

电子科技大学教务处制表

# 电子科技大学

## 实 验 报 告

学生姓名: Lolipop 学 号: 2018091202000 指导教师: xxx

实验地点: 清水河科技实验大楼 实验时间: 2019.6

一、实验室名称: 学校实验中心软件实验室

二、实验项目名称: 非线性结构及相关算法的设计与应用

第 1 部分: 二叉树的链式存储、序列化和反序列化;

第 2 部分: 公园景点间的最短路径查询程序的设计。

三、实验学时: 8 学时

第 1 部分: 二叉树的链式存储、序列化和反序列化

四、实验原理:

i. 树的定义

树是  $n(n \geq 0)$  个结点的有限集, 根据结点数可分为空树( $n=0$ )或非空树( $n>0$ ), 对于非空的树  $T$ , 有着两个定义:

- 1) 有且仅有一个称之为根的结点;
- 2) 除根结点以外的其余结点可分为  $m(m>0)$  个互不相交的有限集  $T_1, T_2, \dots, T_m$ , 其中每个集合本身又是一个树, 并且又称之为根的子树。

ii. 树的基本术语

- 1) 结点: 树中的一个独立单位。包含一个数据元素及若干指向其子树的分支, 如图中的每个字母都是一个结点。
- 2) 结点的度: 结点所拥有的子树数称为结点的度。

- 3) 树的度：树的度是树内各结点度的最大值。
- 4) 叶子：度为 0 的结点称为叶子或终端结点。
- 5) 非终端结点：度不为 0 的结点称为非终端结点或分支结点。除根结点之外，也称为内部结点。
- 6) 双亲和孩子：若一个结点含有子结点，则这个结点称为其子节点的双亲节点。
- 7) 兄弟：具有同个双亲的结点互为兄弟结点。
- 8) 祖先：从根到该结点所经分支上的所有结点。
- 9) 子孙：以某个结点为根的子树中的任一结点都是该根结点的子孙。
- 10) 结点层次：结点的层次从根开始定义，根节点为第一层，根节点的孩子结点为第二层，以此类推。
- 11) 堂兄弟：双亲在同一层的结点互为堂兄弟。
- 12) 树的深度：树中结点的最大层次称之为树的深度或高度。
- 13) 有序树和无序树：如果将树中结点的各子树看成从左至右是由次序的，则称该树的为有序树，否则为无序树。

### iii. 二叉树

二叉树是一种特殊的树，主要有着以下两个特点：

- 1) 有且仅有一个称之为根的结点；
- 2) 除根结点以外的其余结点分为两个互不相交的子集 T1 和 T2，分别称为 T 的左子树和右子树，且 T1 和 T2 本身又都是二叉树。

### iv. 二叉树的数据结构

二叉树的存储结构主要包含两种：

#### (1) 顺序存储结构

用一组地址连续的存储单元来存放二叉树的数据元素。使用顺序存储结构时，一般将会在二叉树上添加一些虚结点，构造成完全二叉树。虚节点对应一维数组中的特殊符号，例如 ‘#’。因此，一般二叉树采用顺序存储结构以后，二叉树中各节点的编号与等高度的完全二叉树中位置上结点编号相同。二叉树顺序存储结构的类型声明如下：

```
typedef ElemType SqBinTree[MAXSIZE]
```

#### (2) 链式存储结构（简称为二叉链）

用一个链表来存储一棵二叉树，二叉树中的每一个结点用链表中的

一个结点来存储。用 `data` 表示值域，用于存储对应的数据元素；用 `lchild` 和 `rchild` 分别表示其左指针域和右指针域。二叉链中通过根节点指针 `b` 来唯一标识整个存储结构，称为二叉树 `b`。二叉链中结点类型 `BTNode` 的声明如下：

```
typedef struct node
{
    ElemType data;
    struct node* lchild;
    struct node* rchild;
} BTNode;
```

#### v. 建立二叉树

利用 `ch` 扫描 `str[j]`，根据 `ch` 的值进行循环处理，直到读取结束。使用栈 `St[MAXSIZE]` 来保存双亲结点，`top` 为栈顶指针，`k` 作为标志指定处理的结点是左孩子（`k=1`）还是右孩子（`k=2`）。代码实现如下：

```
void CreateBTree(BTNode*& b, ElemType* str)
{
    BTNode *St[MAXSIZE], *p;
    int top = -1, k, j = 0;
    ElemType ch;
    b = NULL;
    ch = str[j];
    while(ch != '\0')
    {
        switch (ch)
        {
            case '(': top++; St[top] = p; k = 1; break;
            case ')': top--; break;
            case ',': k = 2; break;
            default:
                p = (BTNode*)malloc(sizeof(BTNode));
                p->data = ch;
                p->lchild = p->rchild = NULL;
                if (b == NULL)
```

```
        {
            b = p;
        }
        else
        {
            switch (k)
            {
                case 1: St[top]->lchild = p; break;
                case 2: St[top]->rchild = p; break;
            }
        }
    }
    j++;
    ch = str[j];
}
}
```

#### vi. 存储与读取二叉链序列

为了下一次调用需要的二叉链序列，要将序列存储到文件中。利用文件指针 `FILE* fp` 和 `fwrite()` 函数可以将字符串内容写入到 `dat` 文件中，代码实现如下：

```
void SaveToFile(char str[MAXSIZE])
{
    FILE* fp;
    if ((fp = fopen(FILE_PLACE, "wb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }
    fwrite(str, strlen(str), 1, fp);
    fclose(fp);
}
```

从文件中读取二叉链序列，可以使用文件指针 `FILE* fp` 和 `fread()` 函

数。为了确保读取完字符串并用 ‘\0’ 标志结尾，可以利用 `fseek()` 和 `ftell()` 函数来获取文件长度。实现代码如下：

```
void ReadFromFile(char str[MAXSIZE])
{
    FILE* fp;
    int len;
    if ((fp = fopen(FILE_PLACE, "rb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }
    //获取文件长度
    fseek(fp, 0, SEEK_END);
    len = ftell(fp);
    rewind(fp);
    //读取文件字符串内容
    fread(str, sizeof(char), len, fp);
    str[len] = '\0';
    fclose(fp);
}
```

#### vii. 先序序列化输出二叉树

利用递归的方法，当 `b` 不指向 `NULL` 时，将 `b` 的 `data` 域赋值给 `St[i]`，递归调用 `b->lchild` 和 `b->rchild` 遍历二叉树。实现代码如下：

```
int Translate(BTNode* b, ElemType St[MAXSIZE], int i)
{
    if (b == NULL)
    {
        St[i] = '#';
        i++;
        return i;
    }
    St[i] = b->data;
    i++;
```

```

        i = Translate(b->lchild, St, i);
        i = Translate(b->rchild, St, i);
        St[i] = '\0';
        return i;
    }

```

#### viii. 反序列化建立二叉树

用 *i* 从头扫描 *str*；采用先序方法，当 *i* 超界时返回 **NULL**；否则当遇到 ‘#’ 字符时返回 **NULL**，当遇到其它字符时，创建一个结点，可以采用递归的方法构造该二叉树；也可以采用非递归方法构造该二叉树。递归算法实现如下：

```

int RTransform(BTNode*& b, char St[MAXSIZE], int i)
{
    if (St[i] == '#' || St[i] == '\0')
    {
        b = NULL;
        i++;
        return i;
    }
    BTNode* p;
    p = (BTNode*)malloc(sizeof(BTNode));
    p->data = St[i];
    b = p;
    i++;
    i = RTransform(b->lchild, St, i);
    i = RTransform(b->rchild, St, i);
    return i;
}

```

非递归算法实现如下：

```

void Transform(BTNode*&b, char a[MAXSIZE])
{
    BTNode *St[MAXSIZE], *p;
    int top = -1;

```

```
int k=0,j=0,tem=0;
char ch;
b = NULL;
ch = a[j];
while (ch!=0)
{
    if (ch!='#')
    {
        top++;
        St[top] = p;
        p = (BTNode*)malloc(sizeof(BTNode));
        p->data = ch;
        p->rchild = NULL;
        p->lchild = NULL;
        if(b==NULL)
        {
            b = p;
        }
        else
        {
            if(k==0)
            {
                St[top]->lchild = p;
            }
            else
            {
                if(k==1)
                {
                    St[top]->rchild = p;
                    k=0;
                }
                if(k==2)
                {
                    St[tem]->rchild = p;
```

```

        k=0;
    }
}
}
}
if(ch=='#')
{
    if(k==0)
    {
        p->lchild = NULL;
        k=1;
    }
    else
    {
        p->rchild = NULL;
        tem = top;
        while(St[tem]->rchild!=NULL) tem--;
        k=2;
    }
}
j++;
ch=a[j];
}
}

```

**ix. 遍历输出二叉树序列**

输出二叉树的序列有三种常见算法：

- 1) 先序遍历。先访问根节点，再遍历左子树，最后遍历右子树；
- 2) 中序遍历。先遍历左子树，再访问根节点，最后遍历右子树；
- 3) 后序遍历。先遍历左子树，再遍历右子树，最后访问根节点。

其中，中序遍历非递归算法的实现，首先需要将根节点及其左下结点依次进栈。当达到根节点的最左下结点时，它是中序序列的开始结点，也是栈顶结点，出栈并访问它，然后转向它的右子树，对右子树的处理



与上述过程类似。实现代码如下：

```
void InOrder(BTNode* b)
{
    BTNode* p;
    SqStack* st;
    InitStack(st);
    p = b;
    while (!StackEmpty(st) || p != NULL)
    {
        while (p != NULL)
        {
            Push(st, p);
            p = p->lchild;
        }
        if (!StackEmpty(st))
        {
            Pop(st, p);
            printf("%c", p->data);
            p = p->rchild;
        }
    }
    printf("\n");
    DestorySt(st);
}
```

后序遍历非递归算法的实现，首先将根节点及其左下结点依次进栈。设置 bool 型变量 flag 判断当前结点是否为栈顶结点，在 do-while 循环的第一个 while 循环结束后开始处理栈顶结点，置 flag 为 true；转向处理右子树时，置 flag 为 false。设置指针变量 r=NULL，让它指向刚刚访问过的结点，一旦正在处理的栈顶结点的右子树=r，说明该栈顶结点的左、右子树已经遍历过了，接下来可以访问该结点。代码实现如下：

```
void PostOrder(BTNode* b)
{
    BTNode* p, * r;
```

```
bool flag;
SqStack* st;
InitStack(st);
p = b;
do
{
    while (p != NULL)
    {
        Push(st, p);
        p = p->lchild;
    }
    r = NULL;
    flag = true;
    while (!StackEmpty(st) && flag)
    {
        GetTop(st, p);
        if (p->rchild == r)
        {
            printf("%c", p->data);
            Pop(st, p);
            r = p;
        }
        else
        {
            p = p->rchild;
            flag = false;
        }
    }
} while (!StackEmpty(st));
printf("\n");
DestorySt(st);
}
```

#### x. 销毁二叉树

利用递归的方法，遍历并销毁二叉树的左子树和右子树，最后销毁根节点 b。代码实现如下：

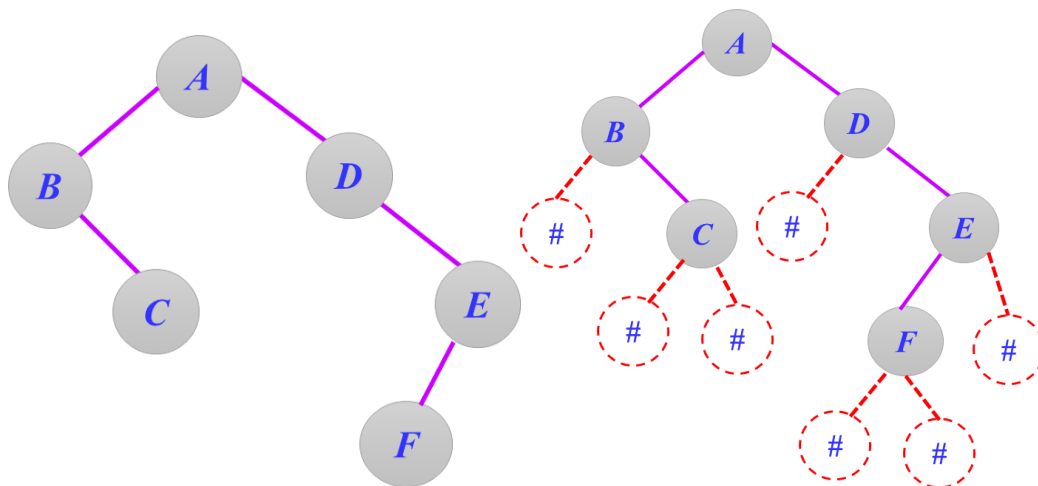
```
void DestoryBTree(BTNode*& b)
{
    if (b != NULL)
    {
        DestoryBTree(b->lchild);
        DestoryBTree(b->rchild);
        free(b);
    }
}
```

## 五、实验目的：

让学生通过实验，理解和掌握二叉树在计算机内存中的存储表示方法；掌握二叉树在磁盘文件中的链式存储方法；掌握二叉树序列化与反序列化的算法；让学生深入掌握二叉树的遍历和构造算法，初步具备解决复杂软件工程的基本知识和技能。

## 六、实验内容：

二叉树是由结点指针将多个结点关联起来的抽象数据结构，是存在于内存中的，不能进行持久化，如果需要将一颗二叉树的结构持久化保存在磁盘文件中，需要将其转换为字符串并保存到文件中。所谓序列化是对二叉树进行先序遍历产生一个字符序列，与一般的先序遍历不一样，需要记录空结点用#字符表示，并且假设序列中没有结点的值为#。如下图所示的序列化存储为：A,B,#,C,##,D,#,E,F,##,##



所谓反序列化就是通过先序序列化的结果串 `str` 构建对应的二叉树，其过程是用 `i` 从头扫描 `str`；采用先序方法，当 `i` 超界时返回 `NULL`；否则当遇到 ‘#’ 字符时返回 `NULL`，当遇到其它字符时，创建一个结点，可以采用递归的方法构造该二叉树；也可以采用非递归方法构造该二叉树。

- 1) 采用二叉链式存储创建二叉树 B1；
- 2) 采用先序序列化显示输出序列，并存储到文件中；
- 3) 从文件中读出序列，并反序列化的递归方法构造二叉树 B2；
- 4) 从文件中读出序列，并反序列化的非递归方法构造二叉树 B3；
- 5) 使用非递归方法输出二叉树中序遍历序列；
- 6) 使用非递归方法输出二叉树后序遍历序列；
- 7) 销毁释放二叉树 B1，B2，B3。

## 七、实验器材（设备、元器件）：

PC 机一台，装有 C 语言集成开发环境。

## 八、数据结构与程序：

### 数据结构设计

```
#define MAXSIZE 30
#define ElemType char

typedef struct Node
{
    ElemType data;
    struct Node* lchild;
    struct Node* rchild;
} BTreeNode;

typedef struct
{
    BTreeNode* data[MAXSIZE];
    int top;
} SqStack;
```

### 程序设计

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <fstream>

#define MAXSIZE 30
#define FILE_PLACE "BTNode.dat"
#define ElemType char

typedef struct Node
{
    ElemType data;
    struct Node* lchild;
    struct Node* rchild;
} BTNode;

typedef struct
{
    BTNode* data[MAXSIZE];
    int top;
} SqStack;

void CreateBTree(BTNode*& b, char* str, int& length_b);
void DestoryBTree(BTNode*& b);
int Translate(BTNode* b, ElemType St[MAXSIZE], int i);
void SaveToFile(char str[MAXSIZE]);
void ReadFromFile(char str[MAXSIZE]);
int RTransform(BTNode*& b, char St[MAXSIZE], int i);
void Transform(BTNode*& b, char St[MAXSIZE]);
void PreOrder(BTNode* b);
void InOrder(BTNode* b);
void PostOrder(BTNode* b);
void InitStack(SqStack*& s);
```

```

bool Push(SqStack*& s, BTNode* b);
bool Pop(SqStack*& s, BTNode*& b);
bool GetTop(SqStack* s, BTNode*& b);
bool StackEmpty(SqStack* s);
void DestorySt(SqStack*& s);
void DispBTree(BTNode* b);

int main(void)
{
    int i = 0, choice, length_b1 = 0, length_b2 = 0, length_b3 = 0;
    ElemType str[MAXSIZE], St[MAXSIZE], BT[MAXSIZE], ch;
    BTNode* b1, * b2, * b3;
    b1 = NULL;

    while (1)
    {
        printf("\n\n2019/6 实验 2 PART.1\n");

        printf("=====
=====\\n");
        printf("1.二叉链式存储创建二叉树 B1\\n");
        printf("2.先序序列化显示输出二叉树 B1 序列，并存储到文件中\\n");
        printf("3.从文件中读出序列，并反序列化的递归方法构造二叉树
B2\\n");
        printf("4.从文件中读出序列，并反序列化的非递归方法构造二叉树
B3\\n");
        printf("5.使用非递归方法输出二叉树中序遍历序列\\n");
        printf("6.使用非递归方法输出二叉树后序遍历序列\\n");
        printf("7.销毁释放二叉树 B1,B2,B3\\n");
        printf("0.退出实验程序\\n");

        printf("=====
=====\\n");

```

```
printf(">>>请选择功能: ");
scanf("%d", &choice);
fflush(stdin); //清空缓存区
printf("\n");

switch (choice)
{
    case 1:
        if (b1 != NULL)
        {
            i = 0;
            DestoryBTree(b1);
        }

        printf("请输入括号表示法的二叉树: ");
        while ((ch = getchar()) != '\n')
        {
            str[i] = ch;
            i++;
        }
        str[i] = '\0';

        CreateBTree(b1, str, length_b1);
        printf("成功创建二叉树 B1: ");
        DispBTree(b1);
        break;

    case 2:
        if (length_b1 == 0)
        {
            printf("未建立二叉树 B1! ");
            break;
        }
}
```

```
        Translate(b1, St, 0);

        printf("(先序遍历)链式存储结构为: %s\n", St);
        SaveToFile(St);
        break;

    case 3:
        ReadFromFile(St);
        printf("已读取到内容: %s\n", St);

        //初始化 b2
        b2 = (BTNode*)malloc(sizeof(BTNode));
        b2->lchild = b2->rchild = NULL;

        RTransform(b2, St, 0);
        printf("已建立二叉树 B2: ");
        DispBTree(b2);
        break;

    case 4:
        ReadFromFile(St);
        printf("已读取到内容: %s\n", St);

        //初始化 b3
        b3 = (BTNode*)malloc(sizeof(BTNode));
        b3->lchild = b3->rchild = NULL;

        Transform(b3, St);
        printf("已建立二叉树 B3: ");
        DispBTree(b3);
        break;

    case 5:
        if (length_b1 == 0)
```



```
{
    printf("未建立二叉树 B1! ");
    break;
}

printf("中序遍历二叉树: ");
InOrder(b1);
break;

case 6:
    if (length_b1 == 0)
    {
        printf("未建立二叉树 B1! ");
        break;
    }

    printf("后序遍历二叉树: ");
    PostOrder(b1);
    break;

case 7:
    if (b1 != NULL)
        DestoryBTree(b1);
    if (b2 != NULL)
        DestoryBTree(b2);
    if (b3 != NULL)
        DestoryBTree(b3);

    printf("二叉树 B1 | B2 | B3 已成功销毁! \n");
    break;

case 0:
    printf("成功退出! \n");
    return 0;
```

```
        default:
            printf("请输入正确数字! \n");
            break;
    }
}

//建立二叉树
void CreateBTree(BTNode*& b, ElemType* str, int& length_b)
{
    BTNode *St[MAXSIZE], *p;
    int top = -1, k, j = 0;
    ElemType ch;

    length_b = 0;
    b = NULL;
    ch = str[j];

    while(ch != '\0')
    {
        switch (ch)
        {
            case '(': top++; St[top] = p; k = 1; break;
            case ')': top--; break;
            case ',': k = 2; break;
            default:
                length_b++;
                p = (BTNode*)malloc(sizeof(BTNode));
                p->data = ch;
                p->lchild = p->rchild = NULL;

                if (b == NULL)
                {
```

```
        b = p;
    }

    else
    {
        switch (k)
        {
            case 1: St[top]->lchild = p; break;
            case 2: St[top]->rchild = p; break;
        }
    }
}
j++;
ch = str[j];
}
}
```

//先序链式转化二叉树

```
int Translate(BTNode* b, ElemType St[MAXSIZE], int i)
```

```
{
    if (b == NULL)
    {
        St[i] = '#';
        i++;
        return i;
    }

    St[i] = b->data;
    i++;
    i = Translate(b->lchild, St, i);
    i = Translate(b->rchild, St, i);
    St[i] = '\0';
    return i;
}
```

```
//将链式二叉树转化为二叉树
int RTransform(BTNode*& b, char St[MAXSIZE], int i) //递归算法
{
    if (St[i] == '#' || St[i] == '\0')
    {
        b = NULL;
        i++;
        return i;
    }

    BTNode* p;
    p = (BTNode*)malloc(sizeof(BTNode));

    p->data = St[i];
    b = p;
    i++;
    i = RTransform(b->lchild, St, i);
    i = RTransform(b->rchild, St, i);
    return i;
}

void Transform(BTNode*&b, char a[MAXSIZE]) //非递归算法
{
    BTNode *St[MAXSIZE], *p;
    int top = -1;
    int k=0,j=0,tem=0;
    char ch;
    b = NULL;
    ch = a[j];

    while (ch!=0)
    {
        if (ch!='#')

```

```
{
    top++;
    St[top] = p;
    p = (BTNode*)malloc(sizeof(BTNode));
    p->data = ch;
    p->rchild = NULL;
    p->lchild = NULL;
    if(b==NULL)
    {
        b = p;
    }

    else
    {
        if(k==0)
        {
            St[top]->lchild = p;
        }
        else
        {
            if(k==1)
            {
                St[top]->rchild = p;
                k=0;
            }

            if(k==2)
            {
                St[tem]->rchild = p;
                k=0;
            }
        }
    }
}
```

```
        if(ch=='#')
        {
            if(k==0)
            {
                p->lchild = NULL;
                k=1;
            }
            else
            {
                p->rchild = NULL;
                tem = top;
                while(St[tem]->rchild!=NULL) tem--;
                k=2;
            }
        }
        j++;
        ch=a[j];
    }
}

//保存字符串到文件中
void SaveToFile(char str[MAXSIZE])
{
    FILE* fp;

    if ((fp = fopen(FILE_PLACE, "wb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }

    fwrite(str, strlen(str), 1, fp);
}
```

```
fclose(fp);
printf("保存成功！ \n");
}

//从文件中读取字符串
void ReadFromFile(char str[MAXSIZE])
{
    FILE* fp;
    int len;

    if ((fp = fopen(FILE_PLACE, "rb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }

    //获取文件长度
    fseek(fp, 0, SEEK_END);
    len = ftell(fp);
    rewind(fp);

    //读取文件字符串内容
    fread(str, sizeof(char), len, fp);
    str[len] = '\0';

    fclose(fp);
}

//输出顺序存储结构的二叉树
void DispBTree(BTNode* b)
{
    if (b != NULL)
    {
        printf("%c", b->data);
    }
}
```

```
        if (b->lchild != NULL || b->rchild != NULL)
        {
            printf("(");
            DispBTree(b->lchild);
            if (b->rchild != NULL)
                printf(",");
            DispBTree(b->rchild);
            printf(")");
        }
    }
}
```

//先序遍历二叉树

```
void PreOrder(BTNode* b)
{
    if (b != NULL)
    {
        printf("%c", b->data);
        PreOrder(b->lchild);
        PreOrder(b->rchild);
    }
}
```

//中序遍历二叉树

```
void InOrder(BTNode* b) //非递归
{
    BTNode* p;
    SqStack* st;
    InitStack(st);
    p = b;

    while (!StackEmpty(st) || p != NULL)
    {
        while (p != NULL)
```



```
    {
        Push(st, p);
        p = p->lchild;
    }
    if (!StackEmpty(st))
    {
        Pop(st, p);
        printf("%c", p->data);
        p = p->rchild;
    }
}
printf("\n");
DestorySt(st);
}
```

//后序遍历二叉树

void PostOrder(BTNode\* b) //非递归

```
{
    BTNode* p, * r;
    bool flag;
    SqStack* st;
    InitStack(st);

    p = b;
    do
    {
        while (p != NULL)
        {
            Push(st, p);
            p = p->lchild;
        }

        r = NULL;
        flag = true;
```

```
while (!StackEmpty(st) && flag)
{
    GetTop(st, p);

    if (p->rchild == r)
    {
        printf("%c", p->data);
        Pop(st, p);
        r = p;
    }

    else
    {
        p = p->rchild;
        flag = false;
    }
}
} while (!StackEmpty(st));
printf("\n");
DestorySt(st);
}
```

//栈的操作

```
void InitStack(SqStack*& s)
{
    s = (SqStack*)malloc(sizeof(SqStack));
    s->top = -1;
}
```

```
bool Push(SqStack*& s, BTreeNode* b)
{
    if (s->top == MAXSIZE-1)
        return false;
```

```
s->top++;
s->data[s->top] = b;
return true;
}

bool Pop(SqStack*& s, BTreeNode*& b)
{
    if (s->top == -1)
        return false;
    b = s->data[s->top];
    s->top--;
    return true;
}

bool GetTop(SqStack* s, BTreeNode*& b)
{
    if (s->top == -1)
        return false;
    b = s->data[s->top];
    return true;
}

bool StackEmpty(SqStack* s)
{
    if (s->top == -1)
        return true;
    return false;
}

void DestorySt(SqStack*& s)
{
    free(s);
}
```

```
//销毁二叉树
void DestoryBTree(BTNode*& b)
{
    if (b != NULL)
    {
        DestoryBTree(b->lchild);
        DestoryBTree(b->rchild);
        free(b);
    }
}
```

## 九、程序运行结果：

- 1) 采用二叉链式存储创建二叉树 B1。输入括号表示法的二叉树序列，读取处理生成二叉树 B1，如图 1.9.1 所示。

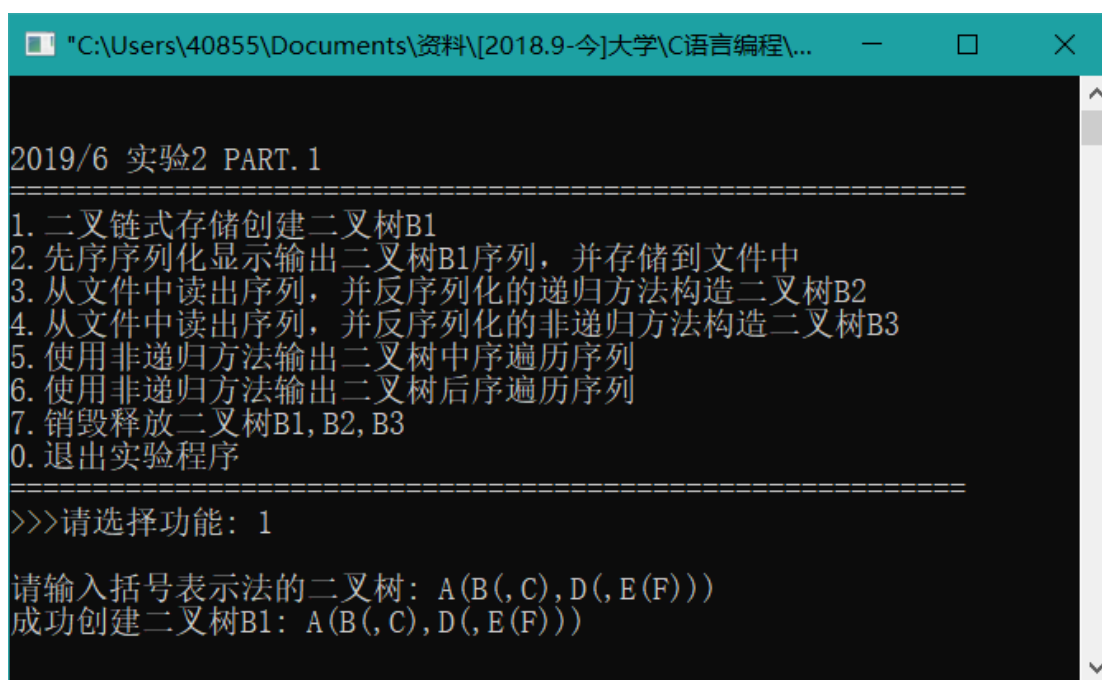


图 1.9.1

- 2) 采用先序序列化显示输出序列，并存储到文件中。先序遍历二叉树 B1 并生成保存其顺序存储结构的字符串，打印出来，如图 1.9.2.1 所示；并保存到文件中，如图 1.9.2.2 所示。

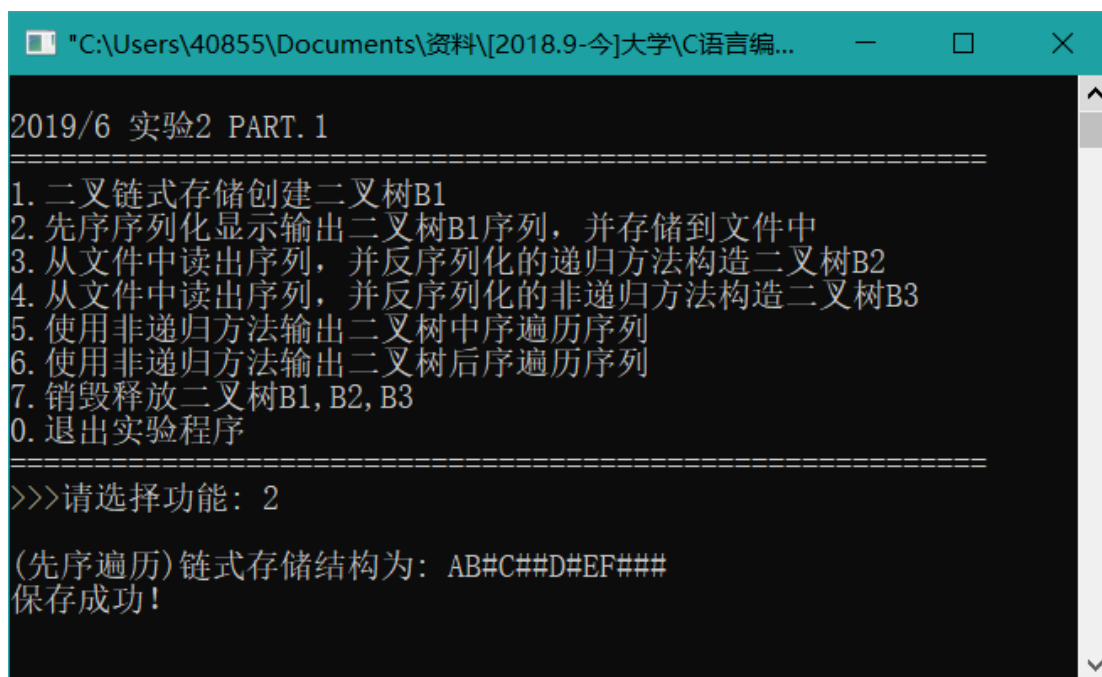


图 1.9.2.1

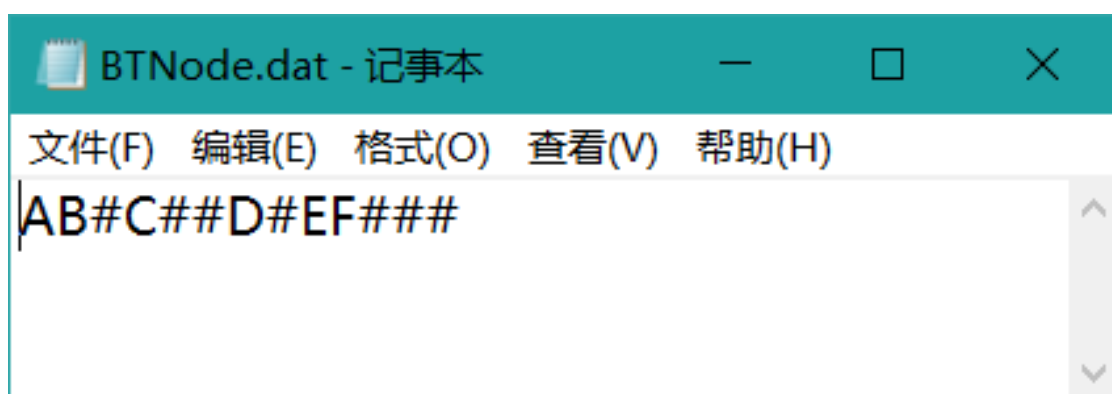
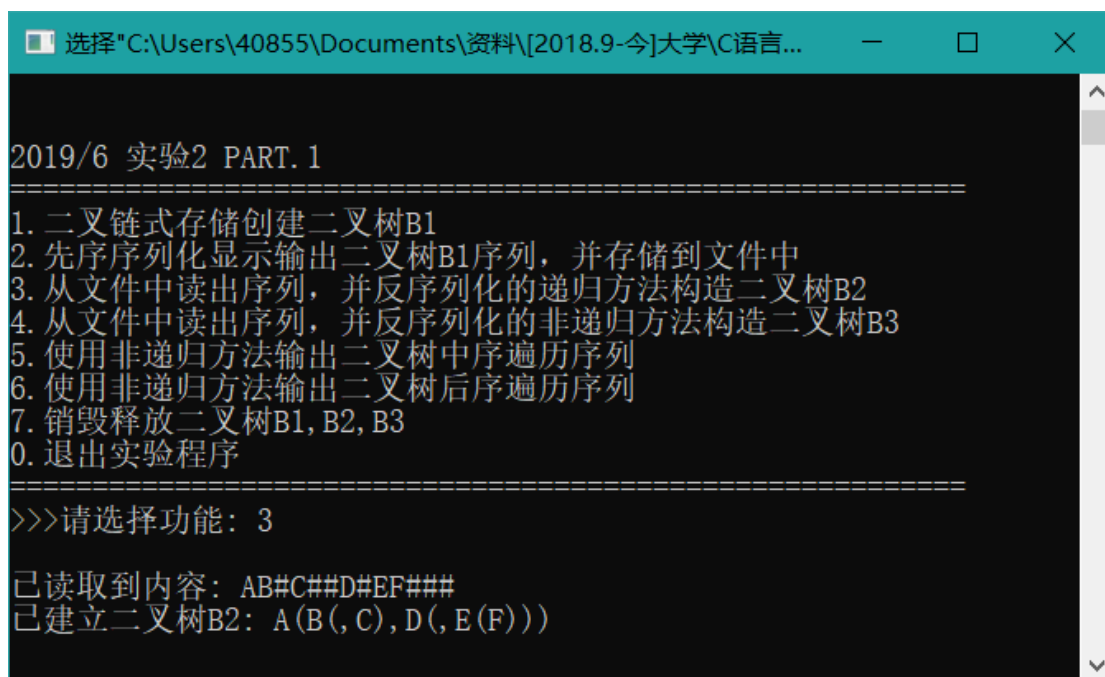


图 1.9.2.2

- 3) 从文件中读出序列，并反序列化的递归方法构造二叉树 B2。将从文件中读取到的二叉树顺序存储法的序列内容打印出来，并用反序列化的递归方法构造二叉树，最后先序遍历二叉树，用括号表示法打印出二叉树结构的内容。如图 1.9.3 所示。

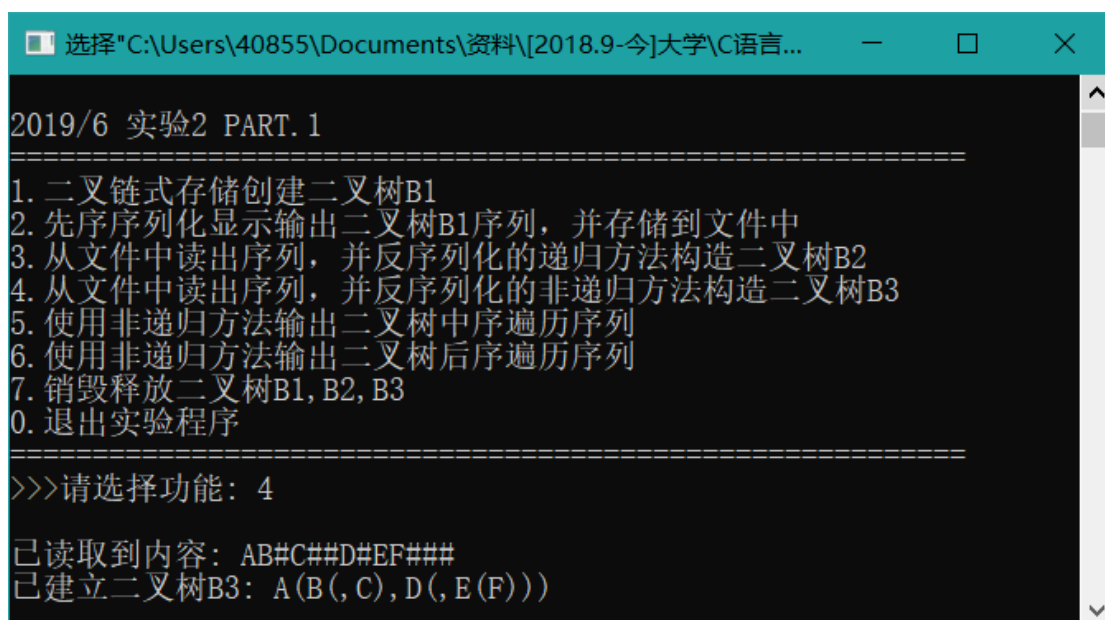


```
选择"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言..."
2019/6 实验2 PART. 1
=====
1. 二叉链式存储创建二叉树B1
2. 先序序列化显示输出二叉树B1序列，并存储到文件中
3. 从文件中读出序列，并反序列化的递归方法构造二叉树B2
4. 从文件中读出序列，并反序列化的非递归方法构造二叉树B3
5. 使用非递归方法输出二叉树中序遍历序列
6. 使用非递归方法输出二叉树后序遍历序列
7. 销毁释放二叉树B1, B2, B3
8. 退出实验程序
=====
>>>请选择功能： 3

已读取到内容：AB#C##D#EF###
已建立二叉树B2：A(B(, C), D(, E(F)))
```

图 1.9.3

- 4) 从文件中读出序列，并反序列化的非递归方法构造二叉树 B3。将从文件中读取到的二叉树顺序存储法的序列内容打印出来，并用反序列化的非递归方法构造二叉树，最后先序遍历二叉树，用括号表示法打印出二叉树结构的内容。如图 1.9.4 所示。



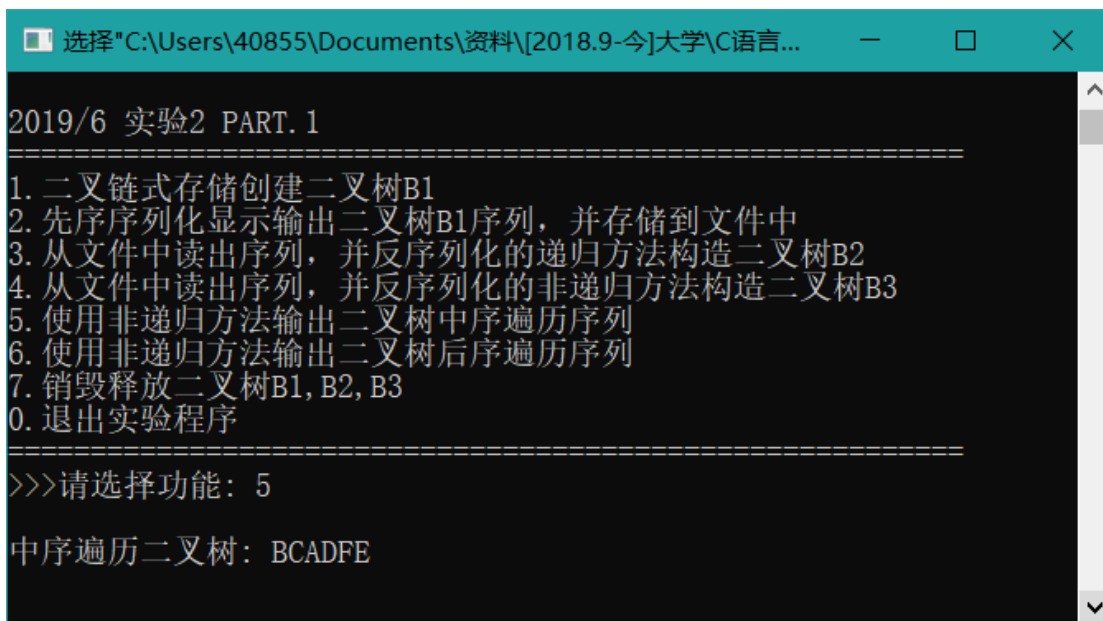
```
选择"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言..."
2019/6 实验2 PART. 1
=====
1. 二叉链式存储创建二叉树B1
2. 先序序列化显示输出二叉树B1序列，并存储到文件中
3. 从文件中读出序列，并反序列化的递归方法构造二叉树B2
4. 从文件中读出序列，并反序列化的非递归方法构造二叉树B3
5. 使用非递归方法输出二叉树中序遍历序列
6. 使用非递归方法输出二叉树后序遍历序列
7. 销毁释放二叉树B1, B2, B3
8. 退出实验程序
=====
>>>请选择功能： 4

已读取到内容：AB#C##D#EF###
已建立二叉树B3：A(B(, C), D(, E(F)))
```

图 1.9.4

- 5) 使用非递归方法输出二叉树中序遍历序列。用非递归的中序遍历算法遍

历二叉树 B1，并将访问的结点依次打印出来。如图 1.9.5 所示。

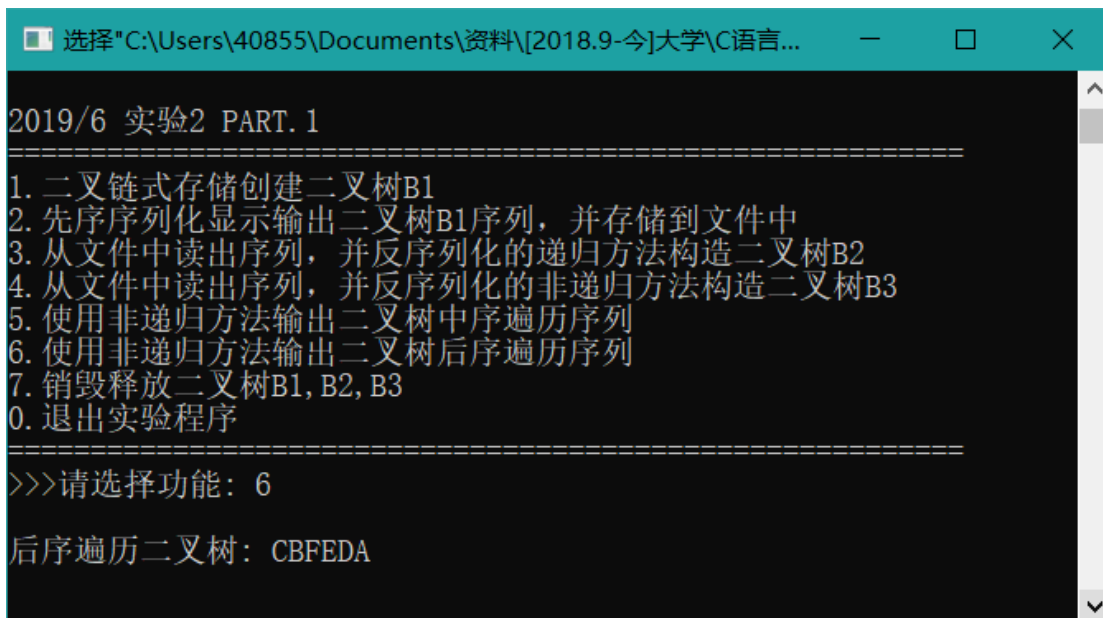


```
选择"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言..."
2019/6 实验2 PART. 1
=====
1. 二叉链式存储创建二叉树B1
2. 先序序列化显示输出二叉树B1序列，并存储到文件中
3. 从文件中读出序列，并反序列化的递归方法构造二叉树B2
4. 从文件中读出序列，并反序列化的非递归方法构造二叉树B3
5. 使用非递归方法输出二叉树中序遍历序列
6. 使用非递归方法输出二叉树后序遍历序列
7. 销毁释放二叉树B1, B2, B3
0. 退出实验程序
=====
>>>请选择功能： 5

中序遍历二叉树：BCADFE
```

图 1.9.5

- 6) 递归方法输出二叉树后序遍历序列。用非递归的后序遍历算法遍历二叉树 B1，并将访问的结点依次打印出来。如图 1.9.6 所示。

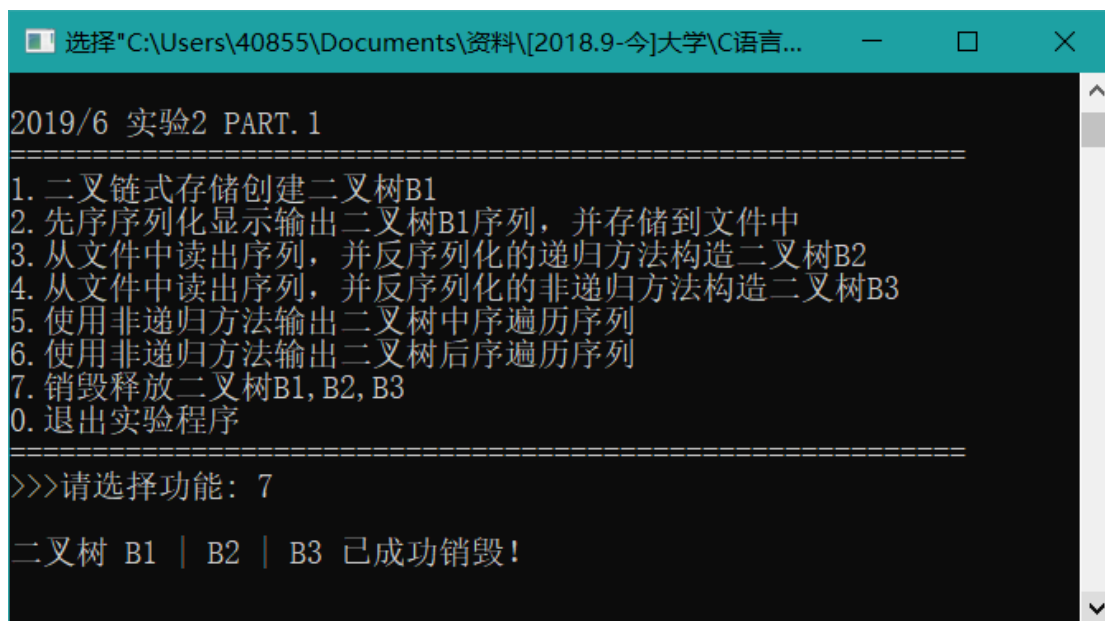


```
选择"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言..."
2019/6 实验2 PART. 1
=====
1. 二叉链式存储创建二叉树B1
2. 先序序列化显示输出二叉树B1序列，并存储到文件中
3. 从文件中读出序列，并反序列化的递归方法构造二叉树B2
4. 从文件中读出序列，并反序列化的非递归方法构造二叉树B3
5. 使用非递归方法输出二叉树中序遍历序列
6. 使用非递归方法输出二叉树后序遍历序列
7. 销毁释放二叉树B1, B2, B3
0. 退出实验程序
=====
>>>请选择功能： 6

后序遍历二叉树：CBFEDA
```

图 1.9.6

- 7) 销毁释放二叉树 B1，B2，B3。如果存在二叉树 B1 或 B2 或 B3，则销毁它们占用的内存。如图 1.9.7 所示。



```
选择"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言...
2019/6 实验2 PART. 1
=====
1. 二叉链式存储创建二叉树B1
2. 先序序列化显示输出二叉树B1序列，并存储到文件中
3. 从文件中读出序列，并反序列化的递归方法构造二叉树B2
4. 从文件中读出序列，并反序列化的非递归方法构造二叉树B3
5. 使用非递归方法输出二叉树中序遍历序列
6. 使用非递归方法输出二叉树后序遍历序列
7. 销毁释放二叉树B1, B2, B3
0. 退出实验程序
=====
>>>请选择功能： 7

二叉树 B1 | B2 | B3 已成功销毁！
```

图 1.9.7

## 十、实验结论：

读取 dat（二进制）文件时，要与读取 txt（文本）文件加以区分；需要充分理解文件指针的含义与作用，充分利用 fread()和 fwrite()函数完成程序设计。对于文件的长度，可以利用 fseek()和 ftell()函数获取，方便后续的读写操作。

利用二叉树顺序结构存储的特点，可以将任意二叉树补成完全二叉树，将结点编号并存储到数组数据结构之中。并用特殊符号表示空节点。

利用二叉树链式存储结构的特点，可相对于一般的二叉树节省存储空间，并可以轻松地访问某结点的孩子结点。

利用栈的性质，在本实验的程序设计中，可以存储二叉树的双亲结点，便于扫描字符串进行判断，完成入栈、出栈操作，以生成二叉树。

利用递归算法，可以通过先序遍历、中序遍历或后序遍历极为轻松地访问并读取二叉链存储的数据。

相较于递归函数，非递归函数在处理二叉链时较为繁琐。但利用好栈的性质，仍可以完成遍历函数的构造。

利用先序遍历的递归算法，可以简便快速地访问并释放内存中二叉树各节点所占用的空间，以达到销毁二叉树的目的。



## 第2部分：公园景点间的最短路径查询程序的设计

### 四、实验原理：

#### i. 图的定义

无论多么复杂的图，都是由顶点和边构成的。因此，采用形式化的定义，图（graph） $G$  由两个集合  $V$ （vertex）和  $E$ （edge）构成，记为  $G=(V, E)$ 。其中：

- （1） $V$  是顶点的有限集合，记为  $V(G)$ ；
- （2） $E$  是连接  $V$  中两个不同顶点的边的有限集合，记为  $E(G)$ 。

对于图  $G$ ，如果表示边的顶点对是有序的，则称作有向图，表示边的顶点对用尖括号括起来，如  $\langle i, j \rangle$ ；若是无序的，则称作无向图，表示边的顶点对用圆括号括起来，如  $(i, j)$ 。

#### ii. 图的基本术语

- （1）端点和邻接点：

对于无向图中的一条边  $(i, j)$ ，其顶点  $i$  和顶点  $j$  被称为该边的两端点，它们互为邻接点；对于有向图的一条有向边  $\langle i, j \rangle$ ，顶点  $i$  被称为该边的起始端点， $j$  被称为该边的终止端点，则  $j$  是  $i$  的出边邻接点， $i$  是  $j$  的入边邻接点。

- （2）顶点的度、入度和出度：

无向图中，一个顶点所关联的边的数目称为该顶点的度；有向图中，以一个顶点为终点的边的数目被称为入度，以它为起点的边的数目被称为出度，入度和出度之和为该顶点的度。

- （3）完全图：

无向图中的任意两个顶点之间都存在一条边，有向图中的任意两个顶点之间都存在着方向相反的两条边，则把图成为完全图。

- （4）子图：

设有两个图  $G=\langle V, E \rangle$  和  $G'=\langle V', E' \rangle$ ，若  $V'$  是  $V$  的子集， $E'$  是  $E$  的子集，则称  $G'$  是  $G$  的子集。

- （5）回路和环：

若一条路径（概念见 iii.）上的开始点和结束点为同一个顶点，则称此路径为回路或环。其中，开始点和结束点相同的简单路径（概念见 iii.）被称为简单回路或简单环。

## (6) 连通、连通图和连通分量：

无向图中，若两顶点之间有路径，则称两顶点连通。若任意两顶点之间都有路径，则称该无向图为连通图，否则为非连通图。其中，极大连通子图被称为连通分量。

## (7) 强连通图和强连通分量：

有向图中，若两顶点之间有路径，则称两顶点连通。若任意两顶点之间都有路径，则称该有向图为强连通图，否则为非强连通图。其中，极大强连通子图被称为强连通分量。

## (8) 权和网：

图中的每一条边都可以附有一个对应的数值，这样的与边相关的数值被称为权。边上有权的图被称为带权图或网。

## iii. 路径和最短路径的概念

对于不带权图图  $G=(V, E)$ ，从顶点  $i$  到顶点  $j$  的一条**路径**是一个顶点序列  $(i, i', i'', \dots, n, j)$ 。**路径长度**是指一条路径上经过的边的数目，在数值上等于该路径上的顶点数减 1。其中，路径长度最小的路径被称为**最短距离**。

对于带权图，将一条路径上所经边的权之和定义为该路径的路径长度。其中，路径长度最小的路径被称为**最短路径**。事实上，可以将不带权图上的每条边看成权值为 1 的边，那么带权图和不带权图的最短距离和最短距离的定义就一样了。

若一条路径上除开始点和结束点可以相同之外，其它顶点各不相同，则称该路径为**简单路径**。

## iv. 最短路径算法

Dijkstra 算法是目前应用最广泛的计算给定两个顶点之间的最短路径算法。代码实现如下：

```
void Dijkstra(MatGraph* M, int v, int p)
{
    int dist[MAXV], path[MAXV];
    int S[MAXV];
    int MINdis, i, j, u;
    for (i=0; i<M->n; i++)
    {
```

```

dist[i] = M->edges[v][i];
S[i] = 0;
if (M->edges[v][i] < INF)
    path[i] = v;
else
    path[i] = -1;
}
S[v] = 1;
path[v] = 0;
for (i=0; i<M->n-1; i++)
{
    MINdis = INF;
    for (j=0; j<M->n; j++)
    {
        if (S[j] == 0 && dist[j]<MINdis)
        {
            u = j;
            MINdis = dist[j];
        }
    }
    S[u] = 1;
    for (j=0; j<M->n; j++)
    {
        if (S[j] == 0)
        {
            if (M->edges[u][j] < INF && dist[u]+M->edges[u][j] <
dist[j])
            {
                dist[j] = dist[u]+M->edges[u][j];
                path[j] = u;
            }
        }
    }
}
}

```

```
    Dispath(M, dist, path, S, v, p);
}

void Dispath(MatGraph* M, int dist[], int path[], int S[], int v, int p)
{
    int j, k;
    int apath[MAXV], d;

    if (S[p] == 1 && p != v)
    {
        printf("从 %d 号景点到 %d 号景点的最短路径长度为: %d \t 路  
径为: ", v, p, dist[p]);
        d = 0;
        apath[d] = p;
        k = path[p];
        if (k == -1)
        {
            printf("无路径\n");
        }
        else
        {
            while (k != v)
            {
                d++;
                apath[d] = k;
                k = path[k];
            }
            d++;
            apath[d] = v;
            printf("%d", apath[d]);
            for (j=d-1; j>=0; j--)
            {
                printf(",%d", apath[j]);
            }
        }
    }
}
```

```

        printf("\n");
    }
}
}

```

而 Floyd 算法可以通过以每个顶点作为源点循环求出每对顶点之间的最短路径。代码实现如下：

```

void Floyd(MatGraph* M, AdjGraph*& Adj)
{
    int A[MAXV][MAXV], path[MAXV][MAXV];
    int i, j, k;
    for (i=0; i<M->n; i++)
    {
        for (j=0; j<M->n; j++)
        {
            A[i][j] = M->edges[i][j];
            if (i != j && M->edges[i][j] < INF)
            {
                path[i][j] = i;
            }
            else
            {
                path[i][j] = -1;
            }
        }
    }
    for (k=0; k<M->n; k++)
    {
        for (i=0; i<M->n; i++)
        {
            for (j=0; j<M->n; j++)
            {
                if (A[i][j] > A[i][k] + A[k][j])
                {
                    A[i][j] = A[i][k] + A[k][j];

```

```

        path[i][j] = path[k][j];
    }
}
}
}
}
Dispath_F(M, A, path, Adj);
}

void Dispath_F(MatGraph* M, int A[][MAXV], int path[][MAXV],
AdjGraph*& Adj)
{
    int i, j, k, s, count;
    int apath[MAXV], d;
    ArcNode* p;
    Adj->n = M->n;
    Adj->e = M->e;

    for (i=0; i<M->n; i++)
    {
        for (j=0; j<M->n; j++)
        {
            if (A[i][j] != INF && i != j)
            {
                printf("从 %d 号景点到 %d 号景点的路径为: ", i,
j);

                p = (ArcNode*)malloc(sizeof(ArcNode));
                p->adjvex = j; //保存结点序号信息

                k = path[i][j];
                d = 0;
                apath[d] = j;
                while (k!=-1 && k!=i)
                {

```

```

        d++;
        apath[d] = k;
        k = path[i][k];
    }
    d++;
    apath[d] = i;
    printf("%d", apath[d]);
    for (s=d-1; s>=0; s--)
    {
        printf(",%d", apath[s]);
    }

    for (count=0; count<=d; count++)
    {
        s = d - count;
        p->list[count] = apath[s]; //保存最短路径结点序
列

    }
    p->length = A[i][j]; //保存最短路径的长度
    p->nextarc = Adj->adjlist[i].firstarc; //p 的下一个结点
指向对应头结点的后一个结点
    Adj->adjlist[i].firstarc = p; //对应头结点的后一个结
点指向 p

    Adj->adjlist[i].num++;

    printf("\t 路径长度为: %d\n", A[i][j]);
}
}
}
}

```

## 五、实验目的：

让学生通过实验，理解和掌握图在计算机内存中的存储表示方法；掌握图在磁盘文件中的存储方法；掌握计算任意两点间的最短路径的算法；

让学生掌握设计较为复杂软件的基本方法，初步具备解决复杂软件工程的基本知识和技能。

### 六、实验内容：

某公园内有  $n$  个连通的旅游景点，游客需要查询任意景点间最短路径。请设计并编程实现如下功能：

- 1) 设计数据结构与界面，输入直接相邻的两个旅游景点的名字以及它们之间的距离；并将每对直接相连的景点间的距离存到磁盘文件中。
- 2) 设计算法，实现计算给定的两个旅游景点间的最短路径；
- 3) 对公园的所有旅游景点，设计算法实现计算所有的景点对之间的最短路径，并将最短路径上的各旅游景点及每段路径长度写入磁盘文件 AllPath.dat 中。
- 4) 编写程序从文件 AllPath.dat 中读出所有旅游景点间的最短路径信息，到内存链表中管理；请运用所学的数据结构知识，设计内存链表的数据结构，实现用户输入任意两个旅游景点，能快速地从内存链表查询出两景点间的最短路径。

### 七、实验器材（设备、元器件）：

PC 机一台，装有 C 语言集成开发环境。

### 八、数据结构与程序：

#### 数据结构设计

```
//邻接矩阵
typedef struct
{
    int no;
    char name[MAXSIZE];
} SpotINFO;

typedef struct
{
    int edges[MAXV][MAXV];
    int n, e;
    SpotINFO vexs[MAXV];
} MatGraph;
```



```

//内存链表|邻接表
typedef struct ANode
{
    int adjvex;
    struct ANode* nextarc;
    int list[MAXV];
    int length;
} ArcNode;

typedef struct Vnode
{
    int no;
    int num;
    ArcNode* firstarc;
} VNode;

typedef struct
{
    VNode adjlist[MAXV];
    int n;
    int e;
} AdjGraph;

```

程序设计

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

```

#include <stdlib.h>
#include <string.h>
#define MAXSIZE 15
#define MAXV 10
#define INF 32767
#define FILE_PLACE_PARK "Park.dat"
#define FILE_PLACE_ALLP "AllPath.dat"

//邻接矩阵

```

```
typedef struct
{
    int no;
    char name[MAXSIZE];
} SpotINFO;

typedef struct
{
    int edges[MAXV][MAXV];
    int n, e;
    SpotINFO vexs[MAXV];
} MatGraph;

//内存链表|邻接表
typedef struct ANode
{
    int adjvex;
    struct ANode* nextarc;
    int list[MAXV];
    int length;
} ArcNode;

typedef struct Vnode
{
    int no;
    int num;
    ArcNode* firstarc;
} VNode;

typedef struct
{
    VNode adjlist[MAXV];
    int n;
    int e;
```

```

} AdjGraph;

void InitMatGraph(MatGraph*& M);
void SaveToMG(MatGraph*& M, char a[MAXSIZE], char b[MAXSIZE], int d);
bool SpotExistence(MatGraph* M, char s[MAXSIZE], int& x);
void SaveToFile_P(MatGraph* M);
void ReadFromFile_P(MatGraph*& M);
void PrintMG(MatGraph* M);
void PrintSpot(MatGraph* M);
void PrintPath(MatGraph* M);
void Dijkstra(MatGraph* M, int v, int p);
void Dispath(MatGraph* M, int dist[], int path[], int S[], int v, int p);
void Floyd(MatGraph* M, AdjGraph*& Adj);
void Dispath_F(MatGraph* M, int A[][MAXV], int path[][MAXV],
AdjGraph*& Adj);
void DispAdj(AdjGraph* Adj, int a, int b);
void InitAdj(AdjGraph*& Adj);
void SaveToFile_A(AdjGraph* Adj);
void ReadFromFile_A(AdjGraph*& Adj);

int main(void)
{
    int choice, place_length, a, b;
    char place_a[MAXSIZE], place_b[MAXSIZE];
    bool flag;
    MatGraph* M1, * M2;
    AdjGraph* A, * A1;

    while(1)
    {
        printf("\n\n2019/6 实验 2 PART.2\n");

        printf("=====
=====\\n");

```

```

printf("1.创建并存储景点对距离信息\n");
printf("2.查询景点对的最短路径信息\n");
printf("3.打印并保存所有景点对之间的最短路径信息\n");
printf("4.快速查询所有景点对之间的最短路径信息\n");
printf("0.退出实验程序\n");

printf("=====
=====\\n");

printf(">>>请选择功能: ");

scanf("%d", &choice);
fflush(stdin); //清空缓存区
printf("\\n");

switch (choice)
{
    case 1:
        flag = true;
        InitMatGraph(M1);

        while (flag) {
            printf("\\n=====创建并存储景点对距离
信息=====\\n");
            printf("1.创建景点对距离信息\\n");
            printf("2.存储景点对距离信息\\n");
            printf("3.读取并打印已有景点对距离信息\\n");
            printf("0.保存并返回到主界面\\n");

            printf("=====
=====\\n");

            printf(">>>请选择功能: ");
            scanf("%d", &choice);
            fflush(stdin);
            switch (choice) {

```

```
case 1:
    printf("请输入起始景点的名称: ");
    scanf("%s", place_a);
    printf("请输入目标景点的名称: ");
    scanf("%s", place_b);
    printf("请输入景点之间的距离: ");
    scanf("%d", &place_length);
    SaveToMG(M1, place_a, place_b, place_length);
    printf("成功输入数据! (选择 2 将已输入的数据
存储到文件中)\n");

    break;
case 2:
    SaveToFile_P(M1);
    printf("成功将数据保存到文件中! \n");
    break;
case 3:
    M2 = (MatGraph*)malloc(sizeof(MatGraph));
    ReadFromFile_P(M2);
    printf("已有景点如下:\n");
    PrintSpot(M2);
    printf("\n 景点距离信息如下:\n");
    PrintPath(M2);
    free(M2);
    break;
case 0:
    SaveToFile_P(M1);
    flag = false;
    printf("成功返回! \n");
    break;
default:
    printf("请输入正确序号! \n");
    break;
}
}
```

```
        break;
    case 2:
        //交互
        M2 = (MatGraph*)malloc(sizeof(MatGraph));
        ReadFromFile_P(M2);
        printf("已有景点如下:\n");
        PrintSpot(M2);
        printf("\n");
        printf("请输入起始景点的序号: ");
        scanf("%d", &a);
        printf("请输入目标景点的序号: ");
        scanf("%d", &b);
        printf("\n");

        //调用算法
        Dijkstra(M2, a, b);
        free(M2);
        break;
    case 3:
        InitAdj(A);
        M2 = (MatGraph*)malloc(sizeof(MatGraph));
        ReadFromFile_P(M2);
        printf("当前景点间最短距离如下:\n");
        Floyd(M2, A);
        free(M2);
        SaveToFile_A(A);
        break;
    case 4:
        M2 = (MatGraph*)malloc(sizeof(MatGraph));
        ReadFromFile_P(M2);
        printf("已有景点如下:\n");
        PrintSpot(M2);

        A1 = (AdjGraph*)malloc(sizeof(AdjGraph));
```

```
        ReadFromFile_A(A1);
        printf("请输入起始景点的序号: ");
        scanf("%d", &a);
        printf("请输入目标景点的序号: ");
        scanf("%d", &b);
        DispAdj(A1, a, b);
        break;
    case 0:
        free(M1);
        printf("成功退出! \n");
        return 0;
    default:
        printf("请输入正确序号! \n");
        break;
    }
}

//初始化邻接矩阵
void InitMatGraph(MatGraph*& M)
{
    int i, j;
    M = (MatGraph*)malloc(sizeof(MatGraph));
    for (i = 0; i<MAXV; i++)
    {
        for (j = 0; j<MAXV; j++)
        {
            if (i == j)
                M->edges[i][j] = 0;
            else
                M->edges[i][j] = INF;
        }
    }
    M->n = 0;
```

```
M->e = 0;
}

//添加两旅游景点距离信息并储存
void SaveToMG(MatGraph*& M, char a[MAXSIZE], char b[MAXSIZE], int d)
{
    int i, j, loc_a, loc_b;
    if (!SpotExistence(M, a, loc_a))
    {
        strcpy(M->vexs[M->n].name, a); //顶点名称
        M->vexs[M->n].no = M->n; //顶点编号
        loc_a = M->n; //顶点位置
        M->n++; //邻接矩阵顶点数增加
    }

    if (!SpotExistence(M, b, loc_b))
    {
        strcpy(M->vexs[M->n].name, b);
        M->vexs[M->n].no = M->n;
        loc_b = M->n;
        M->n++;
    }

    if (M->edges[loc_a][loc_b] == INF)
        M->e++;
    M->edges[loc_a][loc_b] = d;
}

bool SpotExistence(MatGraph* M, char s[MAXSIZE], int& x)
{
    int i;
    if (M->n == 0) return false; //没有存储景点信息，返回不存在

    for (i=0; i<=M->n; i++)
```



```
{
    if (strcmp(s, M->vexs[i].name) == 0)
    {
        x = i;
        return true; //有该景点名字信息，返回存在
    }
}
return false; //无该景点名字信息，返回不存在
}

//保存最短路径到文件中
void SaveToFile_A(AdjGraph* Adj)
{
    FILE* fp;
    ArcNode* p;

    if ((fp = fopen(FILE_PLACE_ALLP, "wb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }

    fwrite(Adj, sizeof(AdjGraph), 1, fp);
    for (int i=0; i<Adj->n; i++)
    {
        p = Adj->adjlist[i].firstarc;
        while (p != NULL)
        {
            fwrite(p, sizeof(ArcNode), 1, fp);
            p = p->nextarc;
        }
    }

    fclose(fp);
}
```

```
}

//从文件中读取最短路径信息
void ReadFromFile_A(AdjGraph*& Adj)
{
    FILE* fp;
    ArcNode* p;
    int j;

    if ((fp = fopen(FILE_PLACE_ALLP, "rb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }

    fread(Adj, sizeof(AdjGraph), 1, fp);
    for (int i=0; i<Adj->n; i++)
    {
        for (j=Adj->adjlist[i].num; j>0; j--)
        {
            p = (ArcNode*)malloc(sizeof(ArcNode));
            fread(p, sizeof(ArcNode), 1, fp);
            p->nextarc = Adj->adjlist[i].firstarc;
            Adj->adjlist[i].firstarc = p;
        }
    }

    fclose(fp);
}

//保存邻接矩阵到文件中
void SaveToFile_P(MatGraph* M)
{
    FILE* fp;
```

```
if ((fp = fopen(FILE_PLACE_PARK, "wb")) == NULL)
{
    fprintf(stderr, "can't open\n");
    exit(EXIT_FAILURE);
}

fwrite(M, sizeof(MatGraph), 1, fp);

fclose(fp);
}

//从文件中读取邻接矩阵
void ReadFromFile_P(MatGraph*& M)
{
    FILE* fp;

    if ((fp = fopen(FILE_PLACE_PARK, "rb")) == NULL)
    {
        fprintf(stderr, "can't open\n");
        exit(EXIT_FAILURE);
    }

    fread(M, sizeof(MatGraph), 1, fp);

    fclose(fp);
}

//打印邻接矩阵
void PrintMG(MatGraph* M)
{
    int i, j;

    if (M->n == 0)
```

```
{
    printf("邻接矩阵为空！ \n");
    return;
}

for (i=0; i<M->n; i++)
{
    for (j=0; j<M->n; j++)
    {
        printf("%d ", M->edges[i][j]);
    }
    printf("\n");
}

//打印所有景点信息
void PrintSpot(MatGraph* M)
{
    int i;

    if (M->n == 0)
    {
        printf("景点信息为空！ \n");
        return;
    }

    for (i=0; i<M->n; i++)
    {
        printf("%d: %s\n", M->vexs[i].no, M->vexs[i].name);
    }
}

//打印景点距离信息
void PrintPath(MatGraph* M)
```

```

{
    int i, j;

    if (M->n == 0)
    {
        printf("景点信息为空! \n");
        return;
    }

    for (i=0; i<MAXV; i++)
    {
        for (j=0; j<MAXV; j++)
        {
            if (M->edges[i][j] != 0 && M->edges[i][j] != INF)
                printf("%s 到 %s 的距离为: %d\n", M->vexs[i].name,
M->vexs[j].name, M->edges[i][j]);
        }
    }
}

//Dijkstra 算法
void Dijkstra(MatGraph* M, int v, int p)
{
    int dist[MAXV], path[MAXV];
    int S[MAXV];
    int MINdis, i, j, u;
    for (i=0; i<M->n; i++)
    {
        dist[i] = M->edges[v][i];
        S[i] = 0;
        if (M->edges[v][i] < INF)
            path[i] = v;
        else
            path[i] = -1;
    }
}

```

```

    }
    S[v] = 1;
    path[v] = 0;
    for (i=0; i<M->n-1; i++)
    {
        MINdis = INF;
        for (j=0; j<M->n; j++)
        {
            if (S[j] == 0 && dist[j]<MINdis)
            {
                u = j;
                MINdis = dist[j];
            }
        }
        S[u] = 1;
        for (j=0; j<M->n; j++)
        {
            if (S[j] == 0)
            {
                if (M->edges[u][j] < INF && dist[u]+M->edges[u][j] < dist[j])
                {
                    dist[j] = dist[u]+M->edges[u][j];
                    path[j] = u;
                }
            }
        }
        Dispath(M, dist, path, S, v, p);
    }

void Dispath(MatGraph* M, int dist[], int path[], int S[], int v, int p)
{
    int j, k;
    int apath[MAXV], d;

```

```
if (S[p] == 1 && p != v)
{
    printf("从 %d 号景点到 %d 号景点的最短路径长度为: %d \t 路径  
为: ", v, p, dist[p]);
    d = 0;
    apath[d] = p;
    k = path[p];
    if (k == -1)
    {
        printf("无路径\n");
    }
    else
    {
        while (k != v)
        {
            d++;
            apath[d] = k;
            k = path[k];
        }
        d++;
        apath[d] = v;
        printf("%d", apath[d]);
        for (j=d-1; j>=0; j--)
        {
            printf(",%d", apath[j]);
        }
        printf("\n");
    }
}

//Floyd 算法
void Floyd(MatGraph* M, AdjGraph*& Adj)
```

```
{
    int A[MAXV][MAXV], path[MAXV][MAXV];
    int i, j, k;
    for (i=0; i<M->n; i++)
    {
        for (j=0; j<M->n; j++)
        {
            A[i][j] = M->edges[i][j];
            if (i != j && M->edges[i][j] < INF)
            {
                path[i][j] = i;
            }
            else
            {
                path[i][j] = -1;
            }
        }
    }
    for (k=0; k<M->n; k++)
    {
        for (i=0; i<M->n; i++)
        {
            for (j=0; j<M->n; j++)
            {
                if (A[i][j] > A[i][k] + A[k][j])
                {
                    A[i][j] = A[i][k] + A[k][j];
                    path[i][j] = path[k][j];
                }
            }
        }
    }
    Dispath_F(M, A, path, Adj);
}
```



```
void Dispath_F(MatGraph* M, int A[][MAXV], int path[][MAXV],
AdjGraph*& Adj)
{
    int i, j, k, s, count;
    int apath[MAXV], d;
    ArcNode* p;
    Adj->n = M->n;
    Adj->e = M->e;

    for (i=0; i<M->n; i++)
    {
        for (j=0; j<M->n; j++)
        {
            if (A[i][j] != INF && i != j)
            {
                printf("从 %d 号景点到 %d 号景点的路径为: ", i, j);

                p = (ArcNode*)malloc(sizeof(ArcNode));
                p->adjvex = j; //保存结点序号信息

                k = path[i][j];
                d = 0;
                apath[d] = j;
                while (k!=-1 && k!=i)
                {
                    d++;
                    apath[d] = k;
                    k = path[i][k];
                }
                d++;
                apath[d] = i;
                printf("%d", apath[d]);
                for (s=d-1; s>=0; s--)
```

```

        {
            printf(",%d", apath[s]);
        }

        for (count=0; count<=d; count++)
        {
            s = d - count;
            p->list[count] = apath[s]; //保存最短路径结点序列
        }
        p->length = A[i][j]; //保存最短路径的长度
        p->nextarc = Adj->adjlist[i].firstarc; //p 的下一个结点指向对
应头结点的后一个结点
        Adj->adjlist[i].firstarc = p; //对应头结点的后一个结点指向 p
        Adj->adjlist[i].num++;

        printf("\t 路径长度为: %d\n", A[i][j]);
    }
}

void InitAdj(AdjGraph*& Adj)
{
    int i;
    Adj = (AdjGraph*)malloc(sizeof(AdjGraph));
    for(i=0; i<MAXV; i++)
    {
        Adj->adjlist[i].no = i;
        Adj->adjlist[i].num = 0;
        Adj->adjlist[i].firstarc = NULL;
    }
    Adj->n = 0;
    Adj->e = 0;
}

```

```
void DispAdj(AdjGraph* Adj, int a, int b)
{
    int i;
    ArcNode* p;

    if (a == b)
    {
        printf("不存在自身到自身的最短路径序列! \n");
        return;
    }

    p = Adj->adjlist[a].firstarc;
    while (p != NULL)
    {
        if (p->adjvex == b)
            break;
        else
            p = p->nextarc;
    }

    printf("从 %d 号景点到 %d 号景点的最短路径序列为: ", a, b);
    for (i=0; ; i++)
    {
        if (p->list[i] >= 0 && p->list[i] < MAXV)
            printf("%d ", p->list[i]);
        else
            break;
    }
    printf("\t 路径长度为: %d\n", p->length);
}
```

## 九、程序运行结果：

- 1) 设计数据结构与界面，输入直接相邻的两个旅游景点的名字以及它们之

间的距离,如图 2.9.1 所示;并将每对直接相连的景点间的距离存到磁盘文件中,如图 2.9.2 所示。

在本演示中,对应的序号、有向边和距离分别为:

- (0) a->b = 1
- (1) a->c = 2
- (2) a->d = 3
- (3) b->c = 2
- (4) b->d = 1
- (5) c->d = 1

```

"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言编程\2...
2019/6 实验2 PART. 2
=====
1. 创建并存储景点对距离信息
2. 查询景点对的最短路径信息
3. 打印并保存所有景点对之间的最短路径信息
4. 快速查询所有景点对之间的最短路径信息
0. 退出实验程序
=====
>>>请选择功能: 1

=====创建并存储景点对距离信息=====
1. 创建景点对距离信息
2. 存储景点对距离信息
3. 读取并打印已有景点对距离信息
0. 保存并返回到主界面
=====
>>>请选择功能: 1
请输入起始景点的名称: a
请输入目标景点的名称: b
请输入景点之间的距离: 1
成功输入数据! (选择 2 将已输入的数据存储到文件中)

```

图 2.9.1

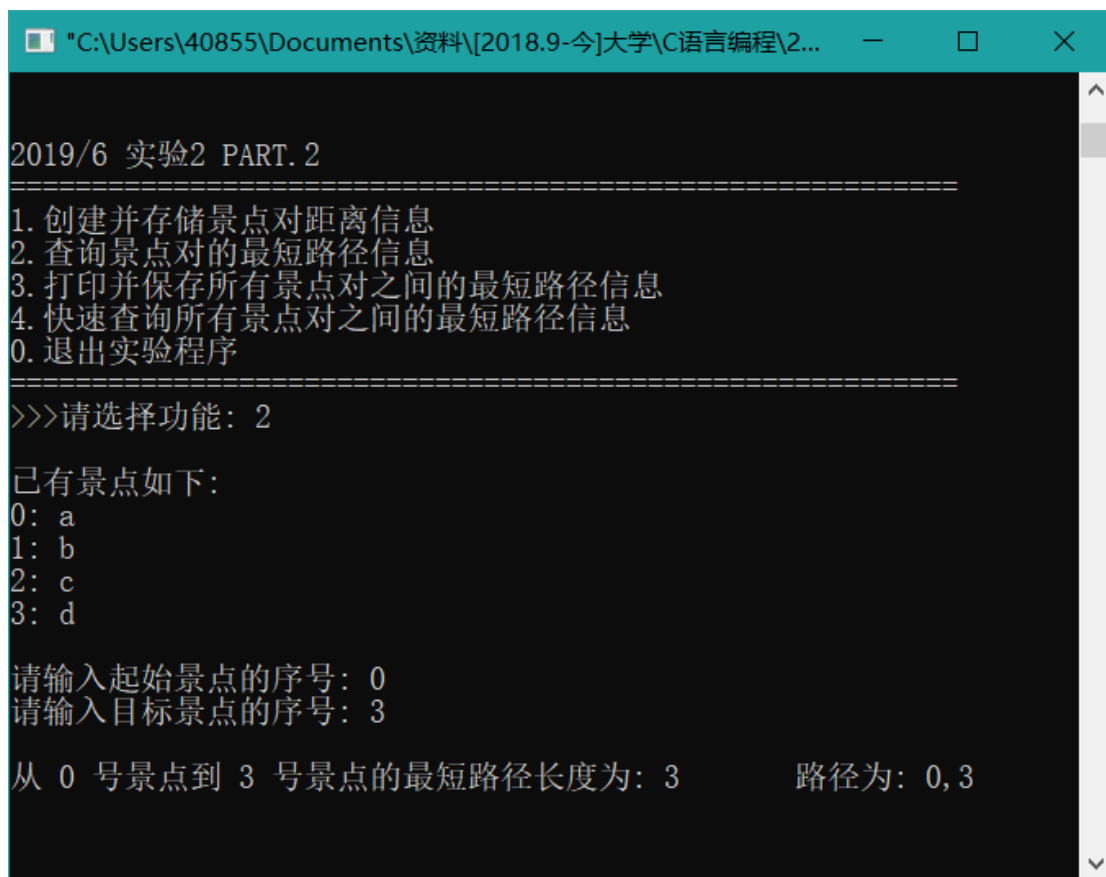
```

=====创建并存储景点对距离信息=====
1. 创建景点对距离信息
2. 存储景点对距离信息
3. 读取并打印已有景点对距离信息
0. 保存并返回到主界面
=====
>>>请选择功能: 2
成功将数据保存到文件中!

```

图 2.9.2

- 2) 设计算法,实现计算给定的两个旅游景点间的最短路径,如图 2.9.3 所示。



```
"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言编程\2...  _  □  ×

2019/6 实验2 PART. 2
=====
1. 创建并存储景点对距离信息
2. 查询景点对的最短路径信息
3. 打印并保存所有景点对之间的最短路径信息
4. 快速查询所有景点对之间的最短路径信息
0. 退出实验程序
=====
>>>请选择功能：2

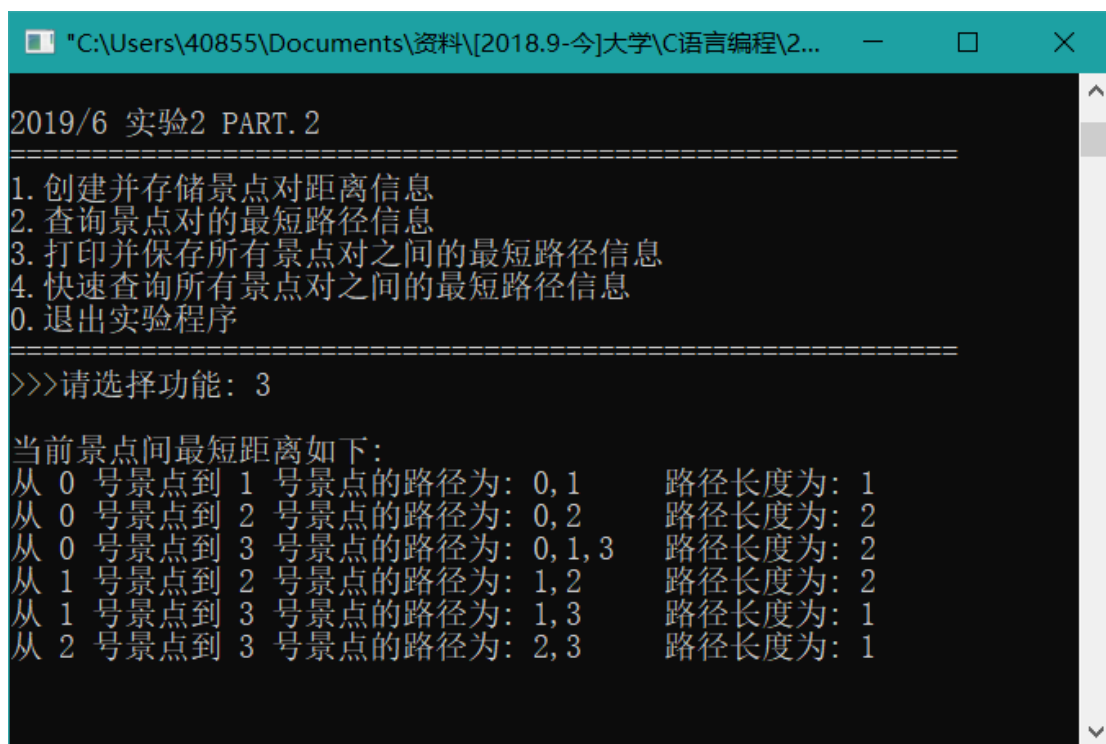
已有景点如下：
0: a
1: b
2: c
3: d

请输入起始景点的序号：0
请输入目标景点的序号：3

从 0 号景点到 3 号景点的最短路径长度为：3      路径为：0, 3
```

图 2.9.3

- 3) 对公园的所有旅游景点,设计算法实现计算所有的景点对之间的最短路径,并将最短路径上的各旅游景点及每段路径长度写入磁盘文件 AllPath.dat 中,如图 2.9.4 所示。



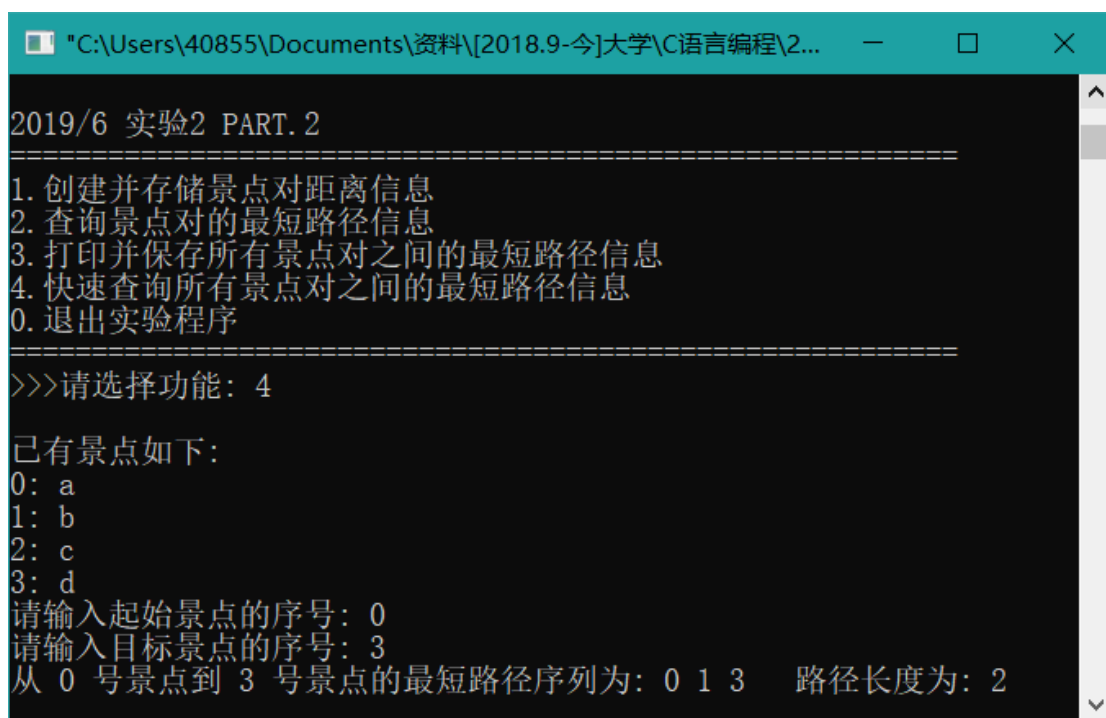
```
"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言编程\2...  —  □  ×

2019/6 实验2 PART. 2
=====
1. 创建并存储景点对距离信息
2. 查询景点对的最短路径信息
3. 打印并保存所有景点对之间的最短路径信息
4. 快速查询所有景点对之间的最短路径信息
0. 退出实验程序
=====
>>>请选择功能：3

当前景点间最短距离如下：
从 0 号景点到 1 号景点的路径为：0,1      路径长度为：1
从 0 号景点到 2 号景点的路径为：0,2      路径长度为：2
从 0 号景点到 3 号景点的路径为：0,1,3    路径长度为：2
从 1 号景点到 2 号景点的路径为：1,2      路径长度为：2
从 1 号景点到 3 号景点的路径为：1,3      路径长度为：1
从 2 号景点到 3 号景点的路径为：2,3      路径长度为：1
```

图 2.9.4

- 4) 编写程序从文件 AllPath.dat 中读出所有旅游景点间的最短路径信息，到内存链表中管理；采用邻接表的数据结构，实现用户输入任意两个旅游景点，快速地从内存链表查询出两景点间的最短的路径和距离，如图 2.9.5 所示。



```
"C:\Users\40855\Documents\资料\[2018.9-今]大学\C语言编程\2...  —  □  ×

2019/6 实验2 PART. 2
=====
1. 创建并存储景点对距离信息
2. 查询景点对的最短路径信息
3. 打印并保存所有景点对之间的最短路径信息
4. 快速查询所有景点对之间的最短路径信息
0. 退出实验程序
=====
>>>请选择功能：4

已有景点如下：
0: a
1: b
2: c
3: d
请输入起始景点的序号：0
请输入目标景点的序号：3
从 0 号景点到 3 号景点的最短路径序列为：0 1 3    路径长度为：2
```

图 2.9.5