```
#include "stdafx.h"
#include "PreConst.h"
#include "stdlib.h"
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
typedef char TElemType; //  元素数据类型

/*  二叉链表储存结构  */

typedef struct BiTNode {
    TElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;

Status CreateBiTree(BiTree &T) {
    //先序序列建立二叉树
    char ch;
    scanf("%c",&ch);
    if (ch=='*') T = NULL;
    else {
        if (!(T = (BiTNode *)malloc(sizeof(BiTNode)))) return ERROR;
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return OK;
}

Status PrintElement(TElemType e) {
    //  访问函数
    printf("%c", e);
    return OK;
}

Status PreOrderTraverse(BiTree T, Status(*Visit)(TElemType)) {
    //  先序遍历二叉树的递归算法

    if (T) {
        if (Visit(T->data))
            if (PreOrderTraverse(T->lchild, Visit))
                if (PreOrderTraverse(T->rchild, Visit)) return OK;
        return ERROR;
    }else return OK;
}
```

```
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType)) {
    // 中序遍历二叉树的递归算法
    if (T) {
        if (InOrderTraverse(T->lchild, Visit))
            if (Visit(T->data))
                if (InOrderTraverse(T->rchild, Visit)) return OK;
        return ERROR;
    }else return OK;
}

Status PostOrderTraverse(BiTree T, Status(*Visit)(TElemType)) {
    // 后序遍历二叉树的递归算法
    if (T) {
        if (PostOrderTraverse(T->lchild, Visit))
            if (PostOrderTraverse(T->rchild, Visit))
                if (Visit(T->data)) return OK;
        return ERROR;
    }else return OK;
}

/*  栈存储结构及操作   */

typedef struct {
    BiTree *base;
    BiTree *top;
    int stacksize;
}Stack;

Status InitStack(Stack &S) {
    //构造空栈
    S.base = (BiTree*)malloc(STACK_INIT_SIZE * sizeof(BiTree));
    if (!S.base) exit(OVERFLOW);
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}

Status GetTop(Stack S, BiTree &e){
    //读栈顶元素
    if (S.top == S.base) return ERROR;
        e = *(S.top - 1);
        return OK;
}
```

```
Status Push(Stack &S, BiTree e){
    //入栈
    if (S.top - S.base >= S.stacksize) {
        S.base = (BiTree*)realloc(S.base, (S.stacksize + STACKINCREMENT) *
sizeof(BiTree));
        if (!S.base) exit(OVERFLOW);
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e;
    return OK;
}

Status Pop(Stack &S, BiTree &e){
    //出栈
    if (S.top == S.base) return ERROR;
    e = *--S.top;
    return OK;
}

Status StackEmpty(Stack S){
    //判栈空
    if (S.base == S.top) return TRUE;
    else return FALSE;
}

Status InOrderTraverse2(BiTree T, Status (*Visit)(TElemType)) {
    // 中序遍历二叉树的非递归算法
    Stack S;
    BiTree p;
    InitStack(S);    Push(S, T);
    while (!StackEmpty(S)) {
        while (GetTop(S, p) && p) Push(S, p->lchild);
        Pop(S, p);
        if (!StackEmpty(S)) {
            Pop(S, p);
            if (!Visit(p->data)) return ERROR;
            Push(S, p->rchild);
        }
    }
    return OK;
}
```

```
#define MAXLEN 100

void LevelOrderTraverse(BiTree T, Status (*Visit)(TElemType)) {
    // 层次遍历二叉树
    struct node
    {
        BiTree vec[MAXLEN];
        int f,r;
    }q;
        q.f=0;
    q.r=0;
    if (T != NULL) Visit(T->data);
    q.vec[q.r]=T;
    q.r=q.r+1;
    while (q.f<q.r) {
        T=q.vec[q.f];     q.f=q.f+1;
        if (T->lchild != NULL) {
            Visit(T->lchild->data);
            q.vec[q.r]=T->lchild;
            q.r=q.r+1;
        }
        if (T->rchild != NULL) {
            Visit(T->rchild->data);
            q.vec[q.r]=T->rchild;
            q.r=q.r+1;
        }
    }
}

int BiTreeDepth(BiTree T) {
    //求二叉树的深度
     int depthval, depthLeft, depthRight;
     if (!T) depthval = 0;
     else {
         depthLeft = BiTreeDepth( T->lchild );
         depthRight= BiTreeDepth( T->rchild );
         depthval = 1 + (depthLeft > depthRight ? depthLeft : depthRight);
     }
     return depthval;
}
```

```
/*   树的二叉链表储存结构  */
typedef struct CSNode{
        TElemType data;
        struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;

/*   队列存储结构及操作   */
typedef struct QNode {
      CSTree data;
      struct QNode *next;
}QNode, *QueuePtr;

typedef struct {
      QueuePtr front;
      QueuePtr rear;
}LinkQueue;

Status InitQueue(LinkQueue &Q) {
      //构造空队列
      Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
      if (!Q.front) exit(OVERFLOW);
      Q.front->next = NULL;
      return OK;
}

Status DestoryQueue(LinkQueue &Q) {
      //销毁队列
      while (Q.front) {
            Q.rear = Q.front->next;
            free(Q.front);
            Q.front = Q.rear;
      }
      return OK;
}

Status EnQueue(LinkQueue &Q, CSTree e) {
      //入队
      QueuePtr p;
      p = (QueuePtr)malloc(sizeof(QNode));
      if (!p) exit(OVERFLOW);
      p->data = e; p->next = NULL;
      Q.rear->next = p;
      Q.rear = p;
      return OK; }
```

```
Status DeQueue(LinkQueue &Q, CSTree &e) {
    //出队
    QueuePtr p;
    if (Q.front == Q.rear) return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next = p->next;
    if (Q.rear == p) Q.rear = Q.front;
    free(p);
    return OK;
}

Status GetHead(LinkQueue &Q, CSTree &e) {
    //读队头
    QueuePtr p;
    if (Q.front == Q.rear) return ERROR;
    p = Q.front->next;
    e = p->data;
    return OK;
}

CSTree GetTreeNode(TElemType e) {
   //建立树的孩子-兄弟链表结点
    CSTree p;
    p = (CSTree)malloc(sizeof(CSNode));
    if (!p) exit(OVERFLOW);
    p->data = e;
    p->firstchild = NULL;
    p->nextsibling = NULL;
    return p;
}

Status CreatTree(CSTree &T) {
   //建立树的孩子-兄弟链表
    char first = ' ', second = ' ';
    int result = 0;
    LinkQueue Q;
    CSTree p, s, r;
    InitQueue(Q);
    T = NULL;
    for(scanf("%c%c", &first, &second); second != '#'; result = scanf("%c%c", &first, &second)) {
        p = GetTreeNode(second);
        EnQueue(Q, p);
        if (first == '#')    T = p;
        else {
            GetHead(Q,s);
            while (s->data != first) {
                DeQueue(Q,s);    GetHead(Q,s);
            }
            if (!(s->firstchild)) {
                s->firstchild = p;
                r = p;
            }else {
                r->nextsibling = p;
                r = p;

            }
        }
    }
    return OK;
}

int TreeDepth(CSTree T) {
    //求树的深度
     int h1, h2;
     if (!T) return 0;
     else {
         h1 = TreeDepth(T->firstchild);
         h2 = TreeDepth(T->nextsibling);
         return(((h1+1)>h2)?(h1+1):h2);
     }
}

int main(int argc, char* argv[])
{
    BiTree testT;

    printf("请输入二叉树先序序列（如 AB*C***）：");
    CreateBiTree(testT);
    printf("\n");

    printf("二叉树的深度是：");
    printf("%d", BiTreeDepth(testT));
    printf("\n");

    printf("先序递归遍历顺序：");
    PreOrderTraverse(testT, PrintElement);
    printf("\n");

    printf("中序递归遍历顺序：");
    InOrderTraverse(testT, PrintElement);
    printf("\n");

    printf("后序递归遍历顺序：");
    PostOrderTraverse(testT, PrintElement);
    printf("\n");

    printf("层次非递归遍历顺序：");
    LevelOrderTraverse(testT, PrintElement);
    printf("\n");

    printf("中序非递归遍历顺序：");
    InOrderTraverse2(testT, PrintElement);
    printf("\n\n");

    while (getchar() != '\n'); //清除缓冲区字符

    CSTree testT2;

    printf("自上而下自左至右输入树的各条边（如#AABACADCECFEG##）：");
    CreatTree(testT2);
    printf("\n");

    printf("树的深度是：");
    printf("%d", TreeDepth(testT2));
    printf("\n");
    return 0;
}
```