique du C

Février 2009

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. Considérons le code suivant :

```
#include <string.h>
int
main
(int argv, char **argc)
{
 int i ;
 for(i=0;i<strlen(argc[0]);i++)  ;
 return 0 ;
}
```

Questions.

(a) Donner la définition de la fonction `strlen` qui prend en argument un pointeur sur une chaîne de caractères et retourne sa taille.

(b) Avec le code de la fonction principale `main` ci-dessus, combien d'accès mémoire à la chaîne de caractères stockée en `argc[0]` sont effectués.

(c) Modifier ce code pour qu'il ne nécessite qu'un nombre d'accès linéaire en la taille de la chaîne.

2. Parmi les lignes de codes suivantes :

```
/* 1 */  enum VENT {BOREE, NOTOS, EUROS, ZEPHYR};
/* 2 */  enum Vent {BOREE, NOTOS, EUROS, ZEPHYR} VENT;
/* 3 */  typedef VENT enum {BOREE, NOTOS, EUROS, ZEPHYR};
/* 4 */  typedef enum VENT {BOREE, NOTOS, EUROS, ZEPHYR};
/* 5 */  typedef enum {BOREE, NOTOS, EUROS, ZEPHYR} VENT;
```

quelle est la bonne déclaration permettant la définition suivante :

```
VENT d;
```

2 Tri bulle générique

Tri Bulle. Le tri bulle consiste à parcourir la table à trier en intervertissant toute paire d'éléments consécutifs non ordonnés afin qu'après un parcours, le plus grand élément se retrouve à la fin de la table. On recommence cette opération avec la table considérée sans le dernier éléments.

Questions.

1. Donnez la définition de la fonction de prototype :

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

   qui tri — de manière destructive — un tableau d'entiers passé en premiers paramètre et dont la taille est passée en second paramètre.

2. Donnez la définition de la fonction de prototype :

   ```
   void TriBulleGenerique (void *tab,int tabsize, int cellsize, int (*compar)(void*, void *)) ;
   ```

   qui tri — de manière destructive — un tableau dont l'origine est passée en premiers paramètre, la taille en second paramètre, la taille (en octet) de chacune de ces cellules en troisième et une fonction de comparaison de cellule en dernier.

3 Matrices pleines de dimensions variables

La taille d'une matrice pleine de dimensions variables n'est pas connue à la compilation mais seulement lors de sa création lors de l'exécution. Les coefficients de cette matrice sont des entiers machines signés.

On se donne la description des types suivante :

- le type `matrix_t` est un pointeur sur des objets définis suivant le modèle d'identificateur `matrix_m` ;
- un tel objet est :
  - soit un objet d'identificateur `zero` étant vrai si la matrice est nulle et faux dans le cas contraire,
  - soit un objet suivant le modèle d'identificateur `truematrix_m`.
- un objet de la famille d'identificateur `truematrix_m` est composé :
  - d'un entier signé d'identificateur `nblig` codant le nombre de lignes ;
  - d'un entier signé d'identificateur `nbcol` codant le nombre de colonnes ;
  - d'un pointeur sur des entiers signés d'identificateur `body`.

On souhaite disposer de l'opération d'addition de telles matrices : l'addition de deux matrices n'est possible que si ces matrices ont le même nombre de colonnes et le même nombre de lignes.

Une variable du type `matrix_t` vaut NULL si elle ne représente pas une matrice valide (par exemple, si on tente de construire une matrice avec une dimension négative ou si on tente d'additionner 2 matrices de tailles différentes).

Question.

1. Donnez les déclarations des types décrits ci-dessus.
2. Donnez la définition d'une fonction de prototype

   ```
   matrix_t makenullmatrix ();
   ```

   qui construit une matrice zéro ;
3. Donnez la définition d'une fonction de prototype

   ```
   matrix_t makematrix (int, int);
   ```

   qui construit une matrice dont le nombre de lignes est passé en premier paramètre et le nombre de colonnes en second (si un de ces entiers est négatif ou nul, on retourne `NULL`). Cette fonction réserve de l'espace pour les coefficients mais n'affecte pas cet espace.

4. Donnez la définition d'une fonction de prototype

   `void killmatrix (matrix_t);`

   qui désalloue une matrice.
5. Donnez la définition d'une fonction de prototype

   `matrix_t addmatrices (matrix_t,matrix_t);`

   qui retourne la matrice résultant de l'addition des matrices passées en paramètres. Si ces matrices sont de dimensions différentes, cette fonction retourne `NULL`. Si chaque coefficient de la somme de ces matrices est zéro, on retourne une matrice zéro.

Rappels : L'allocation mémoire se fait par le biais de la fonction `malloc` et la désallocation par le biais de la fonction `free`.

4 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).

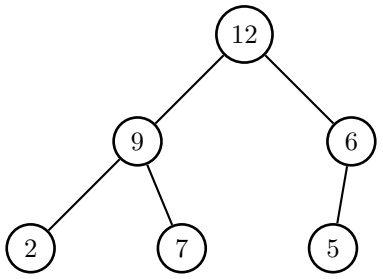
Tas binaire

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

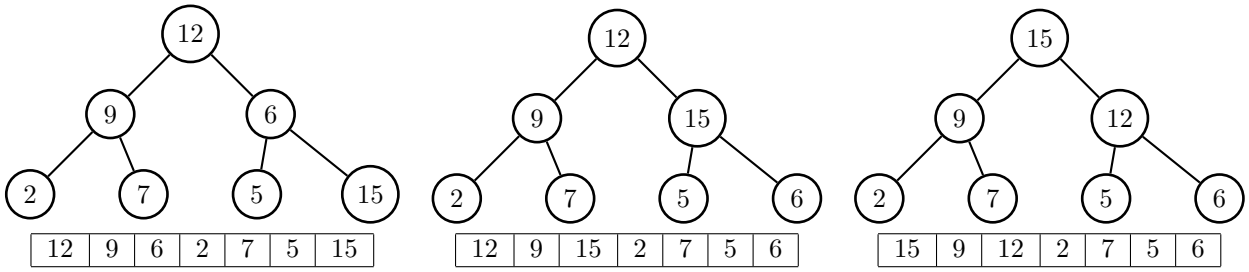
Les tas binaires sont utilisés pour implanter les files de priorités car ils permettent des insertions en temps logarithmiques et un accès direct au plus grand élément.

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 :

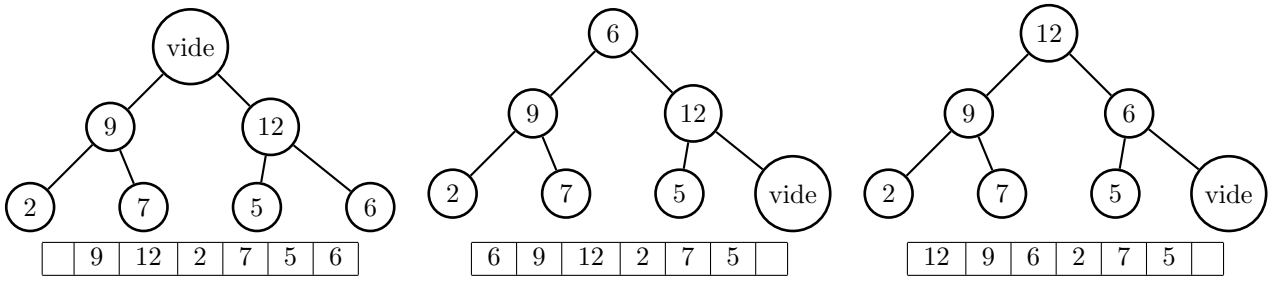


Retirer un élément d'un tas binaire

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 supérieur.

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



4.1 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.