華中科技大學

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级 CS2311

学　　号 U202211820

姓　　名 赵晓烨

指导教师 许贵平

报告日期 2024 年 5 月 20 日

计算机科学与技术学院

目　　录

1 基于链式存储结构的线性表实现

1.1 问题描述

1. 实验要求：

   实现对链表的基本操作；

   设计演示系统完成实验。

2. 实验目的：

   加深对线性表的概念、基本运算的理解；

   熟练掌握线性表的逻辑结构与物理结构的关系；

   熟练掌握线性表的基本运算的实现。

1.2 系统设计

1. 链表数据结构定义

```
typedef struct LNode{单链表（链式结构）结点的定义
        ElemType data;
        struct LNode *next;
}LNode,*LinkList;
```

2. 初始化表

```
        status InitList(LinkList& L)线性表初始化
        {
                if(L){
                        return INFEASIBLE;
                }
                L= (LNode *) malloc(sizeof( LNode ));
                L->data=0;
                L->next=NULL;
                return OK;
        }
```

3. 删除表

```
status DestroyList(LinkList& L)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        LNode *p=L,*q;
        while(p){
                q=p->next;
                free(p);
                p=q;
        }
        L=NULL;
        return OK;
}
```

4. 清空表

```
status ClearList(LinkList& L)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        LNode *p = L->next;
        LNode *q;
        while(p)
        {
                q = p->next;
                free(p);
                p = q;
        }
        L->next = NULL;
```

```
        return OK;
    }
```

5. 表判空

```
    status ListEmpty(LinkList L)
    {
        if(L==NULL){
            return INFEASIBLE;
        }
        if(L->next==NULL) return TRUE;
        return FALSE;
    }
```

6. 求表长

```
status ListLength(LinkList L)
{
    if(L==NULL){
        return INFEASIBLE;
    }
    LNode *p = L->next;
    int length=0;
    while(p){
        length++;
        p=p->next;
    }
    return length;
}
```

7. 获取元素

```
status GetElem(LinkList L,int i,ElemType &e)
{
    if(L==NULL){
```

```
            return INFEASIBLE;
        }
        LNode *p = L->next,*q = L->next;
        int length=0,num=0;
        while(p){
            length++;
            p=p->next;
        }
        if(i<1||i>length) return ERROR;
        while(q){
            num++;
            if(num==i){
                e=q->data;
                return OK;
            }
            q=q->next;
        }
        return OK;
}
```

8. 查找元素

```
status LocateElem(LinkList L,ElemType e)
{
        if(L==NULL){
            return INFEASIBLE;
        }
        int length=0,num=0;
        LNode *p = L->next,*q = L->next;
        while(p){
            length++;
            p=p->next;
```

```
        }
        while(q){
                num++;
                if(q->data==e) return num;
                q=q->next;
        }
        if(num==length) return ERROR;
        return OK;
}
```

9. 求元素前驱

```
status PriorElem(LinkList L,ElemType e,ElemType &pre)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        int length=0,num=0;
        LNode *p=L ,*q = L->next;
        while(q){
                length++;
                q=q->next;
        }
        q = L->next;
        while(q){
                num++;
                if(q->data==e&&num!=1) {pre=p->data;return OK;}
                if(q->data==e&&num==1) break;
                q=q->next;
                p=p->next;
        }
        if(num==length) return ERROR;
```

```
        else if(num==1) return 12;
        return OK;
}
```

10. 求元素后继

```
        status NextElem(LinkList L,ElemType e,ElemType &next)
        {
                if(L==NULL){
                        return INFEASIBLE;
                }
                int length=0,num=0;
                LNode *q=L->next ,*p = L->next;
                while(p){
                        length++;
                        p=p->next;
                }
                if(L->next==NULL||q->next==NULL) return ERROR;
                p = L->next->next;
                while(q){
                        num++;
                        if(q->data==e&&num!=length) {
                                next=p->data;
                                return OK;
                                }
                        if(q->data!=e&&num==length) return ERROR;
                        if(q->data==e&&num==length) return 13;
                        q=q->next;
                        p=p->next;
                }
                return OK;
        }
```

11. 插入元素

```
status ListInsert(LinkList &L,int i,ElemType e)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        int length=0,num=0;
        LNode *p=L ,*q = L->next;
        while(q){
                length++;
                q=q->next;
        }
        q = L->next;
        if(i<1||i>length&&i!=length+1) return ERROR;
        while(q){
                num++;
                if(num==i) {
                        LNode *fp= (LNode*)malloc(sizeof(
                            LNode));
                        p->next=fp;
                        fp->data=e;
                        fp->next=q;
                        return OK;}
                q=q->next;
                p=p->next;
        }
        LNode *fp1= (LNode *) malloc(sizeof( LNode ));
        p->next=fp1;
        fp1->data=e;
        fp1->next=NULL;
        return OK;
```

```
    }
```

12. 删除元素

```
    status ListDelete(LinkList &L,int i,ElemType &e)
    {
        if(L==NULL){
            return INFEASIBLE;
        }
        int length=0,num=0;
        LNode *p=L ,*q = L->next;
        while(q){
            length++;
            q=q->next;
        }
        if(i<1||i>length) return ERROR;
        q = L->next;
        while(q){
            num++;
            if(num==i) {
                p->next=q->next;
                e=q->data;
                free(q);
                return OK;}

            q=q->next;
            p=p->next;
        }
        return OK;
    }
```

13. 遍历

```
status ListTraverse(LinkList L)
```

```
{
        if(L==NULL){
                return INFEASIBLE;
        }
        if(L->next==NULL) return OK;
        LNode *p=L->next;
        while (p){
                printf("%d ",p->data);
                p=p->next;
        }
        printf("\n");
        return OK;
}
        }
```

14. 链表翻转

```
status reverseList(LinkList L)链表翻转
{
        if(L==NULL){
                return INFEASIBLE;
        }
        LinkList p=L->next,newHead = NULL;
        while (p != NULL) {
                LinkList remain = p->next;
                p->next = newHead;
                newHead = p;
                p = remain;
        }
        L->next=newHead;
        return OK;
}
```

15. 删除链表的倒数第 n 个结点

```
status RemoveNthFromEnd(LinkList L,int n,ElemType &e)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        int length=ListLength(L);
        if(n<1||n>length) return ERROR;
        ListDelete(L,length-n+1,e);
        return OK;
}
```

16. 链表排序

```
status sortList(LinkList L)
{
        if(L==NULL){
                return INFEASIBLE;
        }
        struct LNode *q,*tail;
        int tmp;
        for(tail=NULL;L!=tail;tail=q){
                for(q=L;q->next!=tail;q=q->next){
                        if(q->data>q->next->data&&q!=L){
                                tmp=q->data;
                                q->data=q->next->data;
                                q->next->data=tmp;
                        }
                }
        }
        return OK;
}
```

17. 保存文件

```
status SaveList(LinkList L,char FileName[])
{


        if(L==NULL){
                return INFEASIBLE;
        }
        LNode *p=L->next;
        FILE *fp = fopen(FileName , "w");
        if (fp == NULL)
        {
                puts("Fail to open file!");
                exit(1);
        }
        while(p){
                fprintf(fp,"%d ",p->data);
                p=p->next;
        }
        fclose(fp);
        return OK;
}
```

18. 加载文件

```
status LoadList(LinkList &L,char FileName[])
{
        if(L){
                return INFEASIBLE;
        }
        L= (LNode *) malloc(sizeof( LNode ));
        LNode *h=L;
        FILE* fp = fopen(FileName , "r");
```

```
        if (fp == NULL)
        {
                puts("Fail to open file!");
                exit(1);
        }

        while(!feof(fp)){
                LNode *hp= (LNode *) malloc(sizeof( LNode ));
                fscanf(fp,"%d ",&hp->data);
                h->next=hp;
                hp->next=NULL;
                h=hp;
        }
        fclose(fp);
        return OK;
}
```

1.3 系统实现

1.3.1 代码的组织结构

演示系统以一个菜单作为交互界面，用户通过输入命令对应的编号来调用相应的函数来实现创建表，销毁表，清空表，插入元素，删除元素，求表长，判空表，求前驱，求后继，遍历链表等基本操作，以及保存为文件，加载文件，翻转链表，对链表排序等进阶操作。程序主函数为一个 switch 结构，根据输入的数字，执行不同的语句，进而调用不同的函数。交互界面如下图

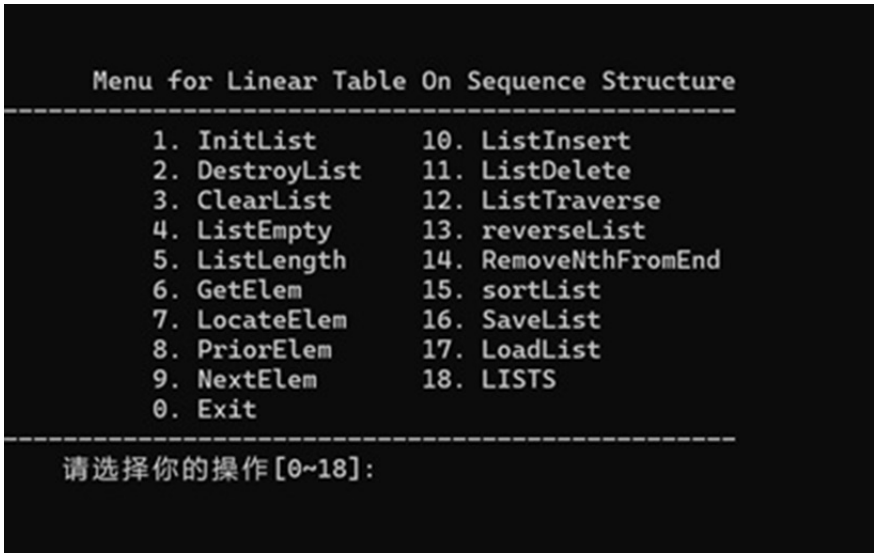


图 1-1　交互界面

1.4 系统测试

程序开发及实现环境：Win11 下使用 Visual Studio Code 进行编译和调试，开发语言为 C 语言。表1-1为正常样例测试的输入，预期结果与实际输出。

表 1-1　正常样例测试

函数	输入	实际输出	预期结果
初始化表	1	1 线性表创建成功	线性表创建成功
销毁表	2	2 线性表销毁成功	线性表销毁成功
销毁表	2	2 线性表不存在	线性表不存在
初始化表	1	1 线性表创建成功	线性表创建成功
线性表判空	4	4 线性表为空	线性表为空
插入元素	10 1 1	10 1 1 插入成功	插入成功
插入元素	10 2 5	10 2 5 插入成功	插入成功
插入元素	10 3 7	10 3 7 插入成功	插入成功
求表长	5	5 线性表长度为3	线性表长度为 3
获取元素	6 2	6 想要获取第几个元素： 2 获取元素为5	获取元素为 5
查找元素	7 7	7 请输入元素：7 元素位置为3	元素位置为 3
查找前驱	8 5	8 请输入元素：5 元素的前驱为1	元素的前驱为 1
查找后继	9 5	9 请输入元素：5 元素的后继为7	元素的后继为 7
遍历	12	12 1 5 7 遍历完成	1 5 7 遍历完成
删除元素	11 2	11 想要删除第几个元素：2 删除成功，被删除的元素为：5	删除元素 5
保存文件	16 123	请输入文件名 123 写入文件完成	保存文件成功
销毁表	2	2 线性表销毁成功	线性表销毁成功
读取文件	17 F”:/shiyan	请输入文件名 123 读入文件完成	读入文件成功
遍历链表	12	12 1 7 遍历完成	1 7 遍历完成

1.5 实验小结

本次实验让我对基于链式存储结构的线性表的了解更进了一步。

演示系统的搭建，让我体会到了主函数和子函数的关系，以及如何搭建一个可以调用不同模块的系统。

在编写插入和删除乃至翻转链表的函数时，如何有条理地更改指针的指向是一大难点，比如在插入新结点时，必须先让新节点的指针域指向 p 指针所指元素的 next，然后才能让 p 指向新节点，反之则会大错特错。经过这次实验，我明白了赋值顺序对程序的巨大影响。

总的来说，本次数据结构实验提高了我的编程能力，让我对系统整体设计有了更深的认识。

2 基于二叉链表的二叉树实现

2.1 问题描述

1. 实验要求：

   实现对树的基本操作；

   设计演示系统完成实验。

2. 实验目的：

   加深对树的概念、基本运算的理解；

   熟练掌握树的逻辑结构与物理结构的关系；

   熟练掌握树的基本运算的实现。

2.2 系统设计

1. 树数据结构定义

```
typedef struct {
        KeyType key;
        char others[20];
} TElemType; 二叉树结点类型定义

typedef struct BiTNode{二叉链表结点的定义
        TElemType data;
        struct BiTNode *lchild,*rchild;
} BiTNode, *BiTree;
```

2. 创建二叉树

```
检测冲突辅助函数
int jiance(TElemType definition[])
{
        int a[10000]={0},b=0,j;
        memset(a, 0, 1000);
        while(definition[b].key!=-1){
```

```
                a[definition[b].key]++;
                b++;
        }
        for(j=1;j<1000;j++){
                if(a[j]>1) return 0;
        }
        return 1;
}
status CreateBiTree(BiTree& T, TElemType definition[],int &i)
{

        if (jiance(definition)==0) return ERROR;
        BiTree ans;
        ans = (BiTree)malloc(sizeof(BiTNode));
        T = ans;
        memset(num, 0, 1000);
        if (definition[0].key == 0|| definition[0].key == -1) {
                T = NULL;
                return OK;
        }
        i = 0;

        T = (BiTree)malloc(sizeof(BiTNode));
        T->data.key = definition[i].key;
        strcpy(T->data.others, definition[i].others);
        num[definition[i].key] = 1;
        i++;
        if (ERROR == CreateBiTNode(T->lchild, definition, num, i
           ) || ERROR == CreateBiTNode(T->rchild, definition,
           num, i))
        return ERROR;
```

```
        return OK;
}
status CreateBiTNode(BiTree& T, TElemType definition[], int num
    [], int& i) {

        if (definition[i].key == 0 || definition[i].key == -1) {
                T = NULL;
                i += definition[i].key + 1;
                return OK;
        }
        T = (BiTree)malloc(sizeof(BiTNode));
        T->data.key = definition[i].key;
        strcpy(T->data.others, definition[i].others);
        num[definition[i].key] = 1;
        i++;
        if (ERROR == CreateBiTNode(T->lchild, definition, num, i
            ) || ERROR == CreateBiTNode(T->rchild, definition,
            num, i))
        return ERROR;
        return OK;
}
```

3. 销毁树

```
status DestroyBiTree(BiTree &T)
{
        if (T)
        {
                DestroyBiTree(T->lchild);
                DestroyBiTree(T->rchild);
                free(T);
                T=NULL;
```

```
        }
        return OK;
}
```

4. 清空树

```
status ClearBiTree(BiTree &T)
将二叉树设置成空，并删除所有结点，释放结点空间
{
        if(T)
        {
                ClearBiTree(T->lchild);
                ClearBiTree(T->rchild);
                free(T);
        }
        T=NULL;
        return OK;
}
```

5. 树判空

```
        status BiTreeEmpty(BiTree T)
{
        if(T)
        return FALSE;
        else
        return TRUE;
}
```

6. 求树的深度

```
int BiTreeDepth(BiTree T)
求二叉树T的深度
{
        if(T==NULL) return 0;
```

```
        else{
return 1 + max(BiTreeDepth(T->lchild),BiTreeDepth(T->rchild));
        }
}
```

7. 查找结点

```
BiTNode* LocateNode(BiTree root,KeyType value)
查找结点
{
        if(root == NULL){
                return NULL;
        }
        else if(root != NULL && root->data.key ==value ){
                return root;
        }
        else{
                BiTree no1 = LocateNode(root->lchild,value);
                BiTree no2 = LocateNode(root->rchild,value);
                if(no1 != NULL && no1->data.key==value){
                        return no1;
                }else if(no2 != NULL && no2->data.key==value){
                        return no2;
                }
                else{
                        return NULL ;
                }
        }
}
```

8. 结点赋值

```
void LocateNode1(BiTree T,KeyType e,int &i)
唯一性判断
```

```
{
        if(T==NULL) return ;
        if(e==T->data.key){
                i++;
        }
        LocateNode1(T->lchild,e,i);
        LocateNode1(T->rchild,e,i);
}

status Assign(BiTree &T,KeyType e,TElemType value)
实现结点赋值。
{
        BiTree q;
        int i=0;
        TElemType j;
        q=LocateNode(T,e);
        if(q==NULL) return ERROR;
        else {
                j=q->data;
                q->data=value;
        }
        LocateNode1(T,value.key,i);
        if(i<=1) return OK;
        else{
                q->data=j;
                return 13;
        }
}
```

9. 获得兄弟结点

```
BiTNode* GetSibling(BiTree T,KeyType e)
```

```
实现获得兄弟结点
{
        if(T->data.key==e) return NULL;
        BiTree q=LocateNode(T,e);
        BiTree T1=Parent(T,q);
        if(T1->lchild==q){
                return T1->rchild;
        }
        if(T1->rchild==q){
                return T1->lchild;
        }
        return NULL;
}
```

10. 插入结点

```
status InsertNode(BiTree &T,KeyType e,int LR,TElemType c)
插入结点
{
        BiTNode* p=LocateNode(T,e);
        if(p==NULL) return ERROR;
        else if(LocateNode(T,c.key)!=NULL)
        return ERROR; 检查是否有相同的关键字
        if(LR==-1)
        {
                BiTNode* q=(BiTNode*)malloc(sizeof(BiTNode));
                q->data=c;
                q->lchild=NULL;
                q->rchild=T;
                T=q;
        }
        if(LR==0)
```

```
        {
                BiTNode* q=(BiTNode*)malloc(sizeof(BiTNode));
                q->data=c;
                q->lchild=NULL;
                q->rchild=p->lchild;
                p->lchild=q;
        }
        if(LR==1)
        {
                BiTNode* q=(BiTNode*)malloc(sizeof(BiTNode));
                q->data=c;
                q->lchild=NULL;
                q->rchild=p->rchild;
                p->rchild=q;
        }
        return OK;
}
```

11. 删除结点

```
status DeleteNode(BiTree &T,KeyType e)
{


        if (T == NULL) {
                return ERROR;
        }
        BiTree nodeToDelete = LocateNode(T, e);
        BiTree q1=T;
        if(T->data.key==e){
                if(T->lchild==NULL&&T->rchild==NULL){
                        T=NULL;
                }
```

```
        if(T->lchild&&T->rchild==NULL){
                T=T->lchild;
                free(q1);
        }
        if(T->rchild&&T->lchild==NULL){
                T=T->rchild;
                free(q1);
        }
        if(T->rchild&&T->lchild){
                BiTree q=T->lchild;
                while(q){
                        if(q->rchild){
                                q=q->rchild;
                        }
                        if(q->rchild==NULL&&q->lchild){
                                q=q->lchild;
                        }
                if(q->rchild==NULL&&q->lchild==NULL){
                                break;
                        }
                }
                q->rchild=T->rchild;
                T=T->lchild;
                free(q1);
        }
        return OK;
}
if (nodeToDelete == NULL) {
        return ERROR;
}
```

```
        BiTree parentNode = Parent(T, nodeToDelete);
        if (nodeToDelete->lchild == NULL && nodeToDelete->rchild
            == NULL) {

                if (parentNode->lchild == nodeToDelete) {
                        parentNode->lchild = NULL;
                } else {
                        parentNode->rchild = NULL;
                }
        }
else if (nodeToDelete->lchild == NULL || nodeToDelete->rchild
   == NULL) {

BiTree childNode = (nodeToDelete->lchild != NULL) ?
   nodeToDelete->lchild : nodeToDelete->rchild;
                if (parentNode == NULL) {
                        T = childNode;
                } else if (parentNode->lchild == nodeToDelete) {
                        parentNode->lchild = childNode;
                } else {
                        parentNode->rchild = childNode;
                }
        }
        else{

                BiTree maxleft=nodeToDelete->lchild;
                while(maxleft){
                        if(maxleft->rchild){
                                maxleft=maxleft->rchild;
                        }
                        if(maxleft->rchild==NULL&&maxleft->lchild)
```

```
                    {
                        maxleft=maxleft->lchild;
                    }
                    if(maxleft->rchild==NULL&&maxleft->lchild
                    ==NULL){
                        break;
                    }
                }
                maxleft->lchild = nodeToDelete->lchild;
                maxleft->rchild = nodeToDelete->rchild;
                if (parentNode->lchild == nodeToDelete) {
                    parentNode->lchild = maxleft;
                } else {
                    parentNode->rchild = maxleft;
                }


        }
        free(nodeToDelete);
        return OK;

}
```

12. 先序遍历

```
status PreOrderTraverse(BiTree T,void (*visit)(BiTree))
{
        if (T == NULL) {
                return OK;
        }
        visit(T);
        if (PreOrderTraverse(T->lchild, visit) == ERROR) {
```

```
            return ERROR;
        }
        if (PreOrderTraverse(T->rchild, visit) == ERROR) {
            return ERROR;
        }
        return OK;
}
```

13. 中序遍历

```
status InOrderTraverse(BiTree T,void (*visit)(BiTree))
{
        if (T == NULL) {
            return OK;
        }
        if (InOrderTraverse(T->lchild, visit) == ERROR) {
            return ERROR;
        }
        visit(T);

        if (InOrderTraverse(T->rchild, visit) == ERROR) {
            return ERROR;
        }
        return OK;
}
```

14. 后序遍历

```
status PostOrderTraverse(BiTree T,void (*visit)(BiTree))

{
        if (T == NULL) {
            return OK;
        }
```

```
        if (PostOrderTraverse(T->lchild, visit) == ERROR) {
                return ERROR;
        }

        if (PostOrderTraverse(T->rchild, visit) == ERROR) {
                return ERROR;
        }
        visit(T);

        return OK;
}
```

15. 层序遍历（队列实现）

```
初始化队列
status InitQueue(LinkQueue* Q) {
        Q->front = Q->rear = (QueuePtr)malloc(sizeof(QueueNode))
            ;
        if (!Q->front) return ERROR;
        Q->front->next = NULL;
        return OK;
}

入队
status EnQueue(LinkQueue* Q, BiTree e) {
        QueuePtr newNode = (QueuePtr)malloc(sizeof(QueueNode));
        if (!newNode) return ERROR;
        newNode->data = e;
        newNode->next = NULL;
        Q->rear->next = newNode;
        Q->rear = newNode;
        return OK;
```

```
}

出队
status DeQueue(LinkQueue* Q, BiTree* e) {
        if (Q->front == Q->rear) return ERROR;
        QueuePtr p = Q->front->next;
        *e = p->data;
        Q->front->next = p->next;
        if (Q->rear == p) Q->rear = Q->front;
        free(p);
        return OK;
}

判断队列是否为空
int QueueEmpty(LinkQueue Q) {
        return Q.front == Q.rear;
}

层次遍历函数

status LevelOrderTraverse(BiTree T,void (*visit)(BiTree))
按层遍历二叉树T
{

        if (T == NULL) return OK;

        LinkQueue Q;
        if (InitQueue(&Q) == ERROR) return ERROR;

        if (EnQueue(&Q, T) == ERROR) return ERROR;
```

```
        while (!QueueEmpty(Q)) {
                BiTree node;
                if (DeQueue(&Q, &node) == ERROR) return ERROR;
                visit(node); 访问当前节点

                if (node->lchild != NULL) {
                        if (EnQueue(&Q, node->lchild) == ERROR)
                           return ERROR;
                }

                if (node->rchild != NULL) {
                        if (EnQueue(&Q, node->rchild) == ERROR)
                           return ERROR;
                }
        }

        return OK;

}
```

16. 求最大路径和

```
int MaxPathSum(int& self,BiTree root,int Sum)
{
        if (root == NULL) return self;
        if ((root->lchild != NULL) && (root->rchild != NULL)){
                MaxPathSum(self,root->lchild, Sum + root->data.
                   key);
                MaxPathSum(self,root->rchild, Sum + root->data.
                   key);
        }
        else if ((root->lchild != NULL) && (root->rchild == NULL
```

```
        ))
        MaxPathSum(self,root->lchild, Sum + root->data.key);
        else if ((root->lchild == NULL) &&(root->rchild != NULL)
        )
        MaxPathSum(self,root->rchild, Sum + root->data.key);
        else{
                if ((Sum + root->data.key) > self)
                self = Sum + root->data.key;
        }
        return OK     ;
}
```

17. 反转二叉树

```
void InvertTree(BiTree &T)
{
        if(T==NULL) return ;
        BiTree p=T->lchild;
        T->lchild=T->rchild;
        T->rchild=p;
        InvertTree(T->lchild);
        InvertTree(T->rchild);
}
```

18. 保存文件

```
status SaveBiTree(BiTree T, char FileName[])
将二叉树的结点数据写入到文件FileName中
{
        FILE* fp;
        if ((fp = fopen(FileName, "w+")) == NULL) {
                printf("File open erroe\n ");
                return ERROR;
        };文件写入失败
```

```
        BiTNode* stack[100], * now = T;
        int top = 0;
        stack[0] = (BiTNode*)malloc(sizeof(BiTNode));
        stack[0]->lchild = stack[0]->rchild = NULL;
        while (top > 0 || now) {输出带空结点的二叉树前序遍历序列
                if (now) {
                        fprintf(fp, "%d %s ", now->data.key, now->
                            data.others);
                        top++;
                        stack[top] = now;
                        now = now->lchild;
                }
                else {
                        fprintf(fp, "0 null ");
                        now = stack[top--]->rchild;
                }
        }
        fprintf(fp, "0 null -1 null ");
        fclose(fp);
        return OK;
}
```

19. 加载文件

```
status LoadBiTree(BiTree& T, char FileName[])
读入文件FileName的结点数据，创建二叉树
{
        if (NULL != T) return INFEASIBLE;
        FILE* fp;
        if ((fp = fopen(FileName, "r+")) == NULL) {
                printf("File open erroe\n ");
                exit(-1);
```

```
        };文件打开失败
        TElemType definition[100];
        int q = 0;
        do {将带空结点的二叉树前序遍历序列存进definition
               fscanf(fp, "%d %s ", &definition[q].key,
                  definition[q].others);
        } while (definition[q++].key != -1);
        fclose(fp);
        if (OK == CreateBiTree(T, definition,i))创建二叉树
        return OK;
        return 0;
}
```

2.3 系统实现

2.3.1 代码的组织结构

演示系统以一个菜单作为交互界面，用户通过输入命令对应的编号来调用相应的函数来实现创建树，销毁树，清空树，查找结点，结点赋值，插入结点，删除结点，求兄弟结点，求树的深度，判空树，先序遍历，中序遍历，后序遍历，层序遍历等基本操作，以及保存为文件，加载文件，翻转树，求最大路径和等进阶操作。程序主函数为一个 switch 结构，根据输入的数字，执行不同的语句，进而调用不同的函数。交互界面如下图

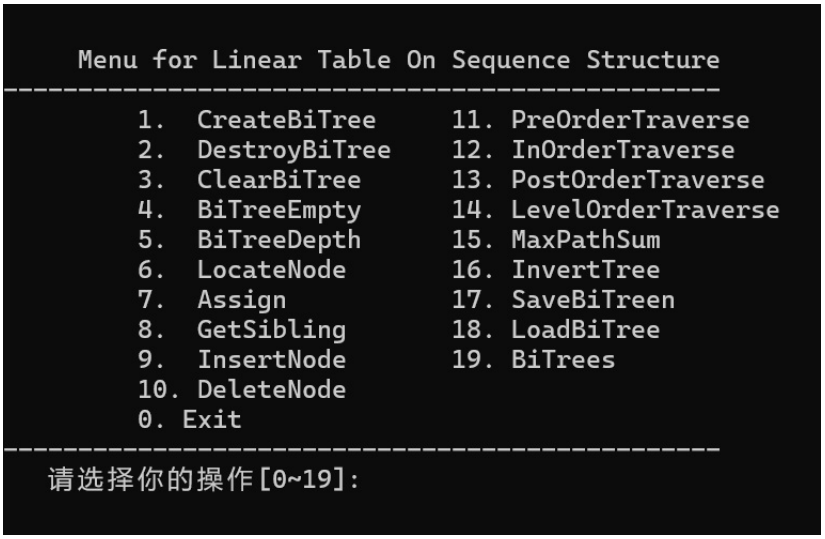


图 2-1 交互界面

2.4 系统测试

程序开发及实现环境：Win11 下使用 Visual Studio Code 进行编译和调试，开发语言为 C 语言。

表2-1为正常样例测试的输入，预期结果与实际输出

以 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 c 0 null 0 null -1 null

建立一棵二叉树二叉树如下图

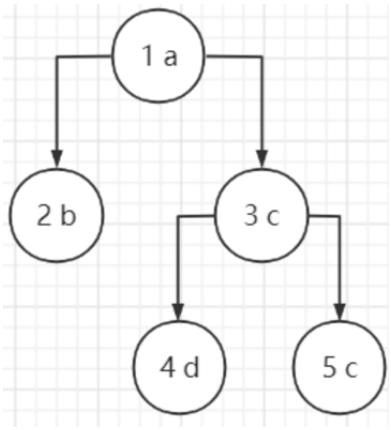
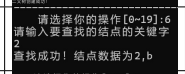
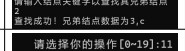
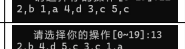
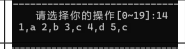
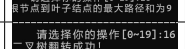
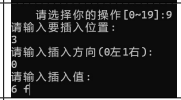
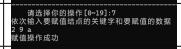
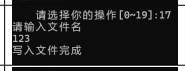
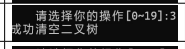
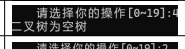


图 2-2 二叉树

表 2-1 正常样例测试

函数	输入	实际输出	预期结果
创建树	1 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 c 0 null 0 null -1 null		二叉树创建成功
查找结点	6 2		2 b
查找兄弟结点	8 2		查找成功 3,c
先序遍历	11		1,a 2,b 3,c 4,d 5,c
中序遍历	12		2,b 1,a 4,d 3,c 5,c
后序遍历	13		2,b 4,d 5,c 3,c 1,a
层序遍历	14		1,a 2,b 3,c 4,d 5,c
求最大路径和	15		根节点到叶子节点最大路径和为 9
翻转二叉树	16		翻转成功
插入结点	9 3 0 6 f		插入成功
（插入二叉树结点后）层序遍历	14		1,a 2,b 3,c 6,f 5,c 4,d
结点赋值	7 2 9 a		赋值操作成功
文件保存	17 123		写入文件完成
清空树	3		成功清空二叉树
树判空	4		二叉树为空树
销毁树	2		成功销毁二叉树

2.5 实验小结

本次实验让我对二叉树实现的了解更进了一步。

演示系统的搭建，让我体会到了主函数和子函数的关系，以及如何搭建一个可以调用不同模块的系统。

在编写插入和删除树的结点时，如何有效地根据关键字找到相应结点是使程序变得简介的一大关键，在编写结点赋值的函数时，如果能够在之前定义定位顶点的函数，并调用，将极大地省去冗杂的代码，这次实验让我对函数的工具性，模块性有了直观的感受。

总的来说，本次数据结构实验提高了我的编程能力，让我对系统整体设计有了更深的认识。

相比之前几次的实验，树的系统实现更为复杂，进阶操作的实现需要了解一些经典的算法，本次实验让我有了较大的进步。

3 课程的收获和建议

3.1 基于链式存储结构的线性表实现

通过对基于链式存储结构的线性表实现的演示系统练习，我基本掌握了线性表的基本操作，能够根据需要调用不同的模块来灵活地使用线性表这一数据结构。

同时在实验过程中，尤其是在 debug 的过程中，我明白了看书看懂了并不代表自己就掌握了一种数据结构，实际上还差的很远，唯有动手实践，多用样例测试，才能了解与熟练运用它。

许许多多的细节问题不通过实践，是无法学到的。

3.2 基于二叉链表的二叉树实现

通过对基于二叉链表的二叉树实现的演示系统练习，我基本掌握了二叉树的基本操作，同时对数据结构的认识更进了一步。

这门课程可以说是计算机专业的基础课程，也是核心课程。

经过一学期的学习，我希望未来数据结构课能够越来越好。

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] 严蔚敏等. 数据结构题集（C 语言版）. 清华大学出版社