Université Bordeaux 1 Licence **INF 255**
DS 7 avril 2009 Durée : 1h20 – Sans document

Projet de programmation 2

On utilisera le langage C pour répondre aux questions de programmation.

Formules booléennes

Une formule booléenne f(x1,x2,…,xn) est dite satisfiable si il existe (au moins) une affectation de ses variables qui la rende vraie. Par exemple, la fonction suivante d'arité 3 est satisfiable :

```
bool f3(bool *variables)
{
    return variables[0] && variables[1] && !variables[2];
}
```

En effet si on définit `bool v[] = {true,true,false}` l'appel à `f3(v)` retournera `true`.

1) Peut-on décider qu'une formule (donnée par un pointeur de fonction et son arité) n'est pas satisfiable sans avoir testé toutes les combinaisons d'affectation de variables possibles ? En déduire la complexité en O( ) du problème de la satisfiabilité en fonction de l'arité de la formule.

Non car pour toute affectation A il existe une formule qui ne soit satisfaite que par A (la formule « égale à A »), si on ne la teste pas alors on ne peut pas distinguer cette formule de la formule faux. Le nombre d'appels à la fonction formule sera de l'ordre de 2 à la puissance arité.

2) Écrire la fonction

```
bool est_satisfiable(bool (*formule)(bool *variables), int arite) ;
```

qui retourne vrai si et seulement si la fonction booléenne passée en paramètre est satisfiable.

```
bool est_satisfiable(bool (*formule)(bool *variables), int arite)
{
      bool v[arite+1] ;
      memset(v,0,(arite+1)*sizeof(bool)) ;
      while (!v[arite])
       {
            int i ;
            if (formule(v))
                  return true;
            for(i = 0 ; v[i] && i < arite ; i++)
                   v[i]=false ;
            v[i] = true ;
      }
      return false ;
}
```

B-arbre

3) Écrire l'interface d'un module b-arbre générique (version dynamique).

Voir votre projet.

La procédure d'insertion utilisée par Cormen & al (appelée ici `insérer_dans_noeud _incomplet ( )`) peut être qualifiée de *pessimiste* car elle éclate parfois inutilement des nœuds

complets. Nous allons proposer une solution d'insertion plus économique en espace : une procédure d'insertion optimiste qui n'éclatera des nœuds que si la feuille cible est complète. Pour cela on va d'abord repérer la position d'insertion (potentielle) de la clef en descendant dans le b-arbre sans éclater de nœud. Arrivé à une feuille incomplète, on insérera naturellement la clef dans celle-ci. Arrivé à une feuille complète, on utilisera la procédure d'insertion pessimiste `inserer_dans_noeud_incomplet ( )` en repartant du *bon* nœud.

4) Supposons que le nœud racine N0 soit incomplet et soit N0, N1, …, Np les nœuds rencontrés par la procédure d'insertion optimiste du nœud racine (N0) au nœud feuille (Np). En supposant que le nœud feuille est complet, quelle est la caractéristique du *bon* nœud auquel il faut remonter pour procéder à l'insertion pessimiste sans éclater inutilement des nœuds ?

Il s'agit du dernier nœud incomplet rencontré, c'est à dire le nœud incomplet Ni tel que pour tout k compris entre i+1 et p Nk est complet.

5) En tenant compte de la question précédente, quel paramètre supplémentaire doit avoir la fonction d'insertion optimiste par rapport à `inserer_dans_noeud_incomplet()`pour pouvoir remonter directement au bon nœud.

Un pointeur vers le *bon nœud*.

6) Donner le code de la fonction d'insertion optimiste pour les b-arbres d'entiers et modifier en conséquence la fonction `b_arbre_inserer_clef` (sans modifier le traitement de la racine pleine) en vous appuyant sur les définitions données au verso.

```
static void
inserer_optimiste(arbre a, b_noeud n, clef c,
                  b_noeud dernier_non_plein)
{
  int position_insertion = position_dans_les_clefs(a, c, n);

  if (n->est_une_feuille)
    {
      if (n->cardinal == a->arite)
       return inserer_dans_noeud_incomplet(a, dernier_non_plein, c);
      ajouter_clef(c,n,position_insertion);
      return;
    }

  inserer_optimiste(a, n->les_fils[position_insertion], c,
               n->cardinal == a->arite ? dernier_non_plein : n);
}


void
b_arbre_inserer_clef(arbre a, clef c)
{
  if (a->racine->cardinal == a->arite)
    {
      b_noeud n = nouveau_noeud(a,0,false);
      n->cardinal = 0;
      n->les_fils[0]=a->racine;
      a->racine = n;
      decouper_fils_plein(a, a->racine, 0);
    }
  inserer_optimiste(a, a->racine, c, a->racine);
}
```

NB. La seule ligne à modifier est précisée en commentaire.

```
typedef struct b_noeud *b_noeuds;

typedef struct b_noeud *tableau_de_noeuds;
typedef int clef;
typedef clef* tableau_de_clefs;

struct b_arbre{
  int arite;
  b_noeud racine;
};

struct b_noeud {
  int cardinal;
  bool est_une_feuille;
  tableau_de_noeuds les_fils;
  tableau_de_clefs les_clefs;
};


static int
position_dans_les_clefs(clef c, b_noeud n)
{
  int i;
  for(i = 0; i < n->cardinal && n->les_clefs[i] < c; i++ )
    ;
  return i;
}

static void
ajouter_clef(clef c, b_noeud n, int position)
{
  memmove(n->les_clefs + position +1,
        n->les_clefs+position,
        (n->cardinal - position) * sizeof(*n->les_clefs));
  memcpy(n->les_clefs + position, &c, sizeof(c));
  n->cardinal++;
}

void
b_arbre_inserer_clef(arbre a, clef c)
{
  if (a->racine->cardinal == a->arite)
    {
      b_noeud n = nouveau_noeud(a,0,false);
      n->les_fils[0]=a->racine;
      a->racine = n;
      decouper_fils_plein(a, a->racine, 0);
    }
  inserer_dans_noeud_incomplet(a, a->racine, c); // à modifier
}


static void
inserer_dans_noeud_incomplet(arbre a, b_noeud n, clef c){
  int position_insertion = position_dans_les_clefs(c, n);
 …
}
```