```
/*********************************************/
/*** Gestion d'arbres binaires de recherche ***/
/*********************************************/

#include "ABR.h"


PArbre RechercheABR(PArbre A, Element x)
{
 if (!A) return ARBRENULL;
 if (x < A->elmt)
  return RechercheABR(A->fg, x);
 else
  if (x > A->elmt)
   return RechercheABR(A->fd, x);
 return A;
}

PArbre InsereABR(PArbre A, Element x)
{
 if (!A) return creerArbre(x);
 if (x < A->elmt)
  A->fg = InsereABR(A->fg, x);
 else
  if (x > A->elmt)
   A->fd = InsereABR(A->fd, x);
 return A;
}


PArbre InsereABRIteratif(PArbre A, Element x)
{
 PArbre tmp = A;

 if (!A) return creerArbre(x);
 do {
  if (x == tmp->elmt)
   return A;
  if (x < tmp->elmt)
   if (tmp->fg)
    tmp = tmp->fg;
   else {
    tmp->fg = creerArbre(x);
    return A;
   }
  else // x > tmp->elmt
   if (tmp->fd)
    tmp = tmp->fd;
   else {
    tmp->fd = creerArbre(x);
    return A;
   }
 }
 while (1);
}
```

```
// Dans SuppABR, va supprimer le fils le plus grand (a droite) du fils gauche pris en compte
PArbre SuppMax(PArbre A, Element *x)
{
 PArbre tmp;

 if (A->fd)
  A->fd = SuppMax(A->fd, x);
 else {
  *x = A->elmt; //recuperation de la valeur a remonter en parametre
  tmp = A;
  A = A->fg;
  free(tmp);
 }
 return A;
}


PArbre SuppABR(PArbre A, Element x)
{
 PArbre tmp;

 if (A)
  if (x > A->elmt)
   A->fd = SuppABR(A->fd, x);
  else
   if (x < A->elmt)
    A->fg = SuppABR(A->fg, x);
   else
    if (A->fg)
     A->fg = SuppMax(A->fg, &(A->elmt)); // on remplace elmt par la grande valeur inferieure
    else { // si pas de fils gauche : la racine devient le fils droit
     tmp = A;
     A = A->fd;
     free(tmp);
    }
 return A;
}

/*********************************************/
/*** Les fonctions suivantes permettent de vérifier si un Arbre binaire est un ABR***/
/*********************************************/

PPile empile(PPile p, Element e) {
 PPile ptr;
 if ((ptr = MALLOC(Pile)) == NULL) {
  fprintf(stderr, "ERREUR ALLOCATION MEMOIRE FILE");
  exit(1);
 }
 *ptr = (Pile) { e, p };
 return ptr;
}

void suppPile(PPile p) {
 PPile tmp;
```

```
 while (p) {
  tmp = p;
  p = p->suiv;
  free(tmp);
 }
}

/* Mets l'arbre dans une pile par parcours infixe */
PPile ABtoPile(PArbre a, PPile p) {
 if (! estVide(a)) {
  p = ABtoPile(filsGauche(a), p);
  p = empile(p, a->elmt);
  p = ABtoPile(filsDroit(a), p);
 }
 return p;
}

int estUnABR2(PArbre a) {
 PPile p = NULL, tmp;
 if (estVide(a))
  return true;
 p = ABtoPile(a, p); // Arbre infixe vers pile
 tmp = p;
 while (p->suiv) { // parcours pile pour ordre
  if (p->val < p->suiv->val) {
   suppPile(tmp);
   return false;
  }
  p = p->suiv;
 }
 suppPile(tmp);
 return true;
}

bool estUnABRRec(PArbre a, Element *e) {
 if (estVide(a))
  return true;
 if ( ! estUnABRRec(filsGauche(a), e))
  return false;
 if (*e > racine(a))
  return false;
 *e = racine(a);
 return estUnABRRec(filsDroit(a), e);
}

bool estUnABR(PArbre a) {
 Element e = ELEMENTNULL;
 return estUnABRRec(a, &e);
}
```