

实验报告 (week-07)  
2020111235 马靖淳

一. 实验任务

1. 创建二叉树
2. 遍历二叉树

二. 实验上机时间 6h

三. 知识点

1. 二叉树的基本操作

- (1) 求二叉树的根
- (2) 求二叉树中某个结点的值
- (3) 求二叉树某一结点的双亲
- (4) 求二叉树某一结点的左孩子
- (5) 求二叉树某一结点的右孩子
- (6) 求二叉树某一结点的左兄弟
- (7) 求二叉树某一结点的右兄弟
- (8) 判空
- (9) 求深度
- (10) 先序遍历**
- (11) 中序遍历**
- (12) 后序遍历**
- (13) 层序遍历**
- (14) 初始化二叉树
- (15) 为二叉树中某个结点赋值
- (16) 用先序遍历创建二叉树**
- (17) 插入某一结点左右孩子
- (18) 清空二叉树
- (19) 销毁二叉树
- (20) 删除

2. 遍历算法

- (1) 先序遍历算法
- (2) 中序遍历算法
- (3) 后序遍历算法
- (4) 层序遍历算法
- (5) 递归算法转化为非递归算法 (中序遍历非递归算法 (1))
- (6) 递归算法转化为非递归算法 (中序遍历非递归算法 (2))
- (7) 递归算法转化为非递归算法 (先序遍历非递归算法)

四. 源代码及结果截屏

1. LinkQueue.h  
详情请见 week04
2. Stack.h  
详情请见 week03

### 3. Status.h

### 4. BiTree.h

```
#ifndef BITREE_H
#define BITREE_H

#include <stdlib.h>
#include "Status.h"

typedef int TElemType;
typedef struct BiTNode//结点结构
{
    TElemType data;
    struct BiTNode* lchild, * rchild;
}BiTNode, * BiTree;

BiTree Root(BiTree T);//1. 求二叉树的根
int Value(BiTree T, BiTree e);//2. 求二叉树中某个结点的值
Status Search(BiTree T, BiTree e);//辅助value函数
BiTree Parent(BiTree T, BiTree e);//3. 求二叉树某一结点的双亲
BiTree LeftChild(BiTree T, BiTree e);//4. 求二叉树某一结点的左孩子
BiTree RightChild(BiTree T, BiTree e);//5. 求二叉树某一结点的右孩子
BiTree LeftSibling(BiTree T, BiTree e);//6. 求二叉树某一结点的左兄弟
BiTree RightSibling(BiTree T, BiTree e);//7. 求二叉树某一结点的右兄弟
Status BiTreeEmpty(BiTree T);//8. 判空
int BiTreeDepth(BiTree T);//9. 求深度
Status Visit(TElemType e);//辅助函数
Status PreOrderTraverse(BiTree T, Status(*Visit)(TElemType));//10. 先序遍历
Status PreOrderTraverse1(BiTree T, Status(*Visit)(TElemType));
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType));//11. 中序遍历
Status InOrderTraverse1(BiTree T, Status(*Visit)(TElemType));
Status InOrderTraverse2(BiTree T, Status(*Visit)(TElemType));
Status PostOrderTraverse(BiTree T, Status(*Visit)(TElemType));//12. 后序遍历
Status LevelOrderTraverse(BiTree T, Status(Visit)(TElemType));//13. 层序遍历
Status InitBiTree(BiTree* T);//14. 构造空二叉树
Status Assign(BiTree T, BiTree* e, TElemType value);//15. 为二叉树中某个结点赋值
Status CreateBiTree(BiTree* T);//16. 用先序遍历创建二叉树
Status InsertChild(BiTree T, BiTree p, int LR, BiTree c);//17. 插入
Status ClearBiTree(BiTree* T);//18. 清空二叉树
Status DestroyBiTree(BiTree* T);//19. 销毁二叉树
Status DeleteChild(BiTree T, BiTree p, int LR);//20. 删除
```

```
#endif
```

## 5. LinkQueue.c

详情请见 week04

## 6. Stack.c

详情请见 week03

## 7. BiTree.c

```
#include "BiTree.h"
```

```
#include "LinkQueue.h"
```

```
#include "LinkQueue.c"
```

```
#include "Stack.h"
```

```
#include "Status.h"
```

```
#include <stdio.h>
```

```
#include <malloc.h>
```

```
BiTree Root(BiTree T)
```

```
{  
    //返回二叉树的根，前提T存在  
    return T;  
}
```

```
int Value(BiTree T, BiTree e)
```

```
{  
    //若e是T中某个结点，返回e的值，前提T存在  
    if (Search(T, e) == TRUE)  
        return e->data;  
    else  
        return '.';  
}
```

```
Status Search(BiTree T, BiTree e)
```

```
{  
    //辅助函数  
    if (!T)  
        return FALSE;  
    if (T == e || Search(T->lchild, e) == TRUE || Search(T->rchild, e) == TRUE)  
        return TRUE;  
    else  
        return FALSE;  
}
```

```
BiTree Parent(BiTree T, BiTree e)
```

```
{  
    //若e是T中某个非根结点，则返回它的双亲，否则返回NULL，前提T存在  
    if (T == e)
```

```

        return NULL;

    if (T->lchild == e || T->rchild == e)
        return T;
    else if (Search(T->lchild, e) == TRUE)
        return Parent(T->lchild, e);
    else if (Search(T->rchild, e) == TRUE)
        return Parent(T->rchild, e);
    else
        return NULL;
}

```

```

BiTree LeftChild(BiTree T, BiTree e)
{
    //若e是T中某个结点，返回e的左孩子。若e无左孩子，则返回NULL，前提T存在
    if (Search(T, e) == TRUE)
        return e->lchild;
    else
        return NULL;
}

```

```

BiTree RightChild(BiTree T, BiTree e)
{
    //若e是T中某个结点，返回e的右孩子。若e无右孩子，则返回NULL，前提T存在
    if (Search(T, e) == TRUE)
        return e->rchild;
    else
        return NULL;
}

```

```

BiTree LeftSibling(BiTree T, BiTree e)
{
    //e是T中某个结点，返回e的左兄弟。若e是其双亲的左孩子或无左兄弟，则返回NULL，前提T存在
    if (Search(T, e) == TRUE && Parent(T, e)->lchild != e)
        return Parent(T, e)->lchild;
    else
        return NULL;
}

```

```

BiTree RightSibling(BiTree T, BiTree e)
{
    //e是T中某个结点，返回e的右兄弟。若e是其双亲的右孩子或无右兄弟，则返回NULL，前提T存在
}

```

```

    if (Search(T, e) == TRUE && Parent(T, e)->rchild != e)
        return Parent(T, e)->rchild;
    else
        return NULL;
}

Status BiTreeEmpty(BiTree T)
{
    //若T为空二叉树, 则返回TRUE, 否则FALSE
    if (T)
        return FALSE;
    else
        return TRUE;
}

int BiTreeDepth(BiTree T)
{
    int lchildHeight, rchildHeight;
    if (T == NULL)
        return 0;
    lchildHeight = BiTreeDepth(T->lchild);
    rchildHeight = BiTreeDepth(T->rchild);
    return (lchildHeight > rchildHeight) ? (1 + lchildHeight) : (1 + rchildHeight);
}

Status Visit(TElemType e)
{
    printf("%c", e);
    return OK;
}

Status PreOrderTraverse(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构, 先序遍历二叉树T的递归算法
    if (T)
    {
        if (Visit(T->data))//访问结点
            if (PreOrderTraverse(T->lchild, Visit))
                if (PreOrderTraverse(T->rchild, Visit))
                    return OK;
        return ERROR;
    }
    else
        return OK;
}

```

```
}
```

```
Status PreOrderTraverse1(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，先序遍历二叉树T的非递归算法
    SqStack S;
    BiTree p;
    InitStack(&S);
    p = T;
    while (p || !StackEmpty(S))
    {
        if (p)
        {
            if (!Visit(p->data))
                return ERROR;
            Push(&S, p->rchild); //右孩子入栈
            p = p->lchild;
        }
        else //返回上一层，继续遍历未曾访问的结点
        {
            Pop(&S, &p);
        }
    }
    return OK;
} //PreOrderTraverse
```

```
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，中序遍历二叉树T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;
} //InOrderTraverse
```

```
Status InOrderTraverse1(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，中序遍历二叉树T的非递归算法 (1)
```

```

SqStack S;
BiTree p;
InitStack(&S);
Push(&S, T); //根指针进栈
while (!StackEmpty(S))
{
    while (GetTop(S, &p) && p)
        Push(&S, p->lchild); //向左走到尽头
    Pop(&S, &p); //空指针退栈
    if (!StackEmpty(S)) //访问结点，向右一步
    {
        Pop(&S, &p);
        if (!Visit(p->data))
            return ERROR;
        Push(&S, p->rchild);
    }
}
return OK;
}

Status InOrderTraverse2(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，中序遍历二叉树T的非递归算法(2)
    SqStack S;
    BiTree p;
    InitStack(&S);
    p = T;
    while (p || !StackEmpty(S))
    {
        if (p)
        {
            Push(&S, p); //非空指针进栈，继续左进
            p = p->lchild;
        }
        else //上层指针退栈，访问其所指结点，再向右进
        {
            Pop(&S, &p);
            if (!Visit(p->data))
                return ERROR;
            p = p->rchild;
        }
    }
    return ERROR;
} //InOrderTraverse

```

```

Status PostOrderTraverse(BiTree T, Status(*Visit)(TElemType))
{
    //采用二叉链表存储结构，后续遍历二叉树T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;
} //PostOrderTraverse

```

```

Status LevelOrderTraverse(BiTree T, Status(Visit)(TElemType))
{
    if (T == NULL)
        return ERROR;
    LinkQueue Q;
    BiTree p;
    InitQueue(&Q);
    EnQueue(&Q, T);
    while (!QueueEmpty(Q))
    {
        DeQueue(&Q, &p);
        if (!Visit(p->data))
            return ERROR;
        if (p->lchild != NULL)
            EnQueue(&Q, p->lchild);
        if (p->rchild != NULL)
            EnQueue(&Q, p->rchild);
    }
    return OK;
}

```

```

Status InitBiTree(BiTree* T)
{
    //构造空二叉树
    (*T) = (BiTree) malloc(sizeof(BiTNode));
    if (!(*T))
        exit(OVERFLOW);
    (*T)->lchild = NULL;
}

```



```

        (*T)->rchild = NULL;
        return OK;
    }

Status Assign(BiTree T, BiTree* e, TElemType value)
{
    //若e是T中某个结点, 结点e赋值为value, 前提T存在
    if (Search(T, *e) == TRUE)
    {
        (*e)->data = value;
        return OK;
    }
    return ERROR;
}

Status CreateBiTree(BiTree *T)
{
    //用先序遍历创建二叉树
    //example1:AB#C##D## example2:ABC##DE#G##F###
    char ch;
    ch = getchar();
    if (ch == '#')
        *T = NULL;
    else
    {
        *T = (BiTNode*)malloc(sizeof(BiTNode));
        if (!(*T))
            return ERROR;
        (*T)->data = ch; //生成结点
        CreateBiTree(&(*T)->lchild); //构造左子树
        CreateBiTree(&(*T)->rchild); //构造右子树
    }
    return OK;
} //CreateBiTree

Status InsertChild(BiTree T, BiTree p, int LR, BiTree c)
{
    //T存在, p指向T中某个结点, LR为0或1, 非空二叉树c与T不相交且右子树为空。
    //根据LR为0或1, 插入c为T中p所指结点的左或右子树。p所指结点的原有左或右子树则成为c的
    右子树。
    if (!T || Search(T, p) == FALSE)
        return ERROR;
    BiTree q;
    if (LR == 0)

```

```

{
    q = p->lchild;
    p->lchild = c;
    c->rchild = q;
}
else if (LR == 1)
{
    q = p->rchild;
    p->rchild = c;
    c->rchild = q;
}
else
    return ERROR;
return OK;
}

```

```

Status ClearBiTree(BiTree* T)
{
    if (!(*T))
    {
        return FALSE;
    }
    ClearBiTree(&((*T)->lchild));
    ClearBiTree(&((*T)->rchild));
    free(*T);
    *T = NULL;
    return TRUE;
}

```

```

Status DestroyBiTree(BiTree* T)
{
    //销毁二叉树，前提T存在
    if (!(*T))
        return OK;
    DestroyBiTree(&((*T)->lchild));
    DestroyBiTree(&((*T)->rchild));
    free(*T);
    return OK;
}

```

```

Status DeleteChild(BiTree T, BiTree p, int LR)
{
    //T存在，p指向T中某个结点，LR为0或1。根据LR为0或1，删除T中p所指结点的左或右子树
    if (T && Search(T, p) == FALSE)

```

```

        return ERROR;
    if (LR == 0)
    {
        DestroyBiTree(&p->lchild);
        p->lchild = NULL;
    }
    else if (LR == 1)
    {
        DestroyBiTree(&p->rchild);
        p->rchild = NULL;
    }
    else
        return ERROR;
    return OK;
}

```

## 8. test.c

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include "LinkQueue.h"
#include "Stack.h"
#include "BiTree.h"
int main()
{
    BiTree T;
    //InitBiTree test
    if (InitBiTree(&T))
        printf("InitBiTree success!\n");
    else
        printf("InitBiTree unsucccess!\n");

    //CreateBiTree test
    char ch1, ch2, ch3;
    printf("请输入一个二叉树T:");
    if (CreateBiTree(&T))
        printf("CreateBiTree success!\n");
    else
        printf("CreateBiTree unsucccess!\n");
    ch1 = getchar();//吸收换行符
}

```

```

//PreOrderTraverse test
printf("先序遍历结果为: ");
PreOrderTraverse(T, Visit);
printf("\n");

//PreOrderTraverse1 test
printf("先序遍历非递归结果为: ");
PreOrderTraverse(T, Visit);
printf("\n");

//ClearBiTree test
if (ClearBiTree(&T))
    printf("ClearBiTree success!\n");
else
    printf("ClearBiTree unsuccessful!\n");
printf("请输入新的二叉树T: ");
CreateBiTree(&T);
ch2 = getchar(); //吸收换行符

//InOrderTraverse test
printf("中序遍历结果为: ");
InOrderTraverse(T, Visit);
printf("\n");

//InOrderTraverse1 test
printf("中序遍历非递归1结果为: ");
InOrderTraverse1(T, Visit);
printf("\n");

//InOrderTraverse2 test
printf("中序遍历非递归2结果为: ");
InOrderTraverse2(T, Visit);
printf("\n");

//Root & Value test
printf("根为: %c\n", Value(T, Root(T)));

//BiTreeDepth test
printf("S的深度为: %d\n", BiTreeDepth(T));

//Parent test
BiTree m = T->lchild->rchild;
BiTree n = T->rchild;

```

```

printf("D为: %c, F为: %c\n", Value(T, m), Value(T, n));
printf("D的双亲为: %c, F的双亲为: %c\n", Value(T, Parent(T, m)), Value(T, Parent(T,
n)));

//Leftchild Rightchild Leftsibling Rightsibling test
printf("D的左孩子为: %c, F的左孩子为: %c\n", Value(T, LeftChild(T, m)), Value(T,
LeftChild(T, n)));
printf("D的右孩子为: %c, F的右孩子为: %c\n", Value(T, RightChild(T, m)), Value(T,
RightChild(T, n)));
printf("D的左兄弟为: %c, F的左兄弟为: %c\n", Value(T, LeftSibling(T, m)), Value(T,
LeftSibling(T, n)));
printf("D的右兄弟为: %c, F的右兄弟为: %c\n", Value(T, RightSibling(T, m)), Value(T,
RightSibling(T, n)));

//Assign test
printf("将F的值改为S\n");
Assign(T, &n, 'S');

//PostOrderTraverse test
printf("后序遍历结果为: ");
PostOrderTraverse(T, Visit);
printf("\n");

//Insertchild test
BiTree c;
InitBiTree(&c);
printf("请输入要插入结点的值:");
CreateBiTree(&c);
ch3= getchar();
InsertChild(T, n, 0, c);//插到F左侧

//LevelOrderTraverse test
printf("插入F左孩子后层序遍历结果为: ");
LevelOrderTraverse(T, Visit);
printf("\n");

//DeleteChild test
DeleteChild(T, m, 0);//删除m左侧
printf("删除D左孩子后先序遍历T: ");
PreOrderTraverse(T, Visit);
printf("\n");

//Destroy test
if (DestroyBiTree(&T))

```

```

        printf("DestroyBiTree success!\n");
    else
        printf("DestroyBiTree unsucces!\n");

    system("pause");
    return 0;
}

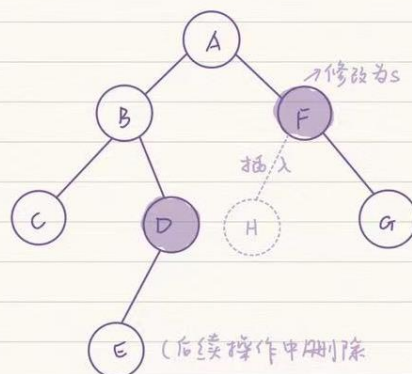
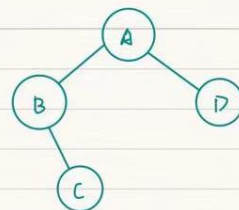
```

C:\Users\DELL\source\repos\BiTree\Debug\BiTree.exe

```

InitBiTree success!
请输入一个二叉树T:AB#C##D##
CreateBiTree success!
先序遍历结果为: ABCD
先序遍历非递归结果为: ABCD
ClearBiTree success!
请输入新的二叉树T: ABC##DE###F#G##
中序遍历结果为: CBEDAFG
中序遍历非递归1结果为: CBEDAFG
中序遍历非递归2结果为: CBEDAFG
根为: A
S的深度为: 4
D为: D, F为: F
D的双亲为: B, F的双亲为: A
D的左孩子为: E, F的左孩子为: .
D的右孩子为: ., F的右孩子为: G
D的左兄弟为: C, F的左兄弟为: B
D的右兄弟为: ., F的右兄弟为: .
将F的值改为S
后序遍历结果为: CEDBGSA
请输入要插入结点的值:H##
插入F左孩子后层序遍历结果为: ABCDSHG
删除D左孩子后先序遍历T: ABCDSHG
DestroyBiTree success!
请按任意键继续. . .

```



## 五. 实验总结

### 1. 直接添加现有项出现错误

E1696	无法打开源文件 "Stack.h"	BiTree	test.c	4
C6308	"realloc"可能返回 null 指针: 将 null 指针赋给"*S.base"(后者将作为参数传递给"realloc")将导致原始内存块泄漏。	BiTree	Stack.c	52
C4047	"函数":"QElemType *"与"QElemType"的间接级别不同	BiTree	LinkQueue.c	124

原因是项目的文件路径在项目没有设置正确

解决方法:

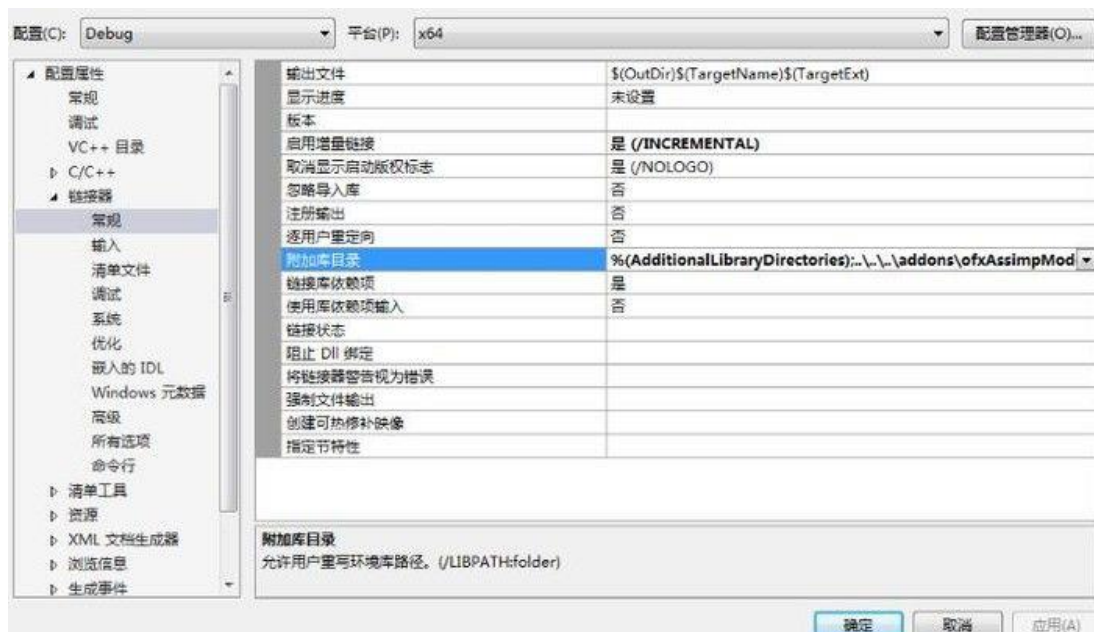
步骤 1、明确自己项目中解决方案的配置和平台 (具体根据自己的项目设置); 在解决方案中选中自己的项目, 右键弹出选项框后选择属性, 之后会出现如下窗口;



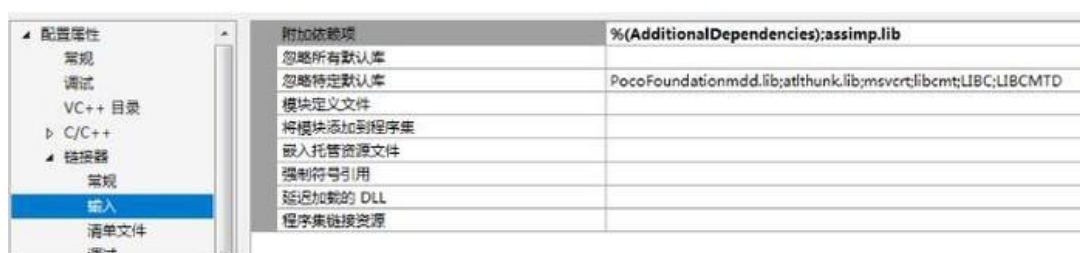
步骤2、设置配置和平台这两个选项,比如你的电脑是64位的就选x64,否则选Win32;选择【C/C++】-【常规】-【附加包含目录】-【编辑】,把自己的文件路径附加进去;



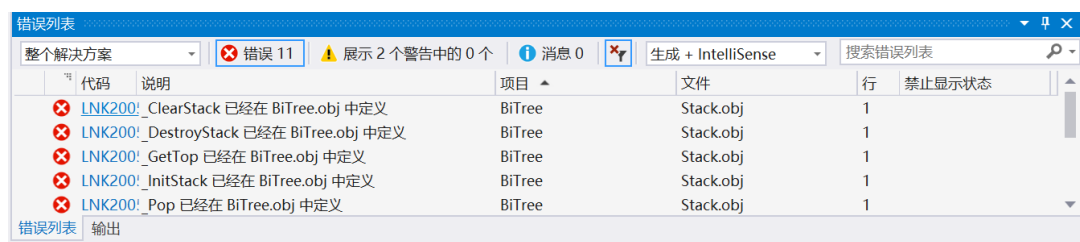
步骤3、【连接器】-【常规】-【附加库目录】-【编辑】,把自己的.dll 库文件路径附加进去;



步骤 4、【连接器】-【输入】-【附加依赖项】-【编辑】，把自己的库文件附加进去；



## 2. 出现错误



可能的原因和解决方法：

通常情况下，此错误表示您已经损坏了一个定义规则，该规则只允许在给定对象文件中对任何已使用的模板、函数、类型或对象进行一种定义，对外部可见对象或函数的整个可执行文件只能有一个定义。

在 BiTree.c 中包括头文件

```
#include "BiTree.h"
#include "LinkQueue.h"
#include "LinkQueue.c"
#include "Stack.h"
#include "Stack.c"
#include "Status.h"
#include <stdio.h>
#include <malloc.h>
```

删除#include "Stack.c"即可

最后查看项目的.vcxproj 和.vcxproj.filters 终于发现问题，原来 VS 对.h 和.c 是区别对待的：

.c 文件始终会被编译成目标文件的，即 obj 文件，.h 文件不会被编译。

生成的目标文件最后全部都会被链接。

解析一下整个 VS2010 编译过程：

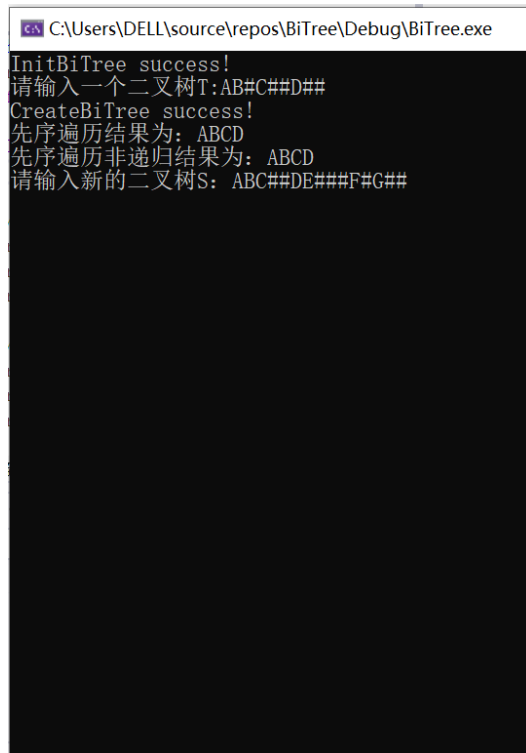
先是预处理，b.c 中的语句被替换成 a.c；

编译 a.c 和 b.c 成目标文件 a.obj 和 b.obj；（此时在 b.obj 中已经包含了 a.c 中函数的声明定义）

链接 a.obj 和 b.obj。（LNK1169）

## 3. 出现错误





```
C:\Users\DELL\source\repos\BiTree\Debug\BiTree.exe
InitBiTree success!
请输入一个二叉树T:AB#C##D##
CreateBiTree success!
先序遍历结果为: ABCD
先序遍历非递归结果为: ABCD
请输入新的二叉树S: ABC##DE###F#G##
```

创建二叉树后想再创建一个新的二叉树时，按回车键后无法继续运行原因是换行符未被吸收。

由此在 CreateBitree 后添加 getchar 函数  
修改后如下：

```
//CreateBiTree test
char ch1, ch2, ch3;
printf("请输入一个二叉树T:");
if (CreateBiTree(&T))
    printf("CreateBiTree success!\n");
else
    printf("CreateBiTree unsucccess!\n");
ch1 = getchar();//吸收换行符
```