華中科技大学

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级：

学　　号：

姓　　名：

指导教师：

报告日期：2019 年 12 月 21 日

网络空间安全学院

目 录

1 基于链式存储结构的线性表实现

1.1 问题描述

采用单链表作为线性表的物理结构，实现线性表的基本运算.

1、要求构造一个具有菜单的功能演示系统。其中，在主程序中完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示。

2、演示系统实现线性表的文件形式保存。其中，①需要设计文件数据记录格式，以高效保存线性表数据逻辑结构(D,{R})的完整信息；②需要设计线性表文件保存和加载操作合理模式。

3、（选做）演示系统可选择实现多个线性表管理。

1.1.1 线性表的基本概念

线性表是最常用且最简单的一种数据结构，即 n 个数据元素的有限序列。线性表中元素的个数 n 定义为线性表的长度，n=0 时成为空表。在非空表中的每个数据元素都有一个确定的位置，如 a1 是第一个数据元素，an 是最后一个数据元素，ai 是第 i 个数据元素。线性表的存储结构分为线性存储和链式存储。

1.1.2 逻辑结构与基本运算

线性表的数据逻辑结构定义如下:

```
ADT List {
    数据对象：D= {ai|ai∈ElemSet，i=1，2，……，n，n≥0}
    数据关系：R1= { <ai-1，ai> | ai-1，ai∈D，i=2，……，n}
}
```

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等 12 种基本运算，具体运算功能定义如下。

⑴初始化表：函数名称是 InitaList(L)；初始条件是线性表 L 不存在已存在；操作结果是构造一个空的线性表。

⑵销毁表：函数名称是 DestroyList(L)；初始条件是线性表 L 已存在；操作结果是销毁线性表 L。

⑶清空表：函数名称是 ClearList(L)；初始条件是线性表 L 已存在；操作结果是将 L 重置为空表。

⑷判定空表：函数名称是 ListEmpty(L)；初始条件是线性表 L 已存在；操作结果是若 L 为空表则返回 TRUE,否则返回 FALSE。

⑸求表长：函数名称是 ListLength(L)；初始条件是线性表已存在；操作结果是返回 L 中数据元素的个数。

⑹获得元素：函数名称是 GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用 e 返回 L 中第 i 个数据元素的值。

⑺查找元素：函数名称是 LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回 L 中第 1 个与 e 满足关系 compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为 0。

⑻获得前驱：函数名称是 PriorElem(L,cur_e,pre_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义。

⑼获得后继：函数名称是 NextElem(L,cur_e,next_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继，否则操作失败，next_e 无定义。

⑽插入元素：函数名称是 ListInsert(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)+1；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

⑾删除元素：函数名称是 ListDelete(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值。

⑿遍历表：函数名称是 ListTraverse(L,visit())，初始条件是线性表 L 已存在；操作结果是依次对 L 的每个数据元素调用函数 visit()。

1.2 系统设计

1.2.1 数据物理结构

链式线性表的数据物理结构如下：

```
typedef struct LNode{    //链式线性表结点的结构体定义
    ElemType data;    //数据元素
    struct LNode* next;  //指向下一结点的指针
}*Link;

typedef struct { //链式线性表整体的结构体定义
    Link head;//链表头指针
    int len;  //链表的长度（不包括头结点）
}LinkList;
```

要实现同时对多个链表管理，只需定义一个结构体指针数组（每个指针指向一个链表）即可：

```
LinkList* L[LISTMAXNUM] = { NULL };
```

1.2.2 系统功能

系统包括 14 个功能，分别为：

1.初始化链表 InitList	2.销毁链表 DestroyList
3.清空链表 ClearList	4.判断空表 ListEmpty
5.求表长 ListLength	6.获得元素 GetElem
7.查找元素 LocateElem	8.获得前驱 PriorElem
9.获得后继 NextElem	10.插入元素 ListInsert
11.删除元素 ListDelete	12.遍历链表 ListTraverse
13.将数据存入文件 SaveElem	14.从文件读取数据 LoadElem

辅助功能：

1.创建链表 CreateList	2.判断链表存在 ListExist
3.比较函数 compare	4.访问数据 visit

1.2.3 系统结构

将菜单演示和用户选择写入到 while 循环中，用 op 获取用户的选择，op 初始化为 1，以便第一次能进入循环。进入循环后系统首先显示功能菜单及当前已经建立的链表名称；然后用户输入分别代表线性表一个基本运算的函数序号 1-14、以及退出系统 0,在主函数中通过 if 或 switch 语句对应到相应的链表功能，其中，除 1、14 外其他链表功能需线性表存在，同时若建立的链表数大于 1，则用户还需输入需要操作的链表的序号。执行完该功能后，继续执行 while 循环，直至用户输入 0 退出当前演示系统。演示系统结构如图 1-1：

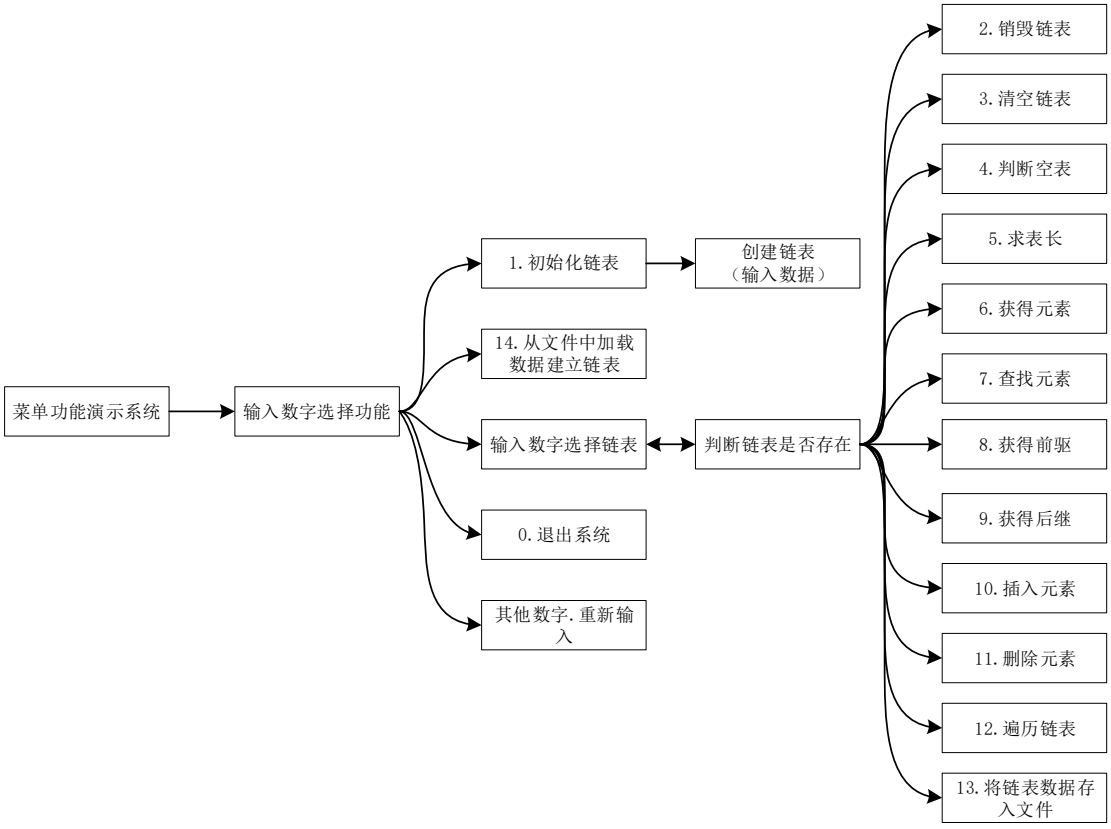

图 1-1 线性表系统设计结构图

1.3 系统实现

1.3.1 主函数实现及代码

1.主函数功能：

输出基本的界面，包括演示系统的 14 个链式线性表功能以及当前已建立的

链表名称。用户输入功能选项，主函数进行对应子函数的预处理后调用子函数实现子函数功能。

2.具体代码：

```
void main(void) {
    LinkList* L[LISTMAXNUM] = { NULL };        //链表指针数组
    char tempname[20];       //临时名称
    int op = 1;
    int temp;
    int oplist;       //选择的链表序号
    int count=num;        //临时建立链表的次数（计数器）
    ElemType elemdest,elemsrc;      //源元素，目的元素

    while (op)
    {
        system("cls");    printf("\n\n");
        printf("      Menu for Linear Table On Chain Structure \n");
        printf("-------------------------------------------------\n");
        printf("        1. InitList         8. PriorElem\n");
        printf("        2. DestroyList      9. NextElem \n");
        printf("        3. ClearList        10.ListInsert \n");
        printf("        4. ListEmpty        11.ListDelete\n");
        printf("        5. ListLength       12.ListTrabverse\n");
        printf("        6. GetElem          13.SaveElem \n");
        printf("        7. LocateElem       14.LoadElem\n");
        printf("        0. Exit\n");
        printf("-------------------------------------------------\n");
        printf("Active List:\t");       //显示当前已建立的链表
        if (num)
```

```
            for (int i = 0; i < num; i++)
            {
                if ((i + 1) % 4 == 0)
                    putchar('\n');
                printf("%d: %s\t", i, listname[i]);
            }
        printf("\n-------------------------------------------------
\n");
        printf("    请选择你的操作[0~14]:");
        scanf("%d", &op);

        if (op == 1)
        {   //输入 1 则初始化链表同时创建链表输入数据
            if              (InitList(&L[num%LISTMAXNUM])              &&
CreateList(L[num%LISTMAXNUM]))
            {   //若建立链表成功对链表临时命名
                memset(tempname, 0, 20);
                strcat(tempname, "temp");
                _itoa(count, tempname + 4, 10);
                strcpy(listname[num%LISTMAXNUM], tempname);
                num = (num >= LISTMAXNUM ? LISTMAXNUM : num + 1);
                //若已建立链表未超过最大数目：建立链表数+1，否则保持最大
//数目不变（超过最大数目的链表一直覆盖第一个链表）
                count++;  //计数器+1
                printf("\n 线性表创建成功！\n");
            }
            else
                printf("\n 线性表创建失败！\n");
            system("pause");
```

```
}
else if (op == 14)
{  //输入 14 加载文件中的数据
   printf("input file name: ");
   scanf("%s", tempname);
   if (LoadElem(L,num%9,tempname))
   {  //加载数据成功则对链表命名为文件名
      strcpy(listname[num % LISTMAXNUM], tempname);
      num = (num >= LISTMAXNUM ? LISTMAXNUM : num+1);
      printf("\n 载入数据成功!\n");
   }
   else
      printf("\nFalse!\n");
   system("pause");
}
else if(op>1&&op<14)
{  //输入其他选项
   if (num > 1)
   {  //若已建立链表数＞1 则操作前需要确定操作链表的序号
      printf("请输入操作链表序号：");
      scanf("%d", &oplist);
   }
   else   //若仅有一个链表，则该链表默认为操作的链表
      oplist = 0;

   if (oplist<0||oplist>=num||!ListExist(L[oplist]))
   {  //检查输入操作链表的序号是否合法 以及操作的链表是否存在
      printf("\nList is NOT EXIST!\n");
      system("pause");
```

```
}
else   //链表序号合法且链表存在
    switch (op)
    {
    case 2:       //输入 2 销毁链表
    if (DestoryList(L,oplist))
        printf("\n 线性表已删除！\n");
    else
        printf("\nFalse!\n");
    system("pause");
    break;

    case 3:       //输入 3 清空链表
        if (ClearList(L[oplist]))
            printf("\n 线性表已清空！\n");
        else
            printf("\nFalse!\n");
        system("pause");
        break;

    case 4:       //判断链表是否为空
        if (ListEmpty(*L[oplist]))
            printf("\nList is empty!\n");
        else
            printf("\nList is NOT empty!\n");
        system("pause");
        break;

    case 5:       //输出链表长度（元素个数）
```

```
                printf("\nThe length of the list is:%d\n",
ListLength(*L[oplist]));
                system("pause");
                break;

            case 6:       //获取链表某元素
                printf("\n 请输入获取的元素序号：");
                scanf("%d", &temp);
                if (GetELem(*L[oplist], temp, &elemdest))
                    printf("\n 获取成功：%d\n", elemdest);
                else
                    printf("\n 获取失败！\n");
                system("pause");
                break;

            case 7:       //根据条件比对元素输出序号
                printf("\n 请输入待比对数据：");
                scanf("%d", &elemsrc);
                printf("\n 第一个满足的数据序号： %d\n",
LocateElem(*L[oplist], elemsrc, compare));
                system("pause");
                break;

            case 8:       //输出某元素的前驱
                printf("\n 请输入一个数据元素：");
                scanf("%d", &elemsrc);
                if (PriorElem(*L[oplist], elemsrc, &elemdest))
                    printf("\n 获取成功：%d\n", elemdest);
                else
```

```
            printf("\n 获取失败! \n");
        system("pause");
        break;

    case 9:        //输出某元素的后件
        printf("\n 请输入一个数据元素：");
        scanf("%d", &elemsrc);
        if (NextElem(*L[oplist], elemsrc, &elemdest))
            printf("\n 获取成功：%d\n", elemdest);
        else
            printf("\n 获取失败! \n");
        system("pause");
        break;

    case 10:       //插入元素
        printf("\n 请输入插入元素序号：");
        scanf("%d", &temp);
        printf("\n 请输入插入元素数值：");
        scanf("%d", &elemsrc);
        if (ListInsert(L[oplist], temp, elemsrc))
            printf("\n 插入成功!\n");
        else
            printf("\nFalse!\n");
        system("pause");
        break;

    case 11:       //删除元素
        printf("\n 请输入删除元素序号：");
        scanf("%d", &temp);
```

```
            if (ListDelete(L[oplist], temp, &elemdest))
                printf("\n 删除成功: Delete %d\n", elemdest);
            else
                printf("\nFalse!\n");
            system("pause");
            break;

        case 12:      //遍历链表
            if (!ListTraverse(*L[oplist], visit))
                printf("线性表是空表! \n");
            system("pause");
            break;

        case 13:      //将链表中的元素存入到文件中
            if(SaveElem(*L[oplist]))
                printf("\n 保存数据成功!\n");
            else
                printf("\nFalse!\n");
            system("pause");
            break;
        }
    }
    else if (op)  //输入其他数字
    {
        printf("\n 输入错误!\n");
        system("pause");
    }
}//end of while
```

```
    for (int i = 0; i < num; i++)       //退出系统前销毁所以已建链表
        DestoryList(L, i);
    printf("欢迎下次再使用本系统！\n");
}//end of main()
```

1.3.2 子函数实现及代码

1.初始化链表 InitList

操作结果：构造一个空的有表头结点的链表 L

操作步骤：给 L 分配一个链表结构体 LinkList 大小的空间，然后创建链表的头结点，并置头结点的 next 指针为空，置链表长度为 0。成功返回 OK，溢出返回 OVERFLOW。

具体代码：

```
status InitList(LinkList** L)
{   //初始化链表：建立链表结构体并创建头结点
    (*L) = (LinkList*)malloc(sizeof(LinkList));
    (*L)->head = (Link)malloc(sizeof(struct LNode)); //带创建头节点
    (*L)->head->next = NULL;
    if (!(*L)->head)
        return OVERFLOW;
    (*L)->len = 0;
    return OK;
}
```

2.销毁链表 DestroyList

初始条件：链表已存在

操作结果：销毁链表 L[oplist]

操作步骤：循环释放该链表结点，然后销毁链表结构体。若已建立链表数目大于 1，则须将销毁链表后的链表前移，最后建立的链表数减 1。

具体代码：

```
status DestoryList(LinkList** L, int oplist)
{  //销毁链表L[oplist]
    Link cur, next;
    cur = L[oplist]->head;
    while (cur)       //销毁链表结点
    {
        next = cur->next;
        free(cur);
        cur = next;
    }
    free(L[oplist]); //销毁链表结构体
    for (; oplist < num - 1; oplist++)
    {
        L[oplist] = L[oplist + 1];      //将销毁链表后的链表前移
        strcpy(listname[oplist], listname[oplist + 1]);      //销毁链
//表后的链表名前移
    }
    memset(listname[oplist], 0, 20);   //最后的链表名清空
    num--;     //已建立链表数-1
    return OK;
}
```

3.清空链表ClearList

初始条件：链表已存在

操作结果：将链表所有存放数据元素的结点释放，保留链表结构体和链表头结点

操作步骤：循环释放链表存放数据元素的结点，然后将链表长度置0。

具体代码：

```
status ClearList(LinkList* L)
{   //清空链表 L：使链表的元素结点释放，保留头结点
    Link cur, next;
    cur = L->head->next;
    while (cur)       //释放元素结点
    {
        next = cur->next;
        free(cur);
        cur = next;
    }
    L->head->next = NULL;
    L->len = 0;       //当前链表长度置 0
    return OK;
}
```

4.判断空表 ListEmpty

初始条件：链表已存在

操作结果：若链表为空表，则返回 TRUE，否则返回 FALSE

操作步骤：判断头结点的 next 指针是否为空，若为空则为空表，若不为空则不为空表。

具体代码：

```
status ListEmpty(LinkList L)
{   //检测链表 L 是否为空表：TRUE 为空表，FLASE 表不为空
    if (L.head->next)
        return FALSE;
    return TRUE;
}
```

5.求链表长度 ListLength

初始条件：链表已存在

操作结果：返回链表中的元素结点个数

操作步骤：返回链表结构体 len 属性的值

具体代码：

```
int ListLength(LinkList L)
{   //返回链表 L 的长度
    return L.len;
}
```

6.获得元素 GetElem

初始条件：链表 L 已存在

操作结果：用 e 返回 L 中第 i 个元素的值

操作步骤：先检测 i 的值是否合法，不合法返回 FALSE 后遍历链表找到对应元素，将其值赋值给 e。

具体代码：

```
status GetELem(LinkList L, int i, ElemType* e)
{   //获取链表 L 第 i 个元素并存入 e 中
    Link temp;
    int j;
    if (i <= 0 || i > L.len) //i 的合法性检测
        return ERROR;
    for (j = 1, temp = L.head->next; j < i; temp = temp->next,
j++);//使 temp 指向待访问元素的结点
    *e = temp->data;//元素赋值给 e
    return OK;
}
```

7.查找元素 LocateElem

初始条件：链表 L 已存在，compare()是数据元素判断函数

操作结果：返回 L 中第一个与 e 满足关系 compare()的数据元素的位序，若不存在这样的数据元素则返回 0

操作步骤：循环遍历列表并用 i 来计数至查找到满足 compare()的元素并返回当前序号 i，若遍历至表尾未找到满足关系额元素则返回 0。

具体代码：

```
int LocateElem(LinkList L, ElemType e, status compare(struct LNode, ElemType))
{   //根据比较函数 compare 返回满足条件的第一个元素的序号
    Link temp;
    int i;
    for (i = 1, temp = L.head->next; temp; temp = temp->next, i++)
        if (compare(*temp, e))   //遍历链表并对比较
            return i;
    return 0;
}
```

8.获得前驱 PriorElem

初始条件：链表 L 已存在

操作结果：若 cur_e 是 L 的数据元素且不是第一个，则它的前驱赋值给 pre_e 并返回 OK，否则返回 ERROR

操作步骤：先检测链表 L 是否不为空表以及 cur_e 不是第一个元素，不满足返回 ERROR；然后用 pre、cur 指针遍历链表查找元素 cur_e，当 cur 找到 cur_e 元素时，将 pre 所指元素赋值给 pre_e，返回 OK，未找到则返回 ERROR。

具体代码：

```
status PriorElem(LinkList L, ElemType cur_e, ElemType* pre_e)
{   //返回链表 L 中元素为 cur_e 的前驱元素到 pre_e 中
    Link pre = L.head, cur = pre->next;
    if (!cur || cur_e == cur->data)     //检测链表是否为空表，同时
//cur_e 不为第一个元素
```

```
        return ERROR;
    for (; cur; pre = cur, cur = cur->next)
        if (cur_e == cur->data)      //判断是否为 cur_e
        {
            *pre_e = pre->data;
            return OK;
        }
    return ERROR; //未找到返回 ERROR E
}
```

9.获得后继 NextElem

初始条件：链表 L 已存在

操作结果：若 cur_e 是 L 的数据元素且不为最后一个，则它的后继赋值给 next_e 并返回 OK，否则返回 ERROR

操作步骤：先检测链表 L 是否不为空表，空表返回 ERROR；然后用 cur、next 指针遍历链表至倒数第二个元素查找元素 cur_e，当 cur 找到 cur_e 元素时，将 next 所指元素赋值给 next_e，返回 OK，未找到则返回 ERROR。

具体代码：

```
status NextElem(LinkList L, ElemType cur_e, ElemType* next_e)
{   //返回链表 L 中元素为 cur_e 的后继元素到 next_e 中
    Link cur = L.head->next, next;
    if (!cur) //检测链表是否为空表
        return ERROR;
    next = cur->next;
    for (; next; cur = next, next = next->next)  //遍历链表至倒数第二
//个元素
        if (cur_e == cur->data)      //判断是否为 cur_e
        {
            *next_e = next->data;
```

```
            return OK;
        }
    return ERROR; //未找到返回 ERROR
}
```

10.插入元素 ListInsert

初始条件：链表 L 已存在

操作结果：在链表 L 第 i（1≤i≤len+1）个位置前插入新的数据元素 e，链表长度加 1

操作步骤：检测序号 i 的合法性，不合法返回 ERROR；然后遍历链表至待插入位置，插入结点并赋值，最后链表长度加 1，返回 OK。

具体代码：

```
status ListInsert(LinkList* L, int i, ElemType e)
{   //在链表 L 的第 i（1<=i<=len+1）个元素前插入元素 e
    Link temp, New;
    int j;
    if (i<1 || i>L->len + 1) //检测 i 的合法性，不合法返回 ERROR
    return ERROR;
    for (j = 1, temp = L->head; j < i; j++, temp = temp->next);
//遍历到待插入位置
    New = (Link)malloc(sizeof(struct LNode));     //新建结点
    if (!New)
        return ERROR;
    New->data = e;        //结点赋值
    New->next = temp->next;      //结点链接到链表
    temp->next = New;
    L->len++; //链表长度+1
    return OK;
}
```

11.删除元素 ListDelete

初始条件：链表 L 已存在且非空

操作结果：删除 L 的第 i（1≤i≤len）个元素，并赋值给 e，链表的长度减 1

操作步骤：先检测 i 的合法性，不合法返回 ERROR，然后遍历链表至待删除结点，释放结点，链表长度减 1，返回 OK。

具体代码：

```
status ListDelete(LinkList* L, int i, ElemType* e)
{   //删除链表 L 的第 i 个元素并将其值赋给 e
    Link temp, dest;
    int j;
    if (i<1 || i>L->len) //检测 i 的合法性，不合法返回 ERROR
        return ERROR;
    for (j = 1, temp = L->head; j < i; j++, temp = temp->next);
//遍历到待删除位置
    dest = temp->next;
    *e = dest->data;      //将删除值赋给 e
    temp->next = dest->next;
    free(dest);        //释放结点
    L->len--;      //链表长度-1
    return OK;
}
```

12.遍历链表 ListTraverse

初始条件：链表已存在

操作结果：若链表不为空，则依次对链表 L 调用函数 visit()，返回 OK；若链表为空则返回 ERROR

操作步骤：先检测链表是否为空，为空则返回 ERROR；然后遍历链表每个结

点使用 visit()函数，返回 OK。

具体代码：

```
status ListTraverse(LinkList L, void (*visit)(ElemType))
{   //使用 vist 函数遍历链表 L
    Link temp;
    if (!L.head->next)
        return ERROR;
    for (temp = L.head->next; temp; temp = temp->next)
        visit(temp->data);
    putchar('\n');
    return OK;
}
```

13.将数据存入文件 SaveElem

初始条件：链表已存在

操作结果：将链表 L 中的元素保存到文件中

操作步骤：自行输入要保存的文件名，写方式打开文件，若失败返回 ERROR；然后遍历链表，将每个结点中的元素逐一写入到文件中，返回 OK。

具体代码：

```
status SaveElem(LinkList L)
{   //将链表 L 中的元素存入文件中
    Link temp;
    FILE* fp;
    char filename[30];
    printf("input file name: ");        //输入文件名
    scanf("%s", filename);
    if ((fp = fopen(filename, "wb")) == NULL)
    {
        printf("File open error!\n ");
```

```
        return ERROR;
    }
    for (temp = L.head->next; temp; temp = temp->next)
        fwrite(&temp->data, sizeof(ElemType), 1, fp);
    //将每个结点中的元素写入文件中
    fclose(fp);
    return OK;
}
```

14.从文件中读取数据 LoadElem

操作结果：创建一链表 L[num]并将文件 filename 中的数据作为元素结点存入链表中

操作步骤：先检测当前 L[num]链表是否存在，不存在则初始化链表，存在则清空链表；然后以读方式打开文件 filename，打开失败返回 ERROR；然后逐一建立链表结点并将读取的一个数据存入到结点中，返回 OK。

具体代码：

```
status LoadElem(LinkList** L, int num, char* filename)
{   //加载文件名为 filename 中的元素到链表 L[num]中
    FILE* fp;
    Link end, New;
    ElemType temp;
    if (!ListExist(L[num]))      //若链表 L[num]不存在则初始化
        InitList(&L[num]);
    else
        ClearList(L[num]);       //若链表 L[num]已存在则清空链表
    if ((fp = fopen(filename, "rb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
```

```
    }
    end = L[num]->head;
    while (fread(&temp, sizeof(ElemType), 1, fp))
    {   //每次读入一个数据并存入到建立的链表结点中
        New = (Link)malloc(sizeof(struct LNode));
        New->data = temp;
        New->next = end->next;
        end->next = New;
        end = New;
        L[num]->len++;
    }
    return OK;
}
```

15 其他函数

（1）创建链表并输入数据 CreateList

初始条件：链表已存在

操作结果：空链表 L 输入数据

操作步骤：循环输入数据元素，若第一次输入为 0，则返回 ERROR；之后若输入不为 0 则创建新的链表结点并存入数据，若输入为 0 则结束建立结点，返回 OK。

具体代码：

```
status CreateList(LinkList* L)
{   //创建链表：创建链表 L 的节点并输入数据（输入 0 时结束创建）
    Link end, New;
    int e;
    end = L->head;
    printf("Input Element:");
    scanf("%d", &e);
```

```
    if (!e)        //开始输入 0 则不创建结点子函数结束
        return ERROR;
    while (e)
    {
        New = (Link)malloc(sizeof(struct LNode)); //创建结点
        if (!New)
            return OVERFLOW;
        New->data = e;        //数据赋值
        New->next = end->next;
        end->next = New;
        end = New;
        L->len++; //链表长度+1
        scanf("%d", &e);
    }
    return OK;
}
```

（2）判断链表存在 ListExist

操作结果：若链表 L 存在返回 TRUE，否则返回 FALSE

操作步骤：检测 L 指向是否为空，不为空则链表存在返回 TRUE，若为空则链表为空返回 FALSE。

具体代码：

```
status ListExist(LinkList* L)
{   //检测链表 L 是否存在，FALSE 不存在，TRUE 存在
    if (!L)
        return FALSE;
    return TRUE;
}
```

（3）比较函数 compare

操作结果：若结点中的元素大于 e 则返回 TRUE，否则返回 FALSE（可自定义）

具体代码：

```
status compare(struct LNode node, ElemType e)
{   //比较函数：若结点中的元素大于 e 则返回 TRUE，否则返回 FALSE（可自
//定义）
    if (node.data > e)
        return TRUE;
    else
        return FALSE;
}
```

（4）访问数据 visit

操作结果：输出该数据元素

具体代码：

```
void visit(ElemType e)
{   //访问（输出）结点 node 的元素
    printf("%d\t", e);
}
```

1.4 实验小结

通过本次实验，个人自己编写了一个具有比较完善功能的演示系统，加深了个人对线性表的链式存储结构的理解和认识，并通过自己的编程切身体会到如何编写链表的初始化、创建、增删遍历等基本操作的函数，同时也加深了自己对文件读取函数的使用，以及如何利用指针数组管理多个链表，提高了自己对 C 语言语法中指针、结构体等的使用能力。

在实验过程中，我也深刻意识到，相比于一个简单的程序，一个比较完整的系统需要考虑更多的因素，需要在一些细节的地方做好处理，尤其是像结构越来越复杂的情况下，如何合理的使用指针、二级指针、取地址符&便显得额外重要，也要注意动态分配的内存空间的回收问题，同时也学会了如何使用函数名作为另一个函数的形参，等等。提高了自己对一个较为复杂的程序的分析设计安排及编

写能力。

2 基于二叉链表的二叉树实现

2.1 问题描述

采用二叉链表作为二叉树的物理结构，实现二叉链表的基本运算。

1、要求构造一个具有菜单的功能演示系统。其中，在主程序中完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示。

2、（选做）演示系统可选择实现二叉树的文件形式保存。

3、（选做）演示系统可选择实现多个二叉树管理。

2.1.1 二叉树的基本概念

二叉树是一种树型结构，即 n 个结点的有限集，它的特点是每个结点至多只有两棵子树（即二叉树中不存在度大于 2 的结点），并且，二叉树的子树有左右之分，其次序不能任意颠倒。

2.1.2 逻辑结构与基本运算

抽象数据类型二叉树的定义如下：

ADT BinaryTree {

数据对象 D：D 是具有相同特性的数据元素的集合。

数据关系 R:

若 D=Φ，则 R=Φ，称 BinaryTree 为空二叉树；

若 D≠Φ，则 R={H},H 是如下二元关系：

（1） 在 D 中存在唯一的成为根的数据元素 root，它在关系 H 中无前驱；

（2） 若 D-{root}≠Φ,则存在 D-{root}={D1,Dr},且 D1∩Dr=Φ；

（3） 若 D1≠Φ，则 D1 中存在唯一的元素 X1，<root,X1>∈H，且存在 D1 上的关系 H1 包含于 H；若 Dr≠Φ,则 Dr 中存在唯一的元素 Xr，<root,Xr>∈H,且存在 Dr 上的关系属于 H；

（4） （D,{H1}）是一棵符合本定义的二叉树,称为根的左子树,（Dr，

{Hr}）是一棵符合本定义的二叉树，称为根的右子树。

}

依据最小最小完备性和常用性相结合的原则，以函数形式定义了二叉树的初始化、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树和求二叉树深度等20种基本运算，具体运算功能定义如下：

⑴初始化二叉树：函数名称是 InitBiTree(T)；初始条件是二叉树 T 不存在；操作结果是构造空二叉树 T。

⑵销毁二叉树：函数名称是 DestroyBiTree(T)；初始条件是二叉树 T 已存在；操作结果是销毁二叉树 T。

⑶创建二叉树：函数名称是 CreateBiTree(T,definition)；初始条件是 definition 给出二叉树 T 的定义；操作结果是按 definition 构造二叉树 T。

⑷清空二叉树：函数名称是 ClearBiTree (T)；初始条件是二叉树 T 存在；操作结果是将二叉树 T 清空。

⑸判定空二叉树：函数名称是 BiTreeEmpty(T)；初始条件是二叉树 T 存在；操作结果是若 T 为空二叉树则返回 TRUE，否则返回 FALSE。

⑹求二叉树深度：函数名称是 BiTreeDepth(T)；初始条件是二叉树 T 存在；操作结果是返回 T 的深度。

⑺获得根结点：函数名称是 Root(T)；初始条件是二叉树 T 已存在；操作结果是返回 T 的根。

⑻获得结点：函数名称是 Value(T, e)；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是返回 e 的值。

⑼结点赋值：函数名称是 Assign(T,&e, value)；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是结点 e 赋值为 value。

⑽获得双亲结点：函数名称是 Parent(T, e) ；初始条件是二叉树 T 已存在，e 是 T 中的某个结点；操作结果是若 e 是 T 的非根结点，则返回它的双亲结点指针，否则返回 NULL。

⑾获得左孩子结点：函数名称是 LeftChild(T, e)；初始条件是二叉树 T 存在，e 是 T 中某个节点；操作结果是返回 e 的左孩子结点指针。若 e 无左孩子，则返回 NULL。

⑿获得右孩子结点：函数名称是 RightChild(T, e)；初始条件是二叉树 T 已存在，e 是 T 中某个结点；操作结果是返回 e 的右孩子结点指针。若 e 无右孩子，则返回 NULL。

⒀获得左兄弟结点：函数名称是 LeftSibling(T, e)；初始条件是二叉树 T 存在，e 是 T 中某个结点；操作结果是返回 e 的左兄弟结点指针。若 e 是 T 的左孩子或者无左兄弟，则返回 NULL。

⒁获得右兄弟结点：函数名称是 RightSibling(T, e)；初始条件是二叉树 T 已存在，e 是 T 中某个结点；操作结果是返回 e 的右兄弟结点指针。若 e 是 T 的右孩子或者无有兄弟，则返回 NULL。

⒂插入子树：函数名称是 InsertChild(T, p, LR, c)；初始条件是二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0 或 1，,非空二叉树 c 与 T 不相交且右子树为空；操作结果是根据 LR 为 0 或者 1，插入 c 为 T 中 p 所指结点的左或右子树，p 所指结点的原有左子树或右子树则为 c 的右子树。

⒃删除子树：函数名称是 DeleteChild(T. p. LR)；初始条件是二叉树 T 存在，p 指向 T 中的某个结点，LR 为 0 或 1。 操作结果是根据 LR 为 0 或者 1，删除 c 为 T 中 p 所指结点的左或右子树。

⒄前序遍历：函数名称是 PreOrderTraverse(T, Visit())；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果：先序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⒅中序遍历：函数名称是 InOrderTraverse(T, Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是中序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⒆后序遍历：函数名称是 PostOrderTraverse(T, Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是后序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

⒇按层遍历：函数名称是 LevelOrderTraverse(T, Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是层序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

2.2 系统设计

2.2.1 数据物理结构

二叉树的数据物理结构如下：

```
typedef struct BiNode { //二叉树结点结构体
    KeyElem key;  //关键字
    ElemType data;    //数据
    struct BiNode* lchild, *rchild;    //左右孩子指针
}BiTNode,*BiTree;

typedef struct BiTList {    //二叉树整体结构体
    BiTree Root;  //二叉树根结点
}BiTList;
```

要实现同时对多个二叉树管理，只需定义一个结构体指针数组（每个指针指向一个二叉树）即可：

```
BiTList* T[TREEMAXNUM] = { NULL };    //二叉树指针数组
```

其他辅助数据物理结构：

```
typedef struct QNode {
    Qelem data;       //数据
    struct QNode* next;      //指针域
}QNode,*QPtr; //队列结点结构体

typedef struct {
    QPtr front, rear;    //队列队首、队尾指针
}LinkQueue;       //队列整体结构体
```

2.2.2 系统功能

系统包括 22 个功能，分别为：

1.初始化二叉树 InitBiTree　　2.销毁二叉树 DestroyBiTree
3.创建二叉树 CreateBiTree　　4.清空二叉树 ClearBiTree
5.判断空二叉树 BiTreeEmpty　　6.求二叉树深度 BiTreeDepth
7.获得根结点 Root　　8.获得结点 Value
9.结点赋值 Assign　　10.获得双亲结点 Parent
11.获得左孩子结点 LeftChild　　12.获得右孩子结点 RightChild
13.获得左兄弟结点 LeftSibling　　14.获得右兄弟结点 RightSibling
15.插入子树 InsertChild　　16.删除子树 DeleteChild
17.先序遍历 PreOrderTraverse　　18.中序遍历 InOrderTraverse
19.后序遍历 PostOrderTraverse　　20.按层遍历 LevelOrderTraverse
21.将二叉树存入文件 SaveBiTreeElem
22.从文件加载二叉树 LoadBiTreeElem

辅助功能：

1.初始化队列 InitQueue　　2.销毁队列 DestroyQueue
3.元素入队 EnQueue　　4.元素出队 DeQueue
5.获得结点指针 Vertex　　6.访问二叉树结点数据 visit

2.2.3 系统结构

将菜单演示和用户选择写入到 while 循环中，用 op 获取用户的选择，op 初始化为 1，以便第一次能进入循环。

进入循环后系统首先显示功能菜单及当前已经建立的二叉树名称；然后用户输入分别代表二叉树一个基本运算的函数序号 1-22、以及退出系统 0，在主函数中通过 if 或 switch 语句对应到相应的二叉树功能，其中，除 1、22 外其他二叉树功能需二叉树存在，同时若建立的二叉树个数大于 1，则用户还需输入需要操作的二叉树的序号。执行完该功能后，继续执行 while 循环，直至用户输入 0 退

出当前演示系统。演示系统结构如图 2-1：

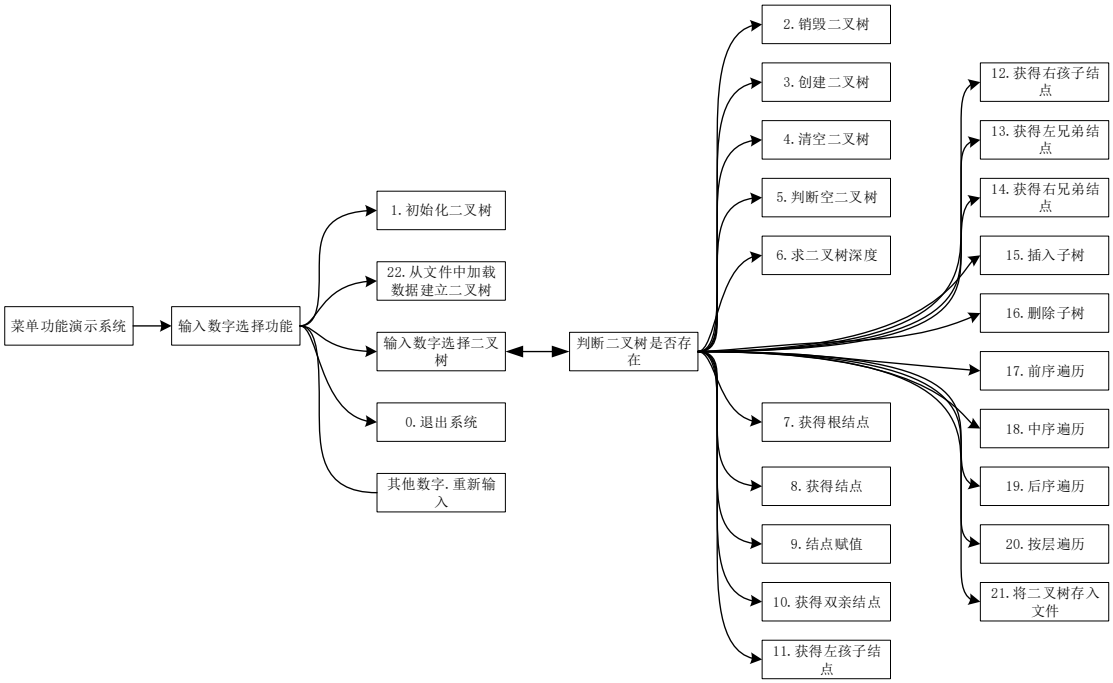

图 2-1 二叉树系统设计结构图

2.3 系统实现

2.3.1 主函数实现及代码

1.主函数功能：

输出基本的界面，包括演示系统的 22 个二叉树功能以及当前已建立的二叉树名称。用户输入功能选项，主函数进行对应子函数的预处理后调用子函数实现子函数功能。

2.具体代码：

```
void main(void) {
    BiTList* T[TREEMAXNUM] = { NULL };      //二叉树指针数组
    BiTree temptree=NULL;     //临时二叉树结点指针
    char tempname[20];        //临时名称
    int op = 1;
    int temp;
    int oplist;        //选择的二叉树序号
```

```
int count = num;     //临时建立二叉树的次数（计数器）
KeyElem tempkey;
ElemType elemdest, elemsrc;     //源元素，目的元素

while (op)
{
    system("cls");   printf("\n\n");
    printf("      Menu for Binary Tree On Chain Structure \n");
    printf("-------------------------------------------------\n");
    printf("        1. InitBiTree        12.RightChild\n");
    printf("        2. DestroyBiTree     13.LeftSibling\n");
    printf("        3. CreatBiTree       14.RightSibling\n");
    printf("        4. ClearBiTree       15.InsertChild\n");
    printf("        5. BiTreeEmpty       16.DeleteChild\n");
    printf("        6. BiTreeDepth       17.PreOrderTraverse \n");
    printf("          7. Root              18.InOrderTraverse\n");
    printf("        8. Value             19.PostOrderTraverse\n");
    printf("       9. Assign            20.LevelOrderTraverse\n");
    printf("        10.Parent            21.SaveBiTreeElem\n");
    printf("        11.LeftChild         22.LoadBiTreeElem\n");
    printf("        0. Exit\n");
    printf("-------------------------------------------------\n");
    printf("Active Tree:\t");       //显示当前已建立的二叉树
    if (num)
        for (int i = 0; i < num; i++)
        {
            if ((i + 1) % 4 == 0)
                putchar('\n');
            printf("%d: %s\t", i, treename[i]);
```

```
        }
    printf("\n-----------------------------------------------
\n");
    printf("    请选择你的操作[0~22]:");
    scanf("%d", &op);

    if (op == 1)
    {   //输入 1 则初始化二叉树
        if (InitBiTree(&T[num % TREEMAXNUM]))
        {   //若建立二叉树成功对链表临时命名
            memset(tempname, 0, 20);
            strcat(tempname, "temp");
            _itoa(count, tempname + 4, 10);
            strcpy(treename[num % TREEMAXNUM], tempname);
            num = (num >= TREEMAXNUM ? TREEMAXNUM : num + 1);
            //若已建立二叉树未超过最大数目：建立二叉树个数+1，否则保
持最大数目不变（超过最大数目的二叉树一直覆盖第一个二叉树）
            count++;  //计数器+1
            printf("\n 二叉树初始化成功! \n");
        }
        else
            printf("\n 二叉树初始化失败! \n");
        system("pause");
    }

    else if (op == 22)
    {   //输入 22 加载文件中的数据
        printf("input file name: ");
        scanf("%s", tempname);
```

```
            if (LoadBiTreeElem(T,num,tempname))
            {   //加载数据成功则对二叉树命名为文件名
                strcpy(treename[num % TREEMAXNUM], tempname);
                num = (num >= TREEMAXNUM ? TREEMAXNUM : num + 1);
                printf("\n 二叉树数据加载成功!\n");
            }
            else
                printf("\nFalse!\n");
            system("pause");
        }
        else if (op > 1 && op < 22)
        {   //输入其他选项
            if (num > 1)
            {   //若已建立链表数>1 则操作前需要确定操作链表的序号
                printf("请输入操作链表序号：");
                scanf("%d", &oplist);
            }
            else   //若仅有一个链表，则该链表默认为操作的链表
                oplist = 0;

            if (oplist < 0 || oplist >= num || !T[oplist])
            {   //检查输入操作二叉树的序号是否合法 以及操作的二叉树是否
存在
                printf("\nBiTree is NOT EXIST!\n");
                system("pause");
            }
            else   //二叉树序号合法且二叉树存在
                switch (op)
                {
```

```
case 2:        //输入 2 销毁二叉树
    DestroyBiTree(T,oplist);
    printf("\n 二叉树已销毁!\n");
    system("pause");
    break;

case 3:        //输入 3 创建二叉树
    T[oplist]->Root = CreateBiTree(1);
    printf("二叉树已建立成功!\n");
    system("pause");
    break;
case 4:        //输入 4 清空二叉树
    if (T[oplist] && T[oplist]->Root)
    {
        ClearBiTree(&T[oplist]->Root);
        printf("\n 二叉树已清空!\n");
    }
    else
        printf("\n 二叉树清空失败! \n");
    system("pause");
    break;

case 5:        //判断二叉树是否为空
    if (BiTreeEmpty(*T[oplist]))
        printf("\nBiTree is empty!\n");
    else
        printf("\nBiTree is NOT empty!\n");
    system("pause");
    break;
```

```
            case 6:       //输出二叉树深度
                printf("\n    二    叉    树    深    度    为    :%d\n",
BiTreeDepth(T[oplist]->Root));
                system("pause");
                break;

            case 7:       //求二叉树根结点并输出
                if (temptree = Root(*T[oplist]))
                {
                    printf("\n 根结点：");
                    visit(temptree->key, temptree->data);
                }
                else
                    printf("\n 根结点为空！\n");
                putchar('\n');
                system("pause");
                break;

            case 8:       //获取结点数据值
                printf("请输入搜索关键字：");
                scanf("%*c%c", &tempkey);
                elemdest = Value(T[oplist]->Root, tempkey);
                if ( elemdest== -1)
                    printf("\n 该结点未找到！\n");
                else
                    printf("\n 该结点已找到，数据值：%d\n", elemdest);
                system("pause");
                break;
```

```
case 9:        //结点赋值
    printf("请输入搜索关键字：");
    scanf("%*c%c", &tempkey);
    printf("\n 请输入新的数据：");
    scanf("%d", &elemsrc);
    if (Assign(T[oplist]->Root, tempkey, elemsrc))
        printf("\n 结点已重新赋值！\n");
    else
        printf("\n 结点赋值失败！\n");
    system("pause");
    break;

case 10:       //获得双亲结点并输出
    printf("请输入搜索关键字：");
    scanf("%*c%c", &tempkey);
    if (temptree = Parent(T[oplist]->Root, tempkey))
    {
        printf("\n 其双亲结点为 ");
        visit(temptree->key, temptree->data);
        putchar('\n');
    }
    else
        printf("\n 结点双亲获得失败！\n");
    system("pause");
    break;

case 11:       //获得左孩子结点并输出
    printf("请输入搜索关键字：");
```

```
        scanf("%*c%c", &tempkey);
        if (temptree = LeftChild(T[oplist]->Root, tempkey))
        {
            printf("\n 其左孩子结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点左孩子获得失败! \n");
        system("pause");
        break;

    case 12:      //获得右孩子结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        if (temptree = RightChild(T[oplist]->Root, tempkey))
        {
            printf("\n 其右孩子结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点右孩子获得失败! \n");
        system("pause");
        break;

    case 13:      //获得左兄弟结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
```

```
            if (temptree = LeftSibling(T[oplist]->Root, tempkey))
            {
                printf("\n 其左兄弟结点为 ");
                visit(temptree->key, temptree->data);
                putchar('\n');
            }
            else
                printf("\n 结点左兄弟获得失败! \n");
            system("pause");
            break;

        case 14:      //获得右兄弟结点并输出
            printf("请输入搜索关键字：");
            scanf("%*c%c", &tempkey);
            if   (temptree   =   RightSibling(T[oplist]->Root,
tempkey))
            {
                printf("\n 其右兄弟结点为 ");
                visit(temptree->key, temptree->data);
                putchar('\n');
            }
            else
                printf("\n 结点右兄弟获得失败! \n");
            system("pause");
            break;

        case 15:      //插入子树
            printf("建立插入子树 c：\n");
            printf("需输入 1 个结点\n 请输入关键字:");
```

```
            scanf("%*c%c", &tempkey);   //输入子树根结点
            if (tempkey == NULLNODE)//若子树根结点为空则不合法
            {
                printf("\n 建立子树不合法！\n");
                system("pause");
                break;
            }

            temptree = (BiTree)malloc(sizeof(BiTNode));      //
若子树根结点不为空继续操作
            if (!temptree)
            {
                printf("建立子树失败！\n");
                system("pause");
                break;
            }
            temptree->key = tempkey;
            printf("请输入数据:");
            scanf("%d", &temptree->data);
            //temptree->father = NULL;//
            temptree->lchild = CreateBiTree(1);   //仅递归创建
子树的左子树，右子树为空
            temptree->rchild = NULL;

            printf("\n 请输入搜索关键字：");
            scanf("%*c%c", &tempkey);
            printf("\n 请输入操作子树(0.左子树/1.右子树):");
            scanf("%d", &temp);
            if (temp != 0 && temp != 1)
```

```
                {
                    printf("\n 输入错误!\n");
                    system("pause");
                    break;
                }
                if  (InsertChild(T[oplist]->Root,  tempkey,  temp,
temptree))
                    printf("\n 插入子树成功! \n");
                else
                    printf("\n 插入子树失败! \n");
                system("pause");
                break;

            case 16:      //删除子树
                printf("\n 请输入搜索关键字: ");
                scanf("%*c%c", &tempkey);
                printf("\n 请输入操作子树(0.左子树/1.右子树):");
                scanf("%d", &temp);
                if (temp != 0 && temp != 1)
                {
                    printf("\n 输入错误!\n");
                    system("pause");
                    break;
                }
                if (DeleteChild(T[oplist]->Root, tempkey, temp))
                    printf("\n 删除子树成功! \n");
                else
                    printf("\n 删除子树失败! \n");
                system("pause");
```

```
        break;

    case 17:
        printf("前序遍历：");
        PreOrderTraverse(T[oplist]->Root,visit);
        putchar('\n');
        system("pause");
        break;

    case 18:
        printf("中序遍历：");
        InOrderTraverse(T[oplist]->Root, visit);
        putchar('\n');
        system("pause");
        break;

    case 19:
        printf("后序遍历：");
        PostOrderTraverse(T[oplist]->Root, visit);
        putchar('\n');
        system("pause");
        break;

    case 20:
        printf("层序遍历：");
        LevelOrderTraverse(T[oplist]->Root, visit);
        putchar('\n');
        system("pause");
        break;
```

```
            case 21:       //将二叉树中的元素存入到文件中
                if (SaveBiTreeElem(*T[oplist]))
                    printf("\n 二叉树数据已保存!\n");
                else
                    printf("\n 二叉树数据保存失败！\n");
                system("pause");
                break;
            }
        }
        else if (op)  //输入其他数字
        {
            printf("\n 输入错误!\n");
            system("pause");
        }
    }//end of while

    for (int i = 0; i < num; i++)       //退出系统前销毁所以已建二叉树
        DestroyBiTree(T,i);
    printf("欢迎下次再使用本系统！\n");
}//end of main()
```

2.3.2 子函数实现及代码

1.初始化二叉树 InitBiTree

操作结果：构造一个空的二叉树结构体 R

操作步骤：给 R 分配一个二叉树结构体 BiTLIst 大小的空间，然后使其根结点指针为空。成功返回 OK，溢出返回 OVERFLOW。

具体代码：

```
status InitBiTree(BiTList** R)
```

```
{   //初始化二叉树 R
    (*R) = (BiTList*)malloc(sizeof(BiTList)); //创建二叉树结构体
    if (!(*R)->Root)
        return OVERFLOW;
    (*R)->Root = NULL;        //二叉树根结点设为空指针
    return OK;
}
```

2.销毁二叉树 DestroyBiTree

初始条件：二叉树已存在

操作结果：销毁二叉树 R[oplist]

操作步骤：先清空二叉树的所有节点，再释放二叉树结构体。若已建立的二叉树个数大于 1，则需将销毁二叉树后的二叉树前移，最后建立的二叉树个数减 1。

具体代码：

```
void DestroyBiTree(BiTList** R,int oplist)
{   //销毁二叉树 R[oplist]
    if(R[oplist]->Root)
        ClearBiTree(&R[oplist]->Root); //先清空二叉树所有结点
    free(R[oplist]); //释放二叉树结构体
    for (; oplist < num - 1; oplist++)
    {
        R[oplist] = R[oplist + 1];      //将销毁二叉树后的二叉树前移
        strcpy(treename[oplist], treename[oplist + 1]);
//销毁二叉树后的二叉树名称前移
    }
    memset(treename[oplist], 0, 20);   //最后二叉树名称清空
    R[oplist] = NULL;     //二叉树指针指向空
    num--;     //已建立二叉树个数-1
```

```
}
```

3.创建二叉树 CreateBiTree

初始条件：二叉树已存在

操作结果：创建一棵二叉树并返回根结点

操作步骤：先输入根结点关键字，若输入为空结点标识则返回空指针；若输入其他值，则创建二叉树结点，并再输入结点的数据。最后递归建立该结点的左右子树。

具体代码：

```
BiTree CreateBiTree(int count)
{   //创建二叉树：count 为当前还需输入的结点个数
    KeyElem e;
    BiTree New = NULL;
    printf("需输入%d 个结点\n 请输入关键字:", count);
    scanf("%*c%c", &e);
    count--;   //需输入结点个数-1
    if (e != NULLNODE)        //若输入的关键字不为空节点标识
    {
        New = (BiTree)malloc(sizeof(BiTNode));     //创建二叉树结点
        if (!New)
            return FALSE;
        New->key = e;      //结点赋值
        printf("请输入数据:");
        scanf("%d", &New->data);
        New->lchild = CreateBiTree(count + 2);     //递归建立子节点
        New->rchild = CreateBiTree(count + 1);
    }
    return New;        //返回二叉树根结点
}
```

4.清空二叉树 ClearBiTree

初始条件：二叉树已存在，且二叉树根结点不为空

操作结果：将二叉树的所有结点释放并置为空

操作步骤：若当前结点有左子树则递归清空左子树；若当前结点有右子树则递归清空右子树，最后释放当前结点并置为空

具体代码：

```
void ClearBiTree(BiTree* T)
{   //清空二叉树 T
    if ((*T)->lchild) //若二叉树 T 的子节点不为空则先递归清空子树
        ClearBiTree(&(*T)->lchild);
    if ((*T)->rchild)
        ClearBiTree(&(*T)->rchild);
    if (!(*T)->lchild && !(*T)->rchild)        //若 T 是叶子结点
    {
        free(*T); //则释放结点内存
        *T = NULL;//指针指向空
    }
}
```

5.判定空二叉树 BiTreeEmpty

初始条件：二叉树 R 已存在

操作结果：若二叉树 R 为空树则返回 TRUE，否则返回 FALSE

操作步骤：检测二叉树 R 的根结点指针是否为空，为空则为空树，不为空则不为空树。

具体代码：

```
status BiTreeEmpty(BiTList R)
{   //判定 R 是否为空二叉树：TRUE 树空，FALSE 树不为空
    if (R.Root)        //若二叉树根结点不为空
```

```
        return FALSE;      //则树不为空
    else        //若二叉树根结点为空
        return TRUE;  //则二叉树为空
}
```

6.求二叉树深度 BiTreeDepth

初始条件：二叉树 T 已存在

操作结果：返回二叉树的深度

操作步骤：先初始化深度为 0，若二叉树的根结点 T 不为空则深度值,加 1；然后分别递归求出其左右子树的深度，将根结点深度加上子树中深度值更大的深度，得到整棵树的深度，返回该值。

具体代码：

```
int BiTreeDepth(BiTree T)
{   //求二叉树 T 的深度：返回二叉树深度的整数
    int dep = 0;  //初始化树的深度为 0
    int i, j;
    if (T) //若 T 的根结点结点不为空
    {
        dep++;      //树的深度+1
        i = BiTreeDepth(T->lchild);      //递归求结点的左右子树深度
        j = BiTreeDepth(T->rchild);
        dep += i > j ? i : j;    //树 T 的深度为原深度+子树深度大的深度
    }
    return dep;        //返回树的深度
}
```

7.获得根结点 Root

初始条件：二叉树 R 已存在

操作结果：返回二叉树 R 根结点的指针

具体代码：

```
BiTree Root(BiTList R)
{   //返回二叉树 R 的根结点
    return R.Root;
}
```

8.获得结点 Value

初始条件：二叉树已存在

操作结果：获得二叉树 T 中关键字为 key 的结点数据值，未找到返回-1

操作步骤：先找到关键字为 key 的结点，若该结点不为空则返回其数据值；否则返回-1。

具体代码：

```
ElemType Value(BiTree T, KeyElem key)
{   //获得结点值：返回二叉树 T 中关键字为 key 的结点的数据值，未找到返回
//-1
    BiTree temp = Vertex(T, key);   //temp 为指向 key 结点的指针
    if (temp)      //若 temp 不为空
        return temp->data;        //则返回结点的数据
    return -1;     //若 temp 为空则返回-1（表示未找到）
}
```

9.结点赋值 Assign

初始条件：二叉树已存在

操作结果:给二叉树 T 中关键字为 key 的结点赋值 value,赋值成功返回 OK，失败返回 ERROR

操作步骤：先找到关键字为 key 的结点，若该结点不为空则重新赋值，返回成功；否则返回失败。

具体代码：

```
status Assign(BiTree T, KeyElem key, ElemType value)
```

```
{   //结点赋值：给二叉树 T 中关键字为 key 的结点赋值 value
    BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
    if (temp)      //若 temp 不为空
    {
        temp->data = value;       //结点的数据重新赋值
        return OK;     //返回成功
    }
    return ERROR;     //未找到返回失败
}
```

10.获得双亲结点 Parent

初始条件：二叉树已存在

操作结果：返回二叉树 T 中关键字为 key 的双亲结点指针，其他情况返回 NULL

操作步骤：若当前节点 T 为空直接返回 NULL；若当前结点 T 不为空，判断是否有左孩子且左孩子关键字为 key，是则返回 T；再判断是否有右孩子且关键字为 key，是则返回 T；若以上两种情况均不满足，则以递归左右子树寻找。

具体代码：

```
BiTree Parent(BiTree T, KeyElem key)
{   //获得双亲结点：返回二叉树 T 中关键字为 key 的结点的双亲结点指针，其
//他情况返回 NULL
    BiTree temp1 = NULL, temp2 = NULL;
    if (T) //若当前节点 T 不为空
    {
        if (T->lchild && T->lchild->key == key)
            return T;      //若 T 有左子树且左子树的关键字为 key 则返回 T
        if (T->rchild && T->rchild->key == key)
            return T;      //若 T 有右子树且右子树的关键字为 key 则返回 T
        if (T->lchild)         //若 T 有左孩子
```

```
        temp1 = Parent(T->lchild, key);     //递归寻找双亲
    if (temp1)
        return temp1; //若在左子树找到双亲则返回
    if (T->rchild)        //否则在右子树查找并返回
        temp2 = Parent(T->rchild, key);
    return temp2;
  }
  return NULL;      //若根结点 T 为空直接返回 NULL
}
```

11.获得左孩子结点 LeftChild

初始条件：二叉树已存在

操作结果：获得二叉树 T 中关键字为 key 的结点的左孩子指针，未找到返回 NULL

操作步骤：先找到关键字为 key 的结点，若该结点不为空则返回其左孩子指针；否则返回 NULL。

具体代码：

```
BiTree LeftChild(BiTree T, KeyElem key)
{  //获得左孩子结点：返回二叉树 T 中关键字为 key 的结点的左孩子结点指针
  BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
  if (temp)
    return temp->lchild;     //temp 不为空返回左孩子指针
  return NULL;      //未找到返回 NULL
}
```

12.获得右孩子结点 RightChild

初始条件：二叉树已存在

操作结果：获得二叉树 T 中关键字为 key 的结点的右孩子指针，未找到返回 NULL

操作步骤：先找到关键字为 key 的结点，若该结点不为空则返回其右孩子指针；否则返回 NULL。

具体代码：

```
BiTree RightChild(BiTree T, KeyElem key)
{   //获得右孩子结点：返回二叉树 T 中关键字为 key 的结点的右孩子结点指针
    BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
    if (temp)
        return temp->rchild;     //temp 不为空返回右孩子指针
    return NULL;      //未找到返回 NULL
}
```

13.获得左兄弟结点 LeftSibling

初始条件：二叉树已存在

操作结果：获得二叉树 T 中关键字为 key 的结点的左兄弟指针，未找到返回 NULL

操作步骤：先找到关键字为 key 的双亲结点，若该结点不为空且左孩子不为 key 结点则返回其左孩子指针；其他情况返回 NULL。

具体代码：

```
BiTree LeftSibling(BiTree T, KeyElem key)
{   //获得左兄弟结点：返回二叉树 T 中关键字为 key 的结点的左兄弟结点指针
    BiTree dad = Parent(T, key);    //dad 为 key 结点的双亲结点
    if (dad && dad->lchild && dad->lchild->key != key)
//若 key 结点有双亲结点、双亲结点有左孩子且左孩子不为 key 结点
        return dad->lchild;      //返回双亲结点的左孩子
    return NULL;      //其他情况返回 NULL
}
```

14.获得右兄弟结点 RightSibling

初始条件：二叉树已存在

操作结果：获得二叉树 T 中关键字为 key 的结点的右兄弟指针，未找到返回 NULL

操作步骤：先找到关键字为 key 的双亲结点，若该结点不为空且右孩子不为 key 结点则返回其右孩子指针；其他情况返回 NULL。

具体代码：

```
BiTree RightSibling(BiTree T, KeyElem key)
{   //获得右兄弟结点：返回二叉树 T 中关键字为 key 的结点的右兄弟结点指针
    BiTree dad = Parent(T, key);    //dad 为 key 结点的双亲结点
    if (dad && dad->rchild && dad->rchild->key != key)
//若 key 结点有双亲结点、双亲结点有右孩子且右孩子不为 key 结点
        return dad->rchild;      //返回双亲结点的右孩子
    return NULL;      //其他情况返回 NULL
}
```

15.插入子树 InsertChild

初始条件：二叉树 T 已存在，右子树为空的插入子树 c 已存在

操作结果：在二叉树 T 中关键字为 key 的结点的 LR(0 为左/1 为右)子树插入子树 c，并将原来 key 结点的 LR 子树移至 c 的右子树中

操作步骤：先找到关键字为 key 的结点，若结点未找到返回 ERROR；判断 LR 的值来选择插入的子树进行插入，返回成功。

具体代码：

```
status InsertChild(BiTree T, KeyElem key, int LR, BiTree c)
{   //插入子树：在二叉树 T 中 key 结点的 LR（0 为左/1 为右）子树插入子树 c
    BiTree p = Vertex(T, key);      //p 指向 key 结点
    if (!p)        //若结点未找到则返回失败
        return ERROR;
    if (LR)        //插入到左子树
    {
        c->rchild = p->rchild;
```

```
        p->rchild = c;
    }
    else       //插入到右子树
    {
        c->rchild = p->lchild;
        p->lchild = c;
    }
    return OK;    //返回成功
}
```

16.删除子树 DeleteChild

初始条件：二叉树已存在

操作结果：删除二叉树 T 中关键字为 key 的结点的 LR 子树

操作步骤：先找到关键字为 key 的结点，若未找到返回 ERROR；判断 LR 的值且删除的子树存在则将子树删除，返回 OK。

具体代码：

```
status DeleteChild(BiTree T, KeyElem key, int LR)
{  //删除子树：删除二叉树 T 中关键字为 key 的结点的 LR 子树
    BiTree p = Vertex(T, key);      //p 为 key 结点
    if (!p)       //若结点未找到返回失败
        return ERROR;
    if (LR && p->rchild) //删除左子树
        ClearBiTree(&p->rchild);
    else if (p->lchild)      //删除右子树
        ClearBiTree(&p->lchild);
    return OK;    //返回成功
}
```

17.先序遍历 PreOrderTraverse

初始条件：二叉树已存在

操作结果：以根左右的顺序递归访问二叉树的数据

操作步骤：若当前节点不为空，则先访问该结点数据，再依次递归先序遍历左右子树。

具体代码：

```
void PreOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{   //先序遍历
    if (T)     //当前结点不为空
    {
        visit(T->key, T->data);     //先访问数据
        PreOrderTraverse(T->lchild, visit);
//再依次递归先序遍历左右子树
        PreOrderTraverse(T->rchild, visit);
    }
}
```

18.中序遍历 InOrderTraverse

初始条件：二叉树已存在

操作结果：以左根右的顺序递归访问二叉树的数据

操作步骤：若当前节点不为空，先递归中序遍历其左子树，在访问当前结点数据，最后递归中序遍历其右子树。

具体代码：

```
void InOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{   //中序遍历
    if (T)     //当前结点不为空
    {
        InOrderTraverse(T->lchild, visit);     //先递归中序遍历左子树
        visit(T->key, T->data);     //再访问当前节点数据
        InOrderTraverse(T->rchild, visit); //最后递归中序遍历右子树
```

```
    }
}
```

19.后序遍历 PostOrderTraverse

初始条件：二叉树已存在

操作结果：以左右根的顺序递归访问二叉树的数据

操作步骤：若当前节点不为空，先依次递归后序遍历其左右子树，最后在访问该结点的数据。

具体代码：

```
void PostOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{   //后序遍历
    if (T)     //当前结点不为空
    {
        PostOrderTraverse(T->lchild, visit);
//先依次递归后序遍历左右子树
        PostOrderTraverse(T->rchild, visit);
        visit(T->key, T->data);      //最后访问数据
    }
}
```

20.按层遍历 LevelOrderTraverse

初始条件：二叉树已存在

操作结果：按照由根结点到叶子结点的顺序从左到右依次访问二叉树数据

操作步骤：先判断二叉树根结点是否为空，若为空直接结束；若不为空，则先初始化队列 Q，并将当前结点初始化为根结点，循环访问当前结点并将其左右孩子入队，若队列中能出队一个元素则将其作为当前结点，并继续循环。

具体代码：

```
void LevelOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{   //层序遍历
```

```
    BiTree temp = T; //temp 指向二叉树根结点
    LinkQueue Q;
    if (!T)
        return;
    InitQueue(&Q);    //初始化队列 Q
    do {
        visit(temp->key, temp->data);  //先访问当前结点
        if (temp->lchild)    //若当前结点有左孩子则左孩子入队
            EnQueue(&Q, temp->lchild);
        if (temp->rchild)    //若当前节点有右孩子则右孩子入队
            EnQueue(&Q, temp->rchild);
    } while (DeQueue(&Q, &temp));
//若队列能出队一个元素则继续循环，出队元素赋值给 temp
    DestroyQueue(&Q);
}
```

21.将二叉树存入文件 SaveBiTreeElem SaveBiTNode

初始条件：二叉树已存在

操作结果：将二叉树的数据存入文件中

操作步骤：自行输入要保存的文件名，写方式打开文件，若打开失败返回 ERROR，然后从根结点开始递归将结点数据存入文件：若结点为空，则仅读入空字符标识，若不为空则依次读入关键字和数据，并递归将其左右孩子存入文件。

具体代码：

```
void SaveBiTNode(BiTree T, FILE* fp)
{  //将结点 T 中数据存入文件
    KeyElem nullkey = NULLNODE;     //nullkey 为空结点标识
    if (!T)        //若当前节点 T 为空则仅读入空节点标识
    {
        fwrite(&nullkey, sizeof(KeyElem), 1, fp);
```

```
        return;
    }
    fwrite(&T->key, sizeof(KeyElem), 1, fp);
//结点 T 不为空则依次读入关键字和数据
    fwrite(&T->data, sizeof(ElemType), 1, fp);
    SaveBiTNode(T->lchild, fp);     //递归将当前节点左右孩子存入文件
    SaveBiTNode(T->rchild, fp);
}

status SaveBiTreeElem(BiTList R)
{   //将二叉树 R 的数据存入文件
    FILE* fp;
    char filename[30];
    printf("input file name: ");        //输入文件名
    scanf("%s", filename);
    if ((fp = fopen(filename, "wb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    SaveBiTNode(R.Root, fp);//从根结点递归存入文件
    fclose(fp);
    return OK;
}
```

22.从文件加载二叉树 LoadBiTreeElem LoadBITNode

操作结果：创建一棵二叉树 R[num]并将文件 filename 中的数据存入二叉树中

操作步骤：先检测当前 R[num]二叉树是否存在，不存在则初始化二叉树，存

在则清空二叉树；然后以读方式打开文件 filename，打开失败返回 ERROR；然后从根节点递归加载数据：先读入一个关键字，若关键字为空结点标识则将当前结点置为空并返回，若当前结点不为空则继续将数据读入并创建二叉树结点赋值，再递归加载当前结点的左右子树。

具体代码：

```
void LoadBiTNode(BiTree* T, FILE* fp)
{   //从文件加载到二叉树结点 T
    KeyElem tempkey;
    ElemType tempdata;
    if (fread(&tempkey, sizeof(KeyElem), 1, fp)) //首先读取一个关键字
    {
        if (tempkey == NULLNODE)    //若读入的为空节点标识
        {
            *T = NULL;    //则当前结点为空并返回
            return;
        }
        fread(&tempdata, sizeof(ElemType), 1, fp);
//若当前节点不为空则继续将数据读入
        (*T) = (BiTree)malloc(sizeof(BiTNode));//创建一个二叉树结点
        (*T)->key = tempkey;    //结点赋值
        (*T)->data = tempdata;
        LoadBiTNode(&(*T)->lchild, fp);//递归加载当前结点的左右子树
        LoadBiTNode(&(*T)->rchild, fp);
    }
}

status LoadBiTreeElem(BiTList** R, int num, char* filename)
{   //从 filename 文件中读入数据到二叉树 R[num]
    FILE* fp;
```

```
    if (!R[num])  //若二叉树 R[num]不存在则初始化二叉树
        InitBiTree(&R[num]);
    else   //若已存在则清空树
        ClearBiTree(&R[num]->Root);
    if ((fp = fopen(filename, "rb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    LoadBiTNode(&R[num]->Root, fp);    //从根结点递归加载数据到结点
    fclose(fp);
    return OK;
}
```

23.其他函数

（1）获得结点指针 Vertex

初始条件：二叉树已存在

操作结果：返回二叉树中关键字为 key 的结点的指针

操作步骤：先判断当前结点是否为空，若为空则返回 NULL；在判断当前结点关键字是否为 key，是则直接返回当前结点指针；若不是则先递归左子树查找，若找到则返回找到的指针；未找到则在右子树查找并返回。

具体代码：

```
BiTree Vertex(BiTree T, KeyElem key)
{  //获得结点：返回二叉树 T 中指向关键字为 key 的结点指针
    BiTree temp1 = NULL, temp2 = NULL;
    if (T)     //若当前结点 T 不为空
    {
        if (T->key == key)   //若当前结点 T 的关键字即为所找
```

```
            return T;      //则直接返回当前结点指针
        if (T->lchild)        //若当前结点有左孩子结点
            temp1 = Vertex(T->lchild, key);//递归寻找结点并返回给 temp1
        if (temp1)     //若 temp1 不为空则已找到结点
            return temp1;      //返回 temp1 所指结点指针
        if (T->rchild)        //若左子树未找到，且有右子树
            temp2 = Vertex(T->rchild, key);//递归寻找结点并返回给 temp2
        return temp2;      //返回 temp2
    }
    return NULL;  //若根结点 T 为空直接返回 NULL
}
```

（2）访问二叉树结点数据 visit

操作结果：输出二叉树结点的关键字和数据

具体代码：

```
void visit(KeyElem key, ElemType e)
{   //结点数据访问
    printf("%c:%d\t", key, e);      //输出结点关键字和数据
}
```

（3）初始化队列 InitQueue

操作结果：建立链式队列的一个结点以及队首队尾指针

具体代码：

```
status InitQueue(LinkQueue* Q)
{   //初始化队列
    Q->front = Q->rear = (QPtr)malloc(sizeof(QNode));//创建首尾结点
    if (!Q->front)
        exit(OVERFLOW);
    Q->front->next = NULL;   //结点指针为空
```

```
    return OK;
}
```

（4）销毁队列 DestroyQueue

操作结果：销毁建立的队列结点

具体代码：

```
status DestroyQueue(LinkQueue* Q)
{   //销毁队列
    while (Q->front) //队首指针不为空则释放当前所指结点
    {
        Q->rear = Q->front->next;
        free(Q->front);
        Q->front = Q->rear;
    }
    return OK;
}
```

（5）元素入队 EnQueue

操作结果：将一个数据元素添加到队列末尾

具体代码：

```
status EnQueue(LinkQueue* Q, Qelem e)
{   //元素入队
    QPtr p;
    p = (QPtr)malloc(sizeof(QNode));   //创建队列结点
    if (!p)
        return FALSE;
    p->data = e;      //队列数据赋值
    p->next = NULL;
    Q->rear->next = p;   //队首队尾指针操作
```

```
    Q->rear = p;
    return OK;
}
```

（6）元素出队 DeQueue

操作结果：将队列的队首元素出队

具体代码：

```
status DeQueue(LinkQueue* Q, Qelem* e)
{   //元素出队
    QPtr p;
    if (Q->front == Q->rear) //队列为空返回失败
        return ERROR;
    p = Q->front->next;
    *e = p->data;     //队首元素赋值
    Q->front->next = p->next;
    if (Q->rear == p)     //若出队后队空则首尾指针指向一处
        Q->rear = Q->front;
    free(p);
    return OK;
}
```

2.4 实验小结

通过本次实验，个人编写了具有比较完善功能的二叉树演示系统，加深了个人对二叉树的链式存储结构的理解和认识，通过编写具体的程序，也让自己从代码层面理解了二叉树的初始化、创建、删除、获取结点以及遍历等功能如何实现。进一步提高了自己的编程能力。

相比于实验一中链表演示系统的编写，虽然演示系统的整体架构基本类似，但是实现二叉树功能的子函数明显比链表要多，甚至在层序遍历的过程中还需要

用到队列数据结构及其相关的函数，其函数中也多运用递归调用，因此二叉树演示系统的问题更复杂、实现难度更大。

在编程过程中也深刻体会到，随着程序实现的功能越多、情况也越复杂，需要尽可能的考虑到所有可能出现的情况，比如空树、插入空树、输入非法数字等特殊情况，程序都需要有相关的解决方案。因而在整体程序写完后，个人也是在代码审核、写注释、写报告的过程中不断发现一些特殊情况，并不断完善程序才能使其尽可能对所有情况做出应答而非程序异常中断。在这个过程中也体会到编写程序的不易之处，以及最后代码调式阶段的重要性。

参考文献

[1] 严蔚敏等. 数据结构(C 语言版). 清华大学出版社

[2] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition, Calvin College, 2005

[3] 殷立峰. Qt C++跨平台图形界面程序设计基础. 清华大学出版社,2014:192～197

[4] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

指导教师评定意见

一、对实验报告的评语

二、对实验报告评分

评分项目（分值）	程序内容（36.8分）	程序规范（9.2分）	报告内容（36.8分）	报告规范（9.2分）	考勤（8分）	逾期扣分	合　计（100分）
得分							

附录 A 基于链式存储结构线性表实现的源程序

```
 /* 线性链表 */


#define _CRT_SECURE_NO_WARNINGS   //VS 编译器的原因需要添加的宏定义

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include<string.h>

//基本标识宏定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
#define LISTMAXNUM 9     //允许建立的最大链表个数


typedef int status;
typedef int ElemType; //数据元素类型定义

typedef struct LNode{
    ElemType data;
    struct LNode* next;
}*Link; //链表节点结构体

typedef struct {
    Link head;
    int len;
}LinkList;              //链表整体结构体

int num = 0;       //建立的链表个数
char listname[LISTMAXNUM][20];   //建立的链表的名称


//子函数声明
status InitList(LinkList** L);        //初始化链表
status CreateList(LinkList* L);          //创建链表结点并输入
status ListExist(LinkList* L);       //判定链表是否存在
status DestoryList(LinkList** L, int oplist);          //销毁链表
```

```
status ClearList(LinkList* L);       //清空链表
status ListEmpty(LinkList L);        //判定链表是否为空表
int ListLength(LinkList L);          //求链表长度
status GetELem(LinkList L, int i, ElemType* e);          //获得元素
status compare(struct LNode node, ElemType e);          //比较函数
int LocateElem(LinkList L, ElemType e, status compare(struct LNode, ElemType));
    //根据条件查找元素
status PriorElem(LinkList L, ElemType cur_e, ElemType* pre_e);          //获得前驱
status NextElem(LinkList L, ElemType cur_e, ElemType* next_e);          //获得后继
status ListInsert(LinkList* L, int i, ElemType e);          //插入元素
status ListDelete(LinkList* L, int i, ElemType* e);         //删除元素
status ListTraverse(LinkList L, void (*visit)(ElemType));          //遍历链表
void visit(ElemType);           //访问结点中元素
status SaveElem(LinkList L);        //链表元素存入文件
status LoadElem(LinkList** L, int num, char* filename);          //从文件读入链表元素


//主函数：主界面和调用子函数预处理
void main(void) {
    LinkList* L[LISTMAXNUM] = { NULL };          //链表指针数组
    char tempname[20];        //临时名称
    int op = 1;
    int temp;
    int oplist;          //选择的链表序号
    int count=num;        //临时建立链表的次数（计数器）
    ElemType elemdest,elemsrc;        //源元素，目的元素

    while (op)
    {
        system("cls");    printf("\n\n");
        printf("          Menu for Linear Table On Chain Structure \n");
        printf("-------------------------------------------------\n");
        printf("           1. InitList              8. PriorElem\n");
        printf("           2. DestroyList          9. NextElem \n");
        printf("           3. ClearList            10.ListInsert \n");
        printf("           4. ListEmpty            11.ListDelete\n");
        printf("           5. ListLength           12.ListTrabverse\n");
        printf("           6. GetElem              13.SaveElem \n");
        printf("                7. LocateElem          14.LoadElem\n");
        printf("           0. Exit\n");
        printf("-------------------------------------------------\n");
        printf("Active List:\t");          //显示当前已建立的链表
        if (num)
```

```
            for (int i = 0; i < num; i++)
            {
                if ((i + 1) % 4 == 0)
                    putchar('\n');
                printf("%d: %s\t", i, listname[i]);
            }
        printf("\n-------------------------------------------------\n");
        printf("       请选择你的操作[0~14]:");
        scanf("%d", &op);

        if (op == 1)
        {   //输入 1 则初始化链表同时创建链表输入数据
            if                  (InitList(&L[num%LISTMAXNUM])                  &&
CreateList(L[num%LISTMAXNUM]))
            {   //若建立链表成功对链表临时命名
                memset(tempname, 0, 20);
                strcat(tempname, "temp");
                _itoa(count, tempname + 4, 10);
                strcpy(listname[num%LISTMAXNUM], tempname);
                num = (num >= LISTMAXNUM ? LISTMAXNUM : num + 1);
                //若已建立链表未超过最大数目：建立链表数+1，否则保持最大
数目不变（超过最大数目的链表一直覆盖第一个链表）
                count++;     //计数器+1
                printf("\n 线性表创建成功！\n");
            }
            else
                printf("\n 线性表创建失败！\n");
            system("pause");
        }
        else if (op == 14)
        {   //输入 14 加载文件中的数据
            printf("input file name: ");
            scanf("%s", tempname);
            if (LoadElem(L,num%9,tempname))
            {   //加载数据成功则对链表命名为文件名
                strcpy(listname[num % LISTMAXNUM], tempname);
                num = (num >= LISTMAXNUM ? LISTMAXNUM : num+1);
                printf("\n 载入数据成功!\n");
            }
            else
                printf("\nFalse!\n");
            system("pause");
        }
        else if(op>1&&op<14)
```

```
{   //输入其他选项
    if (num > 1)
    {   //若已建立链表数＞1 则操作前需要确定操作链表的序号
        printf("请输入操作链表序号：");
        scanf("%d", &oplist);
    }
    else //若仅有一个链表，则该链表默认为操作的链表
        oplist = 0;

    if (oplist<0||oplist>=num||!ListExist(L[oplist]))
    {   //检查输入操作链表的序号是否合法 以及操作的链表是否存在
        printf("\nList is NOT EXIST!\n");
        system("pause");
    }
    else //链表序号合法且链表存在
        switch (op)
        {
        case 2:         //输入 2  销毁链表
        if (DestoryList(L,oplist))
            printf("\n 线性表已删除！ \n");
        else
            printf("\nFalse!\n");
        system("pause");
        break;

        case 3:         //输入 3  清空链表
            if (ClearList(L[oplist]))
                printf("\n 线性表已清空！ \n");
            else
                printf("\nFalse!\n");
            system("pause");
            break;

        case 4:         //判断链表是否为空
            if (ListEmpty(*L[oplist]))
                printf("\nList is empty!\n");
            else
                printf("\nList is NOT empty!\n");
            system("pause");
            break;

        case 5:         //输出链表长度（元素个数）
            printf("\nThe length of the list is:%d\n", ListLength(*L[oplist]));
            system("pause");
```

```
            break;

        case 6:     //获取链表某元素
            printf("\n 请输入获取的元素序号：");
            scanf("%d", &temp);
            if (GetELem(*L[oplist], temp, &elemdest))
                printf("\n 获取成功：%d\n", elemdest);
            else
                printf("\n 获取失败！\n");
            system("pause");
            break;

        case 7:     //根据条件比对元素输出序号
            printf("\n 请输入待比对数据：");
            scanf("%d", &elemsrc);
            printf("\n 第一个满足的数据序号： %d\n",
LocateElem(*L[oplist], elemsrc, compare));
            system("pause");
            break;

        case 8:     //输出某元素的前驱
            printf("\n 请输入一个数据元素：");
            scanf("%d", &elemsrc);
            if (PriorElem(*L[oplist], elemsrc, &elemdest))
                printf("\n 获取成功：%d\n", elemdest);
            else
                printf("\n 获取失败！\n");
            system("pause");
            break;

        case 9:     //输出某元素的后件
            printf("\n 请输入一个数据元素：");
            scanf("%d", &elemsrc);
            if (NextElem(*L[oplist], elemsrc, &elemdest))
                printf("\n 获取成功：%d\n", elemdest);
            else
                printf("\n 获取失败！\n");
            system("pause");
            break;

        case 10:    //插入元素
            printf("\n 请输入插入元素序号：");
            scanf("%d", &temp);
            printf("\n 请输入插入元素数值：");
```

```
                scanf("%d", &elemsrc);
                if (ListInsert(L[oplist], temp, elemsrc))
                    printf("\n 插入成功!\n");
                else
                    printf("\nFalse!\n");
                system("pause");
                break;

            case 11:      //删除元素
                printf("\n 请输入删除元素序号：");
                scanf("%d", &temp);
                if (ListDelete(L[oplist], temp, &elemdest))
                    printf("\n 删除成功: Delete %d\n", elemdest);
                else
                    printf("\nFalse!\n");
                system("pause");
                break;

            case 12:      //遍历链表
                if (!ListTraverse(*L[oplist], visit))
                    printf("线性表是空表！\n");
                system("pause");
                break;

            case 13:      //将链表中的元素存入到文件中
                if(SaveElem(*L[oplist]))
                    printf("\n 保存数据成功!\n");
                else
                    printf("\nFalse!\n");
                system("pause");
                break;
            }
        }
        else if (op)   //输入其他数字
        {
            printf("\n 输入错误!\n");
            system("pause");
        }
    }//end of while

    for (int i = 0; i < num; i++)          //退出系统前销毁所以已建链表
        DestoryList(L, i);
    printf("欢迎下次再使用本系统！\n");
}//end of main()
```

```
status InitList(LinkList** L)
{    //初始化链表：建立链表结构体（*L）并创建头结点
     (*L) = (LinkList*)malloc(sizeof(LinkList));
     (*L)->head = (Link)malloc(sizeof(struct LNode));  //带创建头节点
     (*L)->head->next = NULL;
     if (!(*L)->head)
          return OVERFLOW;
     (*L)->len = 0;
     return OK;
}

status CreateList(LinkList* L)
{    //创建链表：创建链表 L 的节点并输入数据（输入 0 时结束创建）
     Link end, New;
     int e;
     end = L->head;
     printf("Input Element:");
     scanf("%d", &e);
     if (!e)          //开始输入 0 则不创建结点子函数结束
          return ERROR;
     while (e)
     {
          New = (Link)malloc(sizeof(struct LNode));//创建结点
          if (!New)
               return OVERFLOW;
          New->data = e;         //数据赋值
          New->next = end->next;
          end->next = New;
          end = New;
          L->len++;   //链表长度+1
          scanf("%d", &e);
     }
     return OK;
}

status ListExist(LinkList* L)
{    //检测链表 L 是否存在，FALSE 不存在，TRUE 存在
     if (!L)
          return FALSE;
     return TRUE;
```

```
}

status DestoryList(LinkList** L, int oplist)
{    //销毁链表 L[oplist]
     Link cur, next;
     cur = L[oplist]->head;
     while (cur)         //销毁链表结点
     {
          next = cur->next;
          free(cur);
          cur = next;
     }
     free(L[oplist]);    //销毁链表结构体
     for (; oplist < num - 1; oplist++)
     {
          L[oplist] = L[oplist + 1];         //将销毁链表后的链表前移
          strcpy(listname[oplist], listname[oplist + 1]);          //销毁链表后的链表名
前移
     }
     memset(listname[oplist], 0, 20);    //最后的链表名清空
     L[oplist] = NULL;
     num--;         //已建立链表数-1
     return OK;
}

status ClearList(LinkList* L)
{    //清空链表 L：使链表的元素结点释放，保留头结点
     Link cur, next;
     cur = L->head->next;
     while (cur)         //释放元素结点
     {
          next = cur->next;
          free(cur);
          cur = next;
     }
     L->head->next = NULL;
     L->len = 0;         //当前链表长度置 0
     return OK;
}

status ListEmpty(LinkList L)
{    //检测链表 L 是否为空表：TRUE 为空表，FLASE 表不为空
     if (L.head->next)
          return FALSE;
```

```
    return TRUE;
}

int ListLength(LinkList L)
{   //返回链表 L 的长度
    //Link temp;
    //int len;
    //for (len = 0, temp = L.head->next; temp; temp = temp->next)
    //    len++;
    //return len;
    return L.len;
}

status GetELem(LinkList L, int i, ElemType* e)
{   //获取链表 L 第 i 个元素并存入 e 中
    Link temp;
    int j;
    if (i <= 0 || i > L.len)  //i 的合法性检测
        return ERROR;
    for (j = 1, temp = L.head->next; j < i; temp = temp->next, j++);//使 temp 指向待访问元素的结点
    *e = temp->data;//元素赋值给 e
    return OK;
}

status compare(struct LNode node, ElemType e)
{   //比较函数：若结点中的元素大于 e 则返回 TRUE，否则返回 FALSE（可自定义）
    if (node.data > e)
        return TRUE;
    else
        return FALSE;
}

int LocateElem(LinkList L, ElemType e, status compare(struct LNode, ElemType))
{   //根据比较函数 compare 返回满足条件的第一个元素的序号
    Link temp;
    int i;
    for (i = 1, temp = L.head->next; temp; temp = temp->next, i++)
        if (compare(*temp, e))     //遍历链表并对比较
            return i;
    return 0;
}
```

```
status PriorElem(LinkList L, ElemType cur_e, ElemType* pre_e)
{    //返回链表 L 中元素为 cur_e 的前驱元素到 pre_e 中
     Link pre = L.head, cur = pre->next;
     if (!cur || cur_e == cur->data)        //检测链表是否为空表，同时 cur_e 不为第一个元素
          return ERROR;
     for (; cur; pre = cur, cur = cur->next)
          if (cur_e == cur->data)           //判断是否为 cur_e
          {
               *pre_e = pre->data;
               return OK;
          }
     return ERROR;  //未找到返回 ERROR
}

status NextElem(LinkList L, ElemType cur_e, ElemType* next_e)
{    //返回链表 L 中元素为 cur_e 的后继元素到 next_e 中
     Link cur = L.head->next, next;
     if (!cur) //检测链表是否为空表
          return ERROR;
     next = cur->next;
     for (; next; cur = next, next = next->next)   //遍历链表至倒数第二个元素
          if (cur_e == cur->data)           //判断是否为 cur_e
          {
               *next_e = next->data;
               return OK;
          }
     return ERROR;  //未找到返回 ERROR
}

status ListInsert(LinkList* L, int i, ElemType e)
{    //在链表 L 的第 i（1<=i<=len+1）个元素前插入元素 e
     Link temp, New;
     int j;
     if (i<1 || i>L->len + 1)      //检测 i 的合法性，不合法返回 ERROR
          return ERROR;
     for (j = 1, temp = L->head; j < i; j++, temp = temp->next);   //遍历到待插入位置
     New = (Link)malloc(sizeof(struct LNode));      //新建结点
     if (!New)
          return ERROR;
     New->data = e;         //结点赋值
     New->next = temp->next;      //结点链接到链表
     temp->next = New;
```

```
    L->len++;    //链表长度+1
    return OK;
}

status ListDelete(LinkList* L, int i, ElemType* e)
{    //删除链表 L 的第 i 个元素并将其值赋给 e
    Link temp, dest;
    int j;
    if (i<1 || i>L->len)    //检测 i 的合法性，不合法返回 ERROR
        return ERROR;
    for (j = 1, temp = L->head; j < i; j++, temp = temp->next);    //遍历到待删除位
置
    dest = temp->next;
    *e = dest->data;        //将删除值赋给 e
    temp->next = dest->next;
    free(dest);          //释放结点
    L->len--;            //链表长度-1
    return OK;
}


status ListTraverse(LinkList L, void (*visit)(ElemType))
{    //使用 vist 函数遍历链表 L
    Link temp;
    if (!L.head->next)
        return ERROR;
    for (temp = L.head->next; temp; temp = temp->next)
        visit(temp->data);
    putchar('\n');
    return OK;
}

void visit(ElemType e)
{    //访问（输出）结点 node 的元素
    printf("%d\t", e);
}

status SaveElem(LinkList L)
{    //将链表 L 中的元素存入文件中
    Link temp;
    FILE* fp;
    char filename[30];
    printf("input file name: ");          //输入文件名
    scanf("%s", filename);
```

```
    if ((fp = fopen(filename, "wb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    for (temp = L.head->next; temp; temp = temp->next)
        fwrite(&temp->data, sizeof(ElemType), 1, fp);  //将每个结点中的元素写入
文件中
    fclose(fp);
    return OK;
}

status LoadElem(LinkList** L, int num, char* filename)
{   //加载文件名为 filename 中的元素到链表 L[num]中
    FILE* fp;
    Link end, New;
    ElemType temp;
    if (!ListExist(L[num]))          //若链表 L[num]不存在则初始化
        InitList(&L[num]);
    else
        ClearList(L[num]);           //若链表 L[num]已存在则清空链表
    if ((fp = fopen(filename, "rb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    end = L[num]->head;
    while (fread(&temp, sizeof(ElemType), 1, fp))
    {   //每次读入一个数据并存入到建立的链表结点中
        New = (Link)malloc(sizeof(struct LNode));
        New->data = temp;
        New->next = end->next;
        end->next = New;
        end = New;
        L[num]->len++;
    }
    return OK;
}
```

附录 B 基于二叉链表二叉树实现的源程序

```
/*  二叉链表  */


#define _CRT_SECURE_NO_WARNINGS    //VS 编译器的原因需要添加的宏定义
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include<string.h>

//基本标识宏定义
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
#define TREEMAXNUM 9    //允许建立的最大二叉树个数
#define NULLNODE ' '    //空节点的关键字标识


typedef int status;
typedef int ElemType; //数据元素类型定义
typedef char KeyElem;    //关键字元素类型定义

typedef struct BiNode {
    KeyElem key;    //关键字
    ElemType data;  //数据
    struct BiNode* lchild, *rchild;      //左右孩子指针
}BiTNode,*BiTree;   //二叉树结点结构体

typedef struct BiTList {
    BiTree Root;//二叉树根结点
}BiTList;    //二叉树整体结构体

typedef BiTree Qelem;    //重定义二叉树结点类型为队列数据类型

typedef struct QNode {
    Qelem data;       //数据
    struct QNode* next;        //指针域
}QNode,*QPtr;  //队列结点结构体

typedef struct {
```

```
    QPtr front, rear;       //队列队首、队尾指针
}LinkQueue;             //队列整体结构体

int num = 0; //建立的二叉树个数
char treename[TREEMAXNUM][20]; //建立的二叉树的名称


//子函数声明
status InitQueue(LinkQueue* Q);       //初始化队列
status DestroyQueue(LinkQueue* Q);       //销毁队列
status EnQueue(LinkQueue* Q, Qelem e);       //元素入队
status DeQueue(LinkQueue* Q, Qelem* e);          //元素出队
status InitBiTree(BiTList** R);          //初始化二叉树
BiTree CreateBiTree(int count);          //创建二叉树
void ClearBiTree(BiTree* T);       //清空二叉树
void DestroyBiTree(BiTList** R,int oplist);          //销毁二叉树
status BiTreeEmpty(BiTList R);          //判定空二叉树
int BiTreeDepth(BiTree T);          //求二叉树深度
BiTree Root(BiTList R);       //获得根结点
void visit(KeyElem key, ElemType e);       //访问二叉树结点数据
BiTree Vertex(BiTree T, KeyElem key);          //获得结点指针
ElemType Value(BiTree T, KeyElem key);       //获得结点值
status Assign(BiTree T, KeyElem key, ElemType value);      //结点赋值
BiTree Parent(BiTree T, KeyElem key);          //获得双亲结点
BiTree LeftChild(BiTree T, KeyElem key);       //获得左孩子结点
BiTree RightChild(BiTree T, KeyElem key);          //获得右孩子结点
BiTree LeftSibling(BiTree T, KeyElem key);          //获得左兄弟结点
BiTree RightSibling(BiTree T, KeyElem key);        //获得右兄弟结点
status InsertChild(BiTree T, KeyElem key, int LR, BiTree c);      //插入子树
status DeleteChild(BiTree T, KeyElem key, int LR);       //删除子树
void PreOrderTraverse(BiTree T, void visit(KeyElem, ElemType));          //前序遍历
void InOrderTraverse(BiTree T, void visit(KeyElem, ElemType));           //中序遍历
void PostOrderTraverse(BiTree T, void visit(KeyElem, ElemType));         //后序遍历
void LevelOrderTraverse(BiTree T, void visit(KeyElem, ElemType));       //层序遍历
void SaveBiTNode(BiTree T, FILE* fp);          //将二叉树结点元素存入文件
status SaveBiTreeElem(BiTList R);          //将二叉树存入文件
void LoadBiTNode(BiTree* T, FILE* fp);       //从文件加载二叉树结点元素
status LoadBiTreeElem(BiTList** R, int num, char* filename);        //从文件加载二
叉树


//主函数
void main(void) {
    BiTList* T[TREEMAXNUM] = { NULL };          //二叉树指针数组
```

```
BiTree temptree=NULL;  //临时二叉树结点指针
char tempname[20];         //临时名称
int op = 1;
int temp;
int oplist;            //选择的二叉树序号
int count = num;       //临时建立二叉树的次数（计数器）
KeyElem tempkey;
ElemType elemdest, elemsrc;        //源元素，目的元素

while (op)
{
    system("cls");    printf("\n\n");
    printf("           Menu for Binary Tree On Chain Structure \n");
    printf("-------------------------------------------------\n");
    printf("            1. InitBiTree              12.RightChild\n");
    printf("            2. DestroyBiTree           13.LeftSibling\n");
    printf("            3. CreatBiTree             14.RightSibling\n");
    printf("            4. ClearBiTree             15.InsertChild\n");
    printf("            5. BiTreeEmpty             16.DeleteChild\n");
    printf("            6. BiTreeDepth             17.PreOrderTraverse \n");
    printf("                  7. Root                    18.InOrderTraverse\n");
    printf("            8. Value                   19.PostOrderTraverse\n");
    printf("            9. Assign                  20.LevelOrderTraverse\n");
    printf("            10.Parent                  21.SaveBiTreeElem\n");
    printf("            11.LeftChild               22.LoadBiTreeElem\n");
    printf("            0. Exit\n");
    printf("-------------------------------------------------\n");
    printf("Active Tree:\t");          //显示当前已建立的二叉树
    if (num)
        for (int i = 0; i < num; i++)
        {
            if ((i + 1) % 4 == 0)
                putchar('\n');
            printf("%d: %s\t", i, treename[i]);
        }
    printf("\n-------------------------------------------------\n");
    printf("        请选择你的操作[0~22]:");
    scanf("%d", &op);

    if (op == 1)
    {    //输入 1 则初始化二叉树
        if (InitBiTree(&T[num % TREEMAXNUM]))
        {    //若建立二叉树成功对链表临时命名
            memset(tempname, 0, 20);
```

```
                strcat(tempname, "temp");
                _itoa(count, tempname + 4, 10);
                strcpy(treename[num % TREEMAXNUM], tempname);
                num = (num >= TREEMAXNUM ? TREEMAXNUM : num + 1);
                //若已建立二叉树未超过最大数目：建立二叉树个数+1，否则保持最大数目不变（超过最大数目的二叉树一直覆盖第一个二叉树）
                count++;     //计数器+1
                printf("\n 二叉树初始化成功！\n");
            }
            else
                printf("\n 二叉树初始化失败！\n");
            system("pause");
        }

        else if (op == 22)
        {   //输入 22 加载文件中的数据
            printf("input file name: ");
            scanf("%s", tempname);
            if (LoadBiTreeElem(T,num,tempname))
            {   //加载数据成功则对二叉树命名为文件名
                strcpy(treename[num % TREEMAXNUM], tempname);
                num = (num >= TREEMAXNUM ? TREEMAXNUM : num + 1);
                printf("\n 二叉树数据加载成功!\n");
            }
            else
                printf("\nFalse!\n");
            system("pause");
        }
        else if (op > 1 && op < 22)
        {   //输入其他选项
            if (num > 1)
            {   //若已建立链表数＞1 则操作前需要确定操作链表的序号
                printf("请输入操作链表序号：");
                scanf("%d", &oplist);
            }
            else //若仅有一个链表，则该链表默认为操作的链表
                oplist = 0;

            if (oplist < 0 || oplist >= num || !T[oplist])
            {   //检查输入操作二叉树的序号是否合法 以及操作的二叉树是否存在
                printf("\nBiTree is NOT EXIST!\n");
                system("pause");
            }
```

```
else //二叉树序号合法且二叉树存在
    switch (op)
    {
    case 2:        //输入 2  销毁二叉树
        DestroyBiTree(T,oplist);
        printf("\n 二叉树已销毁!\n");
        system("pause");
        break;

    case 3:        //输入 3  创建二叉树
        T[oplist]->Root = CreateBiTree(1);
        printf("二叉树已建立成功!\n");
        system("pause");
        break;
    case 4:        //输入 4  清空二叉树
        if (T[oplist] && T[oplist]->Root)
        {
            ClearBiTree(&T[oplist]->Root);
            printf("\n 二叉树已清空!\n");
        }
        else
            printf("\n 二叉树清空失败！ \n");
        system("pause");
        break;

    case 5:        //判断二叉树是否为空
        if (BiTreeEmpty(*T[oplist]))
            printf("\nBiTree is empty!\n");
        else
            printf("\nBiTree is NOT empty!\n");
        system("pause");
        break;

    case 6:        //输出二叉树深度
       printf("\n 二叉树深度为:%d\n", BiTreeDepth(T[oplist]->Root));
        system("pause");
        break;

    case 7:        //求二叉树根结点并输出
        if (temptree = Root(*T[oplist]))
        {
            printf("\n 根结点： ");
            visit(temptree->key, temptree->data);
        }
```

```
        else
            printf("\n 根结点为空！\n");
        putchar('\n');
        system("pause");
        break;

    case 8:         //获取结点数据值
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        elemdest = Value(T[oplist]->Root, tempkey);
        if ( elemdest== -1)
            printf("\n 该结点未找到！\n");
        else
            printf("\n 该结点已找到，数据值：%d\n", elemdest);
        system("pause");
        break;

    case 9:         //结点赋值
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        printf("\n 请输入新的数据：");
        scanf("%d", &elemsrc);
        if (Assign(T[oplist]->Root, tempkey, elemsrc))
            printf("\n 结点已重新赋值！\n");
        else
            printf("\n 结点赋值失败！\n");
        system("pause");
        break;

    case 10:        //获得双亲结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        if (temptree = Parent(T[oplist]->Root, tempkey))
        {
            printf("\n 其双亲结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点双亲获得失败！\n");
        system("pause");
        break;

    case 11:        //获得左孩子结点并输出
```

```
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        if (temptree = LeftChild(T[oplist]->Root, tempkey))
        {
            printf("\n 其左孩子结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点左孩子获得失败！\n");
        system("pause");
        break;

    case 12:    //获得右孩子结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        if (temptree = RightChild(T[oplist]->Root, tempkey))
        {
            printf("\n 其右孩子结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点右孩子获得失败！\n");
        system("pause");
        break;

    case 13:    //获得左兄弟结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        if (temptree = LeftSibling(T[oplist]->Root, tempkey))
        {
            printf("\n 其左兄弟结点为 ");
            visit(temptree->key, temptree->data);
            putchar('\n');
        }
        else
            printf("\n 结点左兄弟获得失败！\n");
        system("pause");
        break;

    case 14:    //获得右兄弟结点并输出
        printf("请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
```

```
                if (temptree = RightSibling(T[oplist]->Root, tempkey))
                {
                    printf("\n 其右兄弟结点为 ");
                    visit(temptree->key, temptree->data);
                    putchar('\n');
                }
                else
                    printf("\n 结点右兄弟获得失败！\n");
                system("pause");
                break;

            case 15:      //插入子树
                printf("建立插入子树 c：\n");
                printf("需输入 1 个结点\n 请输入关键字:");
                scanf("%*c%c", &tempkey);  //输入子树根结点
                if (tempkey == NULLNODE) //若子树根结点为空则不合法
                {
                    printf("\n 建立子树不合法！\n");
                    system("pause");
                    break;
                }

                temptree = (BiTree)malloc(sizeof(BiTNode));        //若子树根
结点不为空继续操作
                if (!temptree)
                {
                    printf("建立子树失败！\n");
                    system("pause");
                    break;
                }
                temptree->key = tempkey;
                printf("请输入数据:");
                scanf("%d", &temptree->data);
                //temptree->father = NULL;//
                temptree->lchild = CreateBiTree(1);    //仅递归创建子树的左
子树，右子树为空
                temptree->rchild = NULL;

                printf("\n 请输入搜索关键字：");
                scanf("%*c%c", &tempkey);
                printf("\n 请输入操作子树(0.左子树/1.右子树):");
                scanf("%d", &temp);
                if (temp != 0 && temp != 1)
                {
```

```
            printf("\n 输入错误!\n");
            system("pause");
            break;
        }
        if (InsertChild(T[oplist]->Root, tempkey, temp, temptree))
            printf("\n 插入子树成功！\n");
        else
            printf("\n 插入子树失败！\n");
        system("pause");
        break;

    case 16:       //删除子树
        printf("\n 请输入搜索关键字：");
        scanf("%*c%c", &tempkey);
        printf("\n 请输入操作子树(0.左子树/1.右子树):");
        scanf("%d", &temp);
        if (temp != 0 && temp != 1)
        {
            printf("\n 输入错误!\n");
            system("pause");
            break;
        }
        if (DeleteChild(T[oplist]->Root, tempkey, temp))
            printf("\n 删除子树成功！\n");
        else
            printf("\n 删除子树失败！\n");
        system("pause");
        break;

    case 17:
        printf("前序遍历：");
        PreOrderTraverse(T[oplist]->Root,visit);
        putchar('\n');
        system("pause");
        break;

    case 18:
        printf("中序遍历：");
        InOrderTraverse(T[oplist]->Root, visit);
        putchar('\n');
        system("pause");
        break;

    case 19:
```

```
                printf("后序遍历：");
                PostOrderTraverse(T[oplist]->Root, visit);
                putchar('\n');
                system("pause");
                break;

            case 20:
                printf("层序遍历：");
                LevelOrderTraverse(T[oplist]->Root, visit);
                putchar('\n');
                system("pause");
                break;

            case 21:      //将二叉树中的元素存入到文件中
                if (SaveBiTreeElem(*T[oplist]))
                    printf("\n 二叉树数据已保存!\n");
                else
                    printf("\n 二叉树数据保存失败！\n");
                system("pause");
                break;
            }
        }
        else if (op)   //输入其他数字
        {
            printf("\n 输入错误!\n");
            system("pause");
        }
    }//end of while

    for (int i = 0; i < num; i++)          //退出系统前销毁所以已建二叉树
        DestroyBiTree(T,i);
    printf("欢迎下次再使用本系统！\n");
}//end of main()


status InitQueue(LinkQueue* Q)
{   //初始化队列
    Q->front = Q->rear = (QPtr)malloc(sizeof(QNode));     //创建首尾结点
    if (!Q->front)
        exit(OVERFLOW);
    Q->front->next = NULL; //结点指针为空
    return OK;
}
```

```
status DestroyQueue(LinkQueue* Q)
{    //销毁队列
     while (Q->front) //队首指针不为空则释放当前所指结点
     {
          Q->rear = Q->front->next;
          free(Q->front);
          Q->front = Q->rear;
     }
     return OK;
}

status EnQueue(LinkQueue* Q, Qelem e)
{    //元素入队
     QPtr p;
     p = (QPtr)malloc(sizeof(QNode)); //创建队列结点
     if (!p)
          return FALSE;
     p->data = e;        //队列数据赋值
     p->next = NULL;
     Q->rear->next = p;    //队首队尾指针操作
     Q->rear = p;
     return OK;
}

status DeQueue(LinkQueue* Q, Qelem* e)
{    //元素出队
     QPtr p;
     if (Q->front == Q->rear)  //队列为空返回失败
          return ERROR;
     p = Q->front->next;
     *e = p->data;             //队首元素赋值
     Q->front->next = p->next;
     if (Q->rear == p)       //若出队后队空则首尾指针指向一处
          Q->rear = Q->front;
     free(p);
     return OK;
}

status InitBiTree(BiTList** R)
{    //初始化二叉树 R
     (*R) = (BiTList*)malloc(sizeof(BiTList));  //创建二叉树结构体
     if (!(*R)->Root)
          return OVERFLOW;
     (*R)->Root = NULL;       //二叉树根结点设为空指针
```

```
    return OK;
}

BiTree CreateBiTree(int count)
{   //创建二叉树：count 为当前还需输入的结点个数
    KeyElem e;
    BiTree New = NULL;
    printf("需输入%d 个结点\n 请输入关键字:", count);
    scanf("%*c%c", &e);
    count--; //需输入结点个数-1
    if (e != NULLNODE)          //若输入的关键字不为空节点标识
    {
        New = (BiTree)malloc(sizeof(BiTNode));      //创建二叉树结点
        if (!New)
            return FALSE;
        New->key = e;       //结点赋值
        printf("请输入数据:");
        scanf("%d", &New->data);
        //New->father = father;    //
        New->lchild = CreateBiTree(count + 2);      //递归建立子节点
        New->rchild = CreateBiTree(count + 1);
    }
    return New;     //返回二叉树根结点
}


void ClearBiTree(BiTree* T)
{   //清空二叉树 T
    if ((*T)->lchild)  //若二叉树 T 的子节点不为空则先递归清空子树
        ClearBiTree(&(*T)->lchild);
    if ((*T)->rchild)
        ClearBiTree(&(*T)->rchild);
    if (!(*T)->lchild && !(*T)->rchild)         //若 T 是叶子结点
    {
        free(*T);     //则释放结点内存
        *T = NULL; //指针指向空
    }
}

void DestroyBiTree(BiTList** R,int oplist)
{   //销毁二叉树 R[oplist]
    if(R[oplist]->Root)
        ClearBiTree(&R[oplist]->Root);   //先清空二叉树所有结点
    free(R[oplist]);   //释放二叉树结构体
```

```
    for (; oplist < num - 1; oplist++)
    {
        R[oplist] = R[oplist + 1];        //将销毁二叉树后的二叉树前移
        strcpy(treename[oplist], treename[oplist + 1]);        //销毁二叉树后的二叉树名称前移
    }
    memset(treename[oplist], 0, 20);   //最后二叉树名称清空
    R[oplist] = NULL;          //二叉树指针指向空
    num--;         //已建立二叉树个数-1
}


status BiTreeEmpty(BiTList R)
{   //判定 R 是否为空二叉树：TRUE 树空，FALSE 树不为空
    if (R.Root)         //若二叉树根结点不为空
        return FALSE;          //则树不为空
    else       //若二叉树根结点为空
        return TRUE;     //则二叉树为空
}

int BiTreeDepth(BiTree T)
{   //求二叉树 T 的深度：返回二叉树深度的整数
    int dep = 0;  //初始化树的深度为 0
    int i, j;
    if (T)     //若 T 的根结点结点不为空
    {
        dep++;         //树的深度+1
        i = BiTreeDepth(T->lchild);          //递归求结点的左右子树深度
        j = BiTreeDepth(T->rchild);
        dep += i > j ? i : j;     //树 T 的深度为原深度+子树深度大的深度
    }
    return dep;         //返回树的深度
}


BiTree Root(BiTList R)
{   //返回二叉树 R 的根结点
    return R.Root;
}

void visit(KeyElem key, ElemType e)
{   //结点数据访问
    printf("%c:%d\t", key, e);       //输出结点关键字和数据
}
```

```
BiTree Vertex(BiTree T, KeyElem key)
{   //获得结点：返回二叉树 T 中指向关键字为 key 的结点指针
    BiTree temp1 = NULL, temp2 = NULL;
    if (T)          //若当前结点 T 不为空
    {
        if (T->key == key)    //若当前结点 T 的关键字即为所找
            return T;      //则直接返回当前结点指针
        if (T->lchild)      //若当前结点有左孩子结点
            temp1 = Vertex(T->lchild, key);    //递归寻找结点并返回给 temp1
        if (temp1)          //若 temp1 不为空则已找到结点
            return temp1;           //返回 temp1 所指结点指针
        if (T->rchild)          //若左子树未找到，且有右子树
            temp2 = Vertex(T->rchild, key);          //递归寻找结点并返回给 temp2
        return temp2;           //返回 temp2
    }
    return NULL;    //若根结点 T 为空直接返回 NULL
}

ElemType Value(BiTree T, KeyElem key)
{   //获得结点值：返回二叉树 T 中关键字为 key 的结点的数据值，未找到返回-
1
    BiTree temp = Vertex(T, key); //temp 为指向 key 结点的指针
    if (temp)           //若 temp 不为空
        return temp->data;          //则返回结点的数据
    return -1;          //若 temp 为空则返回-1（表示未找到）
}

status Assign(BiTree T, KeyElem key, ElemType value)
{   //结点赋值：给二叉树 T 中关键字为 key 的结点赋值 value
    BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
    if (temp)           //若 temp 不为空
    {
        temp->data = value;         //结点的数据重新赋值
        return OK;          //返回成功
    }
    return ERROR;         //未找到返回失败
}

BiTree Parent(BiTree T, KeyElem key)
{   //获得双亲结点：返回二叉树 T 中关键字为 key 的结点的双亲结点指针，其
他情况返回 NULL
    BiTree temp1 = NULL, temp2 = NULL;
    if (T)    //若当前节点 T 不为空
```

```
    {
        if (T->lchild && T->lchild->key == key)
            return T;      //若 T 有左子树且左子树的关键字为 key 则返回 T
        if (T->rchild && T->rchild->key == key)
            return T;      //若 T 有右子树且右子树的关键字为 key 则返回 T
        if (T->lchild)     //若 T 有左孩子
            temp1 = Parent(T->lchild, key);          //递归寻找双亲
        if (temp1)
            return temp1;     //若在左子树找到双亲则返回
        if (T->rchild)           //否则在右子树查找并返回
            temp2 = Parent(T->rchild, key);
        return temp2;
    }
    return NULL;          //若根结点 T 为空直接返回 NULL
}

BiTree LeftChild(BiTree T, KeyElem key)
{   //获得左孩子结点：返回二叉树 T 中关键字为 key 的结点的左孩子结点指针
    BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
    if (temp)
        return temp->lchild;        //temp 不为空返回左孩子指针
    return NULL;          //未找到返回 NULL
}

BiTree RightChild(BiTree T, KeyElem key)
{   //获得右孩子结点：返回二叉树 T 中关键字为 key 的结点的右孩子结点指针
    BiTree temp = Vertex(T, key);       //temp 为指向 key 结点的指针
    if (temp)
        return temp->rchild;        //temp 不为空返回右孩子指针
    return NULL;          //未找到返回 NULL
}

BiTree LeftSibling(BiTree T, KeyElem key)
{   //获得左兄弟结点：返回二叉树 T 中关键字为 key 的结点的左兄弟结点指针
    BiTree dad = Parent(T, key);  //dad 为 key 结点的双亲结点
    if (dad && dad->lchild && dad->lchild->key != key)//若 key 结点有双亲结点、
双亲结点有左孩子且左孩子不为 key 结点
        return dad->lchild;          //返回双亲结点的左孩子
    return NULL;          //其他情况返回 NULL
}

BiTree RightSibling(BiTree T, KeyElem key)
{   //获得右兄弟结点：返回二叉树 T 中关键字为 key 的结点的右兄弟结点指针
    BiTree dad = Parent(T, key);  //dad 为 key 结点的双亲结点
```

```
    if (dad && dad->rchild && dad->rchild->key != key)//若 key 结点有双亲结点、双亲结点有右孩子且右孩子不为 key 结点
        return dad->rchild;         //返回双亲结点的右孩子
    return NULL;            //其他情况返回 NULL
}

status InsertChild(BiTree T, KeyElem key, int LR, BiTree c)
{   //插入子树：在二叉树 T 中 key 结点的 LR（0 为左/1 为右）子树插入子树 c
    BiTree p = Vertex(T, key);          //p 指向 key 结点
    if (!p)         //若结点未找到则返回失败
        return ERROR;
    if (LR)         //插入到左子树
    {
        c->rchild = p->rchild;
        p->rchild = c;
    }
    else        //插入到右子树
    {
        c->rchild = p->lchild;
        p->lchild = c;
    }
    return OK;          //返回成功
}

status DeleteChild(BiTree T, KeyElem key, int LR)
{   //删除子树：删除二叉树 T 中关键字为 key 的结点的 LR 子树
    BiTree p = Vertex(T, key);          //p 为 key 结点
    if (!p)         //若结点未找到返回失败
        return ERROR;
    if (LR && p->rchild) //删除左子树
        ClearBiTree(&p->rchild);
    else if (p->lchild)         //删除右子树
        ClearBiTree(&p->lchild);
    return OK;          //返回成功
}

void PreOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{   //先序遍历
    if (T)          //当前结点不为空
    {
        visit(T->key, T->data);         //先访问数据
        PreOrderTraverse(T->lchild, visit);     //在依次递归先序遍历左右子树
        PreOrderTraverse(T->rchild, visit);
    }
```

```
}

void InOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{    //中序遍历
    if (T)          //当前结点不为空
    {
        InOrderTraverse(T->lchild, visit);        //先递归中序遍历左子树
        visit(T->key, T->data);            //再访问当前节点数据
        InOrderTraverse(T->rchild, visit);        //最后递归中序遍历右子树
    }
}

void PostOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{    //后序遍历
    if (T)          //当前结点不为空
    {
        PostOrderTraverse(T->lchild, visit);    //先依次递归后序遍历左右子树
        PostOrderTraverse(T->rchild, visit);
        visit(T->key, T->data);            //最后访问数据
    }
}

void LevelOrderTraverse(BiTree T, void visit(KeyElem, ElemType))
{    //层序遍历
    BiTree temp = T;//temp 指向二叉树根结点
    LinkQueue Q;
    if (!T)
        return;
    InitQueue(&Q);  //初始化队列 Q
    do {
        visit(temp->key, temp->data); //先访问当前结点
        if (temp->lchild)       //若当前结点有左孩子则左孩子入队
            EnQueue(&Q, temp->lchild);
        if (temp->rchild)       //若当前节点有右孩子则右孩子入队
            EnQueue(&Q, temp->rchild);
    } while (DeQueue(&Q, &temp));  //若队列能出队一个元素则继续循环，出队
元素赋值给 temp
    DestroyQueue(&Q);
}

void SaveBiTNode(BiTree T, FILE* fp)
{    //将结点 T 中数据存入文件
    KeyElem nullkey = NULLNODE;       //nullkey 为空结点标识
    if (!T)         //若当前节点 T 为空则仅读入空节点标识
```

```
    {
        fwrite(&nullkey, sizeof(KeyElem), 1, fp);
        return;
    }
    fwrite(&T->key, sizeof(KeyElem), 1, fp);   //结点 T 不为空则依次读入关键字和数据
    fwrite(&T->data, sizeof(ElemType), 1, fp);
    SaveBiTNode(T->lchild, fp);        //递归将当前节点左右孩子存入文件
    SaveBiTNode(T->rchild, fp);
}

status SaveBiTreeElem(BiTList R)
{   //将二叉树 R 的数据存入文件
    FILE* fp;
    char filename[30];
    printf("input file name: ");           //输入文件名
    scanf("%s", filename);
    if ((fp = fopen(filename, "wb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    SaveBiTNode(R.Root, fp);    //从根结点递归存入文件
    fclose(fp);
    return OK;
}

void LoadBiTNode(BiTree* T, FILE* fp)
{   //从文件加载到二叉树结点 T
    KeyElem tempkey;
    ElemType tempdata;
    if (fread(&tempkey, sizeof(KeyElem), 1, fp))   //首先读取一个关键字
    {
        if (tempkey == NULLNODE)       //若读入的为空节点标识
        {
            *T = NULL;       //则当前结点为空并返回
            return;
        }
        fread(&tempdata, sizeof(ElemType), 1, fp);       //若当前节点不为空则继续将数据读入
        (*T) = (BiTree)malloc(sizeof(BiTNode));         //创建一个二叉树结点
        (*T)->key = tempkey;           //结点赋值
        (*T)->data = tempdata;
        LoadBiTNode(&(*T)->lchild, fp);       //递归加载当前结点的左右子树
```

```
        LoadBiTNode(&(*T)->rchild, fp);
    }
}

status LoadBiTreeElem(BiTList** R, int num, char* filename)
{   //从 filename 文件中读入数据到二叉树 R[num]
    FILE* fp;

    if (!R[num]) //若二叉树 R[num]不存在则初始化二叉树
        InitBiTree(&R[num]);
    else //若已存在则清空树
        ClearBiTree(&R[num]->Root);
    if ((fp = fopen(filename, "rb")) == NULL)
    {
        printf("File open error!\n ");
        return ERROR;
    }
    LoadBiTNode(&R[num]->Root, fp);          //从根结点递归加载数据到结点
    fclose(fp);
    return OK;
}
```