POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S6) **Structures de données 2010/2011**

Devoir surveillé - éléments de correction

Usage profs uniquement.

A – Exercice : Arbres Binaires pour un tournoi (5pt/20min)

1. Faire les déclarations nécessaires pour représenter de tels arbres. Écrire un algorithme qui, étant donné l'arbre binaire d'un tournoi, affiche les matchs disputés et leur vainqueur.

   CORRECTION : Déclaration :

   ```
   type arbre = Pointeur de Noeud
   structure Noeud
      val:entier
      left:arbre
      right:arbre
   fin structure
   ```

   On réalise un algorithme récursif :
   - Si l'arbre est vide on ne fait rien
   - Si l'arbre est <R,G,D> : si R est différent de -1 et G et R non vides, alors on affiche le match et on appelle affiche sur G et R

   ```
   Action afficheMatchs(ab)
   |  D: ab : ArbreBinaire
   |  L: abg,abd : ArbreBinaire
   |  Si non estVide(ab) alors
   |  |  abg ← getGauche(ab)
   |  |  agd ← getDroite(ab)
   |  |  Si getRacine(ab)≠ −1 et abg ≠ Vide et abg ≠ Vide alors
   |  |  |  Afficher("match entre ",getRacine(abg),"et",getRacine(abg))
   |  |  |  Afficher(" : vainqueur ",getRacine(ab))
   |  |  |  afficheMatchs(abg)
   |  |  |  afficheMatchs(abd)
   |  |  Fsi
   |  Fsi
   FAction
   ```

   ■

2. Écrire un algorithme permettant de stocker le résultat d'un match dans l'arbre du tournoi.
   CORRECTION : Je fais une fonction pour rendre un booléen des que l'insertion est faite, comme cela si l'insertion est faite dans le sous-arbre gauche, il n'y a pas besoin de faire l'appel rec dans le sous-arbre droit !

Fonction *insereResultatMatch(j1,j2,j,ab)*
| **D**: ab : ArbreBinaire
| **D**: j1,j2,j :Entiers
| **L**: resug : Booleen
| abg ← *getGauche*(*ab*)
| agd ← *getDroite*(*ab*)
| **Si** *getRacine(abg)=j1 et getRacine(abd)=j2* **alors**
| | putRacine(ab,j)
| | **Retourner** *vrai*
| **Sinon**
| | resug ← insereResultatMatch(j1,j2,j,abg)
| | **Si** *resug (=vrai)* **alors**
| | | resug ← insereResultatMatch(j1,j2,j,abd)
| | **Fsi**
| | **Retourner** *resug*
| **Fsi**
FFonction

■

B – Problème de BTP (le reste)

Codage du problème et fonctions faciles

Question 1 Dessiner la rangée de briques correspondant au tableau suivant :

0	1	0	0	1	0	0

De quelle taille est cette rangée ?
Correction : La rangée de briques est : 1 brique de taille 2, puis 2 de taille 3, soit une rangée de taille 8. ■

Question 2 Déclarer le type rangée de briques (nommé `Rangee`)
Correction :

```
typedef int Rangee[TAILLEMAX];
```

■

Question 3 Écrire une fonction `void imprimerangee(Rangee rangee)` qui imprime une rangée de briques (sous la forme `011100`).
Correction :

```
void imprimerangee(Rangee rangee){
  int i;
  for (i=0;i<TAILLEMAX;i++)
    printf("%d",rangee[i]);
}
```

■

Question 4 Écrire une fonction `bool sontCompatibles(Rangee  r1,Rangee r2)` qui retourne `true` si les deux rangées données sont compatibles, `false` sinon.

Correction : Très difficile également :

```
bool sontCompatibles(Rangee  r1,Rangee r2){
  int i;
  bool resu = true;
  for (i=0;i<TAILLEMAX;i++){
     if (r1[i]==1 && r2[i] ==1) return false;
  }
  return resu;
}
```

■

Résolution

L'algorithme récursif initial

Question 5 Pourquoi la quantité `count(r,1)` vaut-elle 1 pour tout r ?

CORRECTION : Parce qu'il n'y a qu'une seule façon de faire un mur de hauteur 1 avec une rangée donnée. ■

Question 6 Écrire la déclaration du type `ListeRangees` : liste chaînée de `Rangee`.

CORRECTION : Liste chaînée classique.

```
typedef struct Cell{
  Rangee el;
  struct Cell* suivant;
} Cell;

typedef Cell* ListeRangees;
```

■

Question 7 On suppose $\ell = 5$. Développer sur votre copie les différentes étapes de calcul (avec les appels récursifs) de l'appel `count_all(allrows,2)`, `allrows` étant la liste comportant les deux tableaux codant les rangées `0100` et `0010`.

CORRECTION : On va donc dérouler les appels récursifs (arbre) :

- initialement, `sum` vaut 0
- appel à `count(allrows,0100,2)`. La variable auxiliaire est mise à 0 et on parcourt la liste `allrows` :
  - `0100` n'est pas compatible ;
  - `0010` est compatible, donc appel récursif avec $h = 1$ donc la valeur de retour est 1.

  La valeur retournée par la fonction est donc 1.
- retour : la fonction précédente retournant 1, `sum` vaut maintenant 1.
- appel à `count(allrows,0010,2)` qui se comporte pareil que l'autre, et retourne 1
- à la fin `sum` vaut 2, et on retourne 2.

■

Optimisation avec stockage des valeurs intermédiaires

Question 8 Écrire la déclaration d'un type `Triplet` pour stocker des valeurs $(r, h, nbmurs)$ ($nbmurs$ sera le résultat de $count(r, h)$). Écrire une fonction qui permet de tester l'égalité de deux triplets **Attention, r est une rangée**.

CORRECTION : Un triplet en C :

```
typedef struct {
  int r; // rangee
  int h; // hauteur
  int nbmurs;
} Triplet;
```

Le test d'égalité (test de l'égalité de h et nbmurs, et test de l'égalité des rangées) :

```
bool sont_egaux(Triplet tr1,Triplet tr2){
  int i;
  if ((tr1.h != tr2.h) || (tr1.nbmurs != tr2.nbmurs)) return false;
  else {
      for (i=0;i<TAILLEMAX;i++){
         if (r1[i]!=r2[i]) return false;
      }
  return true;
  }
}
```

■

Question 9 Déclarer tout ce qu'il faut pour pouvoir parler de listes de Triplets. Écrire une fonction d'ajout en tête d'un Triplet dans une liste de Triplets.

CORRECTION : Pour les listes de triplets :

```
typedef struct CellTriplet{
  Triplet elem;
  struct CellTriplet* next;
} CellTriplet;

typedef CellTriplet* ListeTriplet;

void insereEnTeteTriplet(ListeTriplet* pl, Triplet tp)
{
  CellTriplet* p = (CellTriplet*)malloc(sizeof(CellTriplet));
  p->elem = tp;
  p->next=*pl;
  *pl=p;
}
```

■

Question 10 Déclarer le type `HashTable` comme étant un tableau (de taille `SIZEHASH`) de listes de Triplets. Écrire une fonction d'initialisation d'un élément de type `HashTable`.

CORRECTION : Déclaration de la Hashtable.

```
typedef ListeTriplet Hashtable[TAILLEHASH];
```

L'initialisation consiste juste à mettre la liste vide dans chacune des cases de la Hastable :

```
void init_ht(Hashtable ht) {
    int i;
    for (i=0; i<TAILLEHASH; i++) {
         ht[i]=NULL;
     }
}
```

■

Question 11 Écrire la fonction de hachage qui permettra de stocker le triplet $(r, h, nbmurs)$ à l'intérieur de la liste située à la case de la table de hachage d'indice i calculé par la formule : $i = (nbuns(r) * h) \mod SIZEHASH$. La fonction $nbuns(r)$ désigne le nombre de 1 de la rangée r, vous pouvez utiliser cette fonction sans la réécrire.

CORRECTION :

```
int hash(Rangee r, int h){
  return (intbin*hauteur) % SIZEHASH;
}
```

■

▶À ce stade, il nous faut faire en sorte de pouvoir :
- rechercher un triplet donné dans une `HashTable` et récupérer le résultat (3ième champ du triplet).
- insérer un triplet si il n'existe pas dans la table.

Question 12 Écrire une ou plusieurs fonctions qui vous permettront de faire ces traitements. Ces fonctions seront utilisées dans la question suivante. **On n'écrira pas que du C, mais on fera également une vraie analyse algorithmique détaillée.**

CORRECTION : Une solution optimisée consiste en une unique fonction de prototype :

```
void search_and_insert(ListeTriplet l, Triplet tr, bool *resu,
                       int *nwalls, CellTriplet *pos)
```

qui :
- affecte vrai à `*resu` si le triplet a été trouvé, et dans ce cas la valeur de count se trouve dans la variable modifiée `nbwalls`.
- dans le cas contraire, affecte faux à `*resu`, et le dernier paramètre de la fonction contient un pointeur vers la position d'insertion.

flemme du correcteur. ■

Question 13 Compléter le code de l'annexe 2 (`count` et `count_all`) pour :
- déclarer et initialiser une `Hashtable` ;
- sauvegarder dans cette table chaque nouveau triplet calculé ;
- avant tout nouveau calcul, chercher si ce calcul a déjà été fait.

CORRECTION : Pour `count_all` la seule modification à faire est la déclaration, l'initialisation de la Hashtable et appel à la fonction `count` en passant la HashTable en paramètre :

```
...
 Hashtable save ;
 init_ht(save);

  while(pc != NULL) {
    int res = count (allrows,(*pc).el,hauteur,save);
    ...
  }
```

Pour la fonction count :

```
int count (ListeRangees allrows, Rangee r, int hauteur, Hashtable save)
{
  if (hauteur==1) return 1;
  else {
      //ici ajouter de quoi chercher si count(allrows,r,hauteur) a deja ete calcule
       Triplet tp; bool resu;int nwalls;
       CellTriplet *pos = NULL;
       tp.r=r; tp.h=hauteur; tp.nbmurs = 0; // initialisation neuneu
       int indice = hash(r,h);
       search_and_insert(save[indice],tp,&resu,&nwalls,pos)

       if (resu) { //si oui : nwalls contient le résultat
          return nwalls;
       }
       //si non :
```

```
      else {
        Cell* pc = allrows;
        int sum = 0;
        while(pc != NULL) {
          if (sontCompatibles(r,(*pc).el))
            {
              sum = sum+ count(allrows,(*pc).el,hauteur-1);
            }
          pc = (*pc).suivant ;
        }
        //ajouter de quoi sauvegarder
        tp.nbmurs = sum;
        insertAfter(pos,tp); // insertion triplet au bon endroit!
        return sum;
      }//fin du "sinon"
} }
```

■