

电子科技大学信息与软件工程学院

实 验 报 告

学 号 2018091618008

姓 名 袁昊男

(实验) 课程名称 程序设计与算法基础 II

理论教师 曹晟

实验教师 文淑华

电子科技大学

实验报告

学生姓名：袁昊男 学号：2018091618008 指导教师：曹晟

实验地点：基础实验大楼 537 实验时间：2019. 06. 22

一、 实验名称：非线性结构及相关算法的设计与应用

二、 实验学时：8 学时

三、 实验目的：

1. 掌握二叉树的先序遍历、中序遍历、后序遍历及其递归与非递归算法；
2. 掌握从先序序列、中序序列、后序序列递归与非递归建立二叉树的算法；
3. 掌握二叉树的链式存储结构；
4. 掌握二叉树的序列化与反序列化；
5. 掌握带权无向/有向图的存储结构和基本运算方法；
6. 掌握图在磁盘文件中的存储方法；
7. 掌握应用 Dijkstra 或 Floyd 算法计算结点之间最短路径的方法；
8. 掌握 C 语言文件读写的一般方法；
9. 掌握 C 语言循环菜单界面的创建方法；

四、 实验原理：

1. **结构体**：结构是可能具有不同类型的值(成员)的集合，结构的元素(在 C 语言中的说法是结构的成员)可能具有不同的类型。而且每个结构成员都有名字，所以在选择特定的结构成员时需要指明结构成员的名字而不是它的位置。
2. **结构变量的声明**：当需要存储相关数据项的集合时，结构是一种合乎逻辑的选择。例如：假设需要记录存储在仓库中的零件，用来存储每种零件的信息可能包括零件的编号(整数)、零件的名称(字符串)以及现有零件的数量。为了产生一个可以存储全部三种数据项的变量，可以使用类似下面这样的声明：结构 part1 和 part2 中的成员 number 和成员 name 不会与结构 employee1 和 employee2 中的成员 number 和成员 name 冲突。

```
1. struct{
```

```
2. int number;
3. char name[NAME_LEN+1];
4. int on_hand;
5. } part1, part2;
```

每个结构变量都有三个成员：`number`(零件的编号)、`name`(零件的名称)和`on_hand`(现有数量)。注意这里的声明格式和 C 语言中其他变量的声明格式一样，`struct{...}`指明了类型，而 `part1` 和 `part2` 则是具有这种类型的变量。结构的成员在内存中是按照声明的顺序存储的。

3. **结构标记：**结构标记是用于标识某种特定结构的名称。下面的例子声明了名为 `part` 的结构标记：

```
1. struct part{
2. int number;
3. char name[NAME_LEN+1];
4. int on_hand;
5. };
```

一旦创建了标记 `part`，就可以用它来声明变量了：

```
1. struct part part1, part2;
```

但是，不能通过漏掉单词 `struct` 来缩写这个声明，因为 `part` 不是类型名，如果没有单词 `struct` 的话，它没有任何意义。

4. **结构类型的定义：**除了声明结构标记，还可以用 `typedef` 来定义真实的类型名。例如，可以按照如下方式定义名为 `Part` 的类型：

```
1. typedef struct {
2. int number;
3. char name[NAME_LEN+1];
4. int on_hand;
5. } Part;
```

注意，类型 `Part` 的名字必须出现在定义的末尾，而不是在单词 `struct` 后。可以像内置类型那样使用 `Part`。

5. **二叉树的链式存储结构：**二叉树的链式存储结构是指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。结点类型 `BTNode` 的声明如下：

```
1. typedef struct node
2. { ElemType data;
3. struct node *lchild;
4. struct node *rchild;
5. } BTNode;
```

其中，`data` 表示值域，用于存储对应的数据元素，`lchild` 和 `rchild` 分别表示

左指针域和右指针域,分别用于存储左孩子结点和右孩子结点的存储地址。这种链表存储结构通常称为二叉链。二叉链中通过根结点 **root** 来唯一标识整个存储结构。

6. **二叉树的遍历:** 二叉树的遍历是指按照一定的次序访问二叉树中的所有结点,并且每个结点仅被访问一次的过程。它是二叉树最基本的运算,是二叉树所有其他运算实现的基础。一棵二叉树由 3 个部分(即根结点、左子树和右子树)构成,可以从任何部分开始遍历,所以有 3!种遍历方法。若规定子树的遍历总是先左后右(先右后左与之对称),则对于非空二叉树,可得到以下 3 种递归的遍历方法,即先序遍历、中序遍历和后序遍历,对应分别有递归和非递归方法。
7. **二叉树的序列化与反序列化:** 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,存在于内存中,不能持久化。如果需要将一棵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,其与一般的先序遍历不一样,需要记录空结点并用字符“#”表示,并且假设序列中空结点的值均为“#”。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。
8. **图的存储结构:** 图的存储结构除了要存储图中各个顶点本身的信息以外,同时还要存储顶点与顶点之间的所有关系(边的信息)。图的邻接矩阵是一种采用邻接矩阵数组表示顶点之间相邻关系的存储结构。设 $G=(V,E)$ 是含有 n 个顶点的图,各顶点的编号为 $0\sim(n-1)$,以 G 为带权无向图为例, G 的邻接矩阵数组 **A** 是 n 阶方阵,其定义如下:

$$A[i][j]=\begin{cases} w_{ij} & i \neq j \text{ 且 } (i,j) \in E(G), \text{ 该边的权为 } w_{ij} \\ 0 & i = j \\ \infty & \text{其他} \end{cases},$$

图的完整邻接矩阵类型的声明如下:

```
1. #define MAXV <最大顶点个数>
2. #define INF 32767
3. typedef struct
4. {
5.     int no;
6.     InfoType info;
```

```

7. } VertexType; //顶点的类型
8. typedef struct
9. {
10.     int edges[MAXV][MAXV];
11.     int n, e;
12.     VertexType vexs[MAXV];
13. } MatGraph; //完整的图邻接矩阵类型

```

除此之外，图还可以用邻接表来存储。图的邻接表是一种顺序与链式存储相结合的存储方法。对于含有 n 个顶点的图，每个顶点建立一个单链表，第 $i(0 \leq i \leq n-1)$ 个单链表中的结点表示关联于顶点 i 的边，也就是将顶点 i 的所有邻接点链接起来；每个单链表再附设一个头结点，并将头结点构成一个头结点数组 `adjlist`，图的完整邻接表存储类型的声明如下：

```

1. typedef struct ANode
2. {
3.     int adjvex;
4.     struct ANode *nextarc;
5.     int weight;
6. } ArcNode; //边结点的类型
7. typedef struct VNode
8. {
9.     InfoType info;
10.    ArcNode *firstarc;
11. } VNode; //邻接表的头结点类型
12. typedef struct
13. {
14.    VNode adjlist[MAXV];
15.    int n, e;
16. } AdjGraph; //完整的图邻接表类型类型

```

9. **Dijkstra 算法求顶点之间的最短路径：**基本思想是，设 $G=(V,E)$ 是一个带权有向图，把图中的顶点集合 V 分成两组，第 1 组为已求出最短路径的顶点集合(用 S 表示，初始时 S 中只有一个源点，以后每求得一条最短路径 v, \dots, u ，就将顶点 u 加入到集合 S 中，直到全部顶点都加入到 S 中，算法即结束)，第 2 组为其余未确定最短路径的顶点集合(用 U 表示)，按最短路径长度的递增次序依次把第 2 组的顶点加入 S 中。
10. **Floyd 算法求顶点之间的最短路径：**基本思想是，设 $G=(V,E)$ 是一个带权有向图，另外设置一个二维数组 A 用于存放当前顶点之间的最短路径长度，即分量 $A[i][j]$ 表示当前 $i \Rightarrow j$ 的最短路径长度，通过递推产生一个矩阵序列 $A_0, A_1, \dots, A_k, A_{n-1}$ ，其中 $A_k[i][j]$ 表示 $i \Rightarrow j$ 的路径上所经过的顶点编号不大于 k 的最短路径长度。
11. **系统选择界面：**可以通过一个 `while` 条件为 1 的循环来完成程序功能的循

环调用，直至输入退出系统的命令。例如：

```
1. while(1)
2. {
3.     switch(choice) {
4.         case 1: Fucntion1;
5.         break; ...
6.     } }
```

12. 文件操作示例：

```
1. char tempstring[30];
2. FILE *fp;           //定义文件操作的指针
3. //fopen 用于打开文件，接收两个参数，一个是文件的路径，另一个是文件
   //打开的方式。例如 xxxxxx.txt 和该项目的可执行文件在同一目录下，
   //则此处只需要所读取内容的文件名；r 代表以只读方式打开文件
4.     fp = fopen("xxxxxxx.txt", "r");
5.     //如果以 w 方式代表打开只写文件，若文件存在则长度清为 0，即该
   //文件内容消失；若不存在则创建该文件，其余打开方式请自行查阅文档
6.     fp = fopen("xxxxxxx.txt", "w");
7.     //fscanf 用于从 txt 文件中读取内容，下例以字符串形式读入一段
   //字符并存放在 tempstring 中
8.     fscanf(fp, "%s", tempstring);
9.     //或者以格式化的方式读入字符串
10.    fscanf(fp, "\t%s\n", tempstring);
11.    //fprintf 以格式化的方式向 txt 文件中写入内容
12.    fprintf(fp, "%s\t", tempstring);
13.    //检查文件内容是否已经读到文件结束符了
14.    while (!feof(fp)){.....}
15.    //最后需要使用 fclose 关闭文件指针
16. fclose(fp);
```

五、 实验内容：

第 1 部分：二叉树的连式存储、序列化和反序列化 (4 学时)

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，存在于内存中，不能持久化。如果需要将一棵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。如下图所示的序列化存储为：A, B, #, C, #, #, D, #, E, F, #, #, #。所谓序列化是对二叉树进行先序遍历产生一个字符序列，其与一般的先序遍历不一样，需要记录空结点并用字符“#”表示，并且假设序列中空结点的值均为“#”。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。

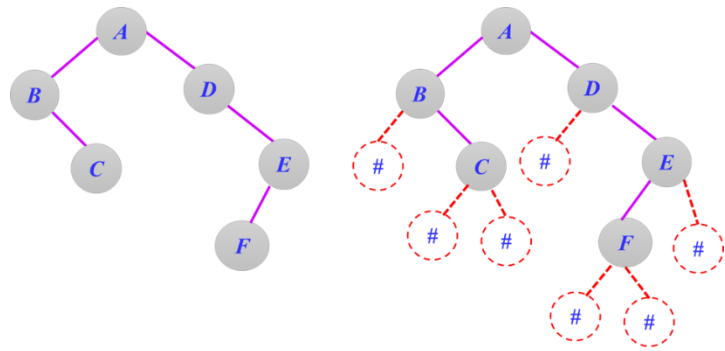


图 5.1.1 二叉树图例

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；

```

C:\袁昊男\学习\大一下\数据结构与算法\
*****
*          二叉树的链式存储、序列化和反序列化          *
*****

请输入二叉树结点信息，以#表示空结点：
A B # C # # D # E F # # #

二叉树B1创建成功！

先序序列化：
A, B, #, C, #, #, D, #, E, F, #, #, #
序列已存储至preorder.dat文件中！

从文件中读取序列成功！

递归反序列化创建树B2成功！
非递归反序列化创建树B3成功！

非递归中序遍历树B2序列：
#, B, #, C, #, A, #, D, #, F, #, E, #
非递归后序遍历树B3序列：
#, #, #, C, B, #, #, #, F, #, E, D, A

是否销毁二叉树B1、B2、B3? (y/n): y
销毁完毕！
请按任意键继续. . .

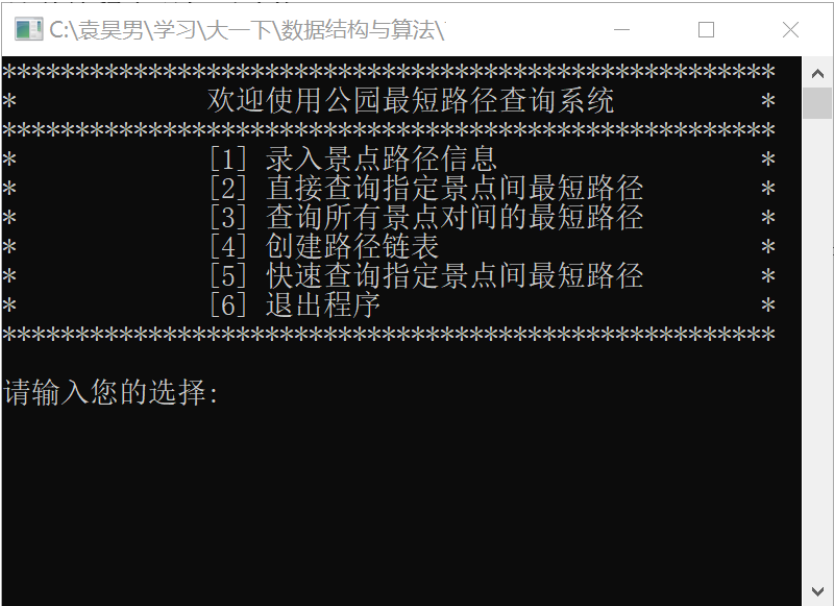
```

图 5.1.1 程序运行结果

第 2 部分：公园景点间的最短路径查询程序的设计 (4 学时)

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. 设计程序运行的功能菜单：



```
*****
*                  欢迎使用公园最短路径查询系统                  *
*****
*                  [1] 录入景点路径信息                          *
*                  [2] 直接查询指定景点间最短路径                *
*                  [3] 查询所有景点对间的最短路径                *
*                  [4] 创建路径链表                              *
*                  [5] 快速查询指定景点间最短路径                *
*                  [6] 退出程序                                  *
*****
请输入您的选择:
```

图 5.2.1 功能菜单界面

2. 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；

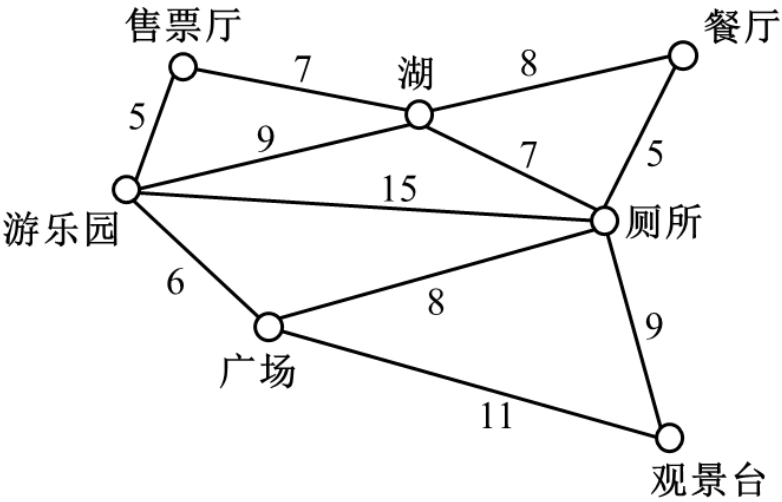


图 5.2.2 公园景点地图

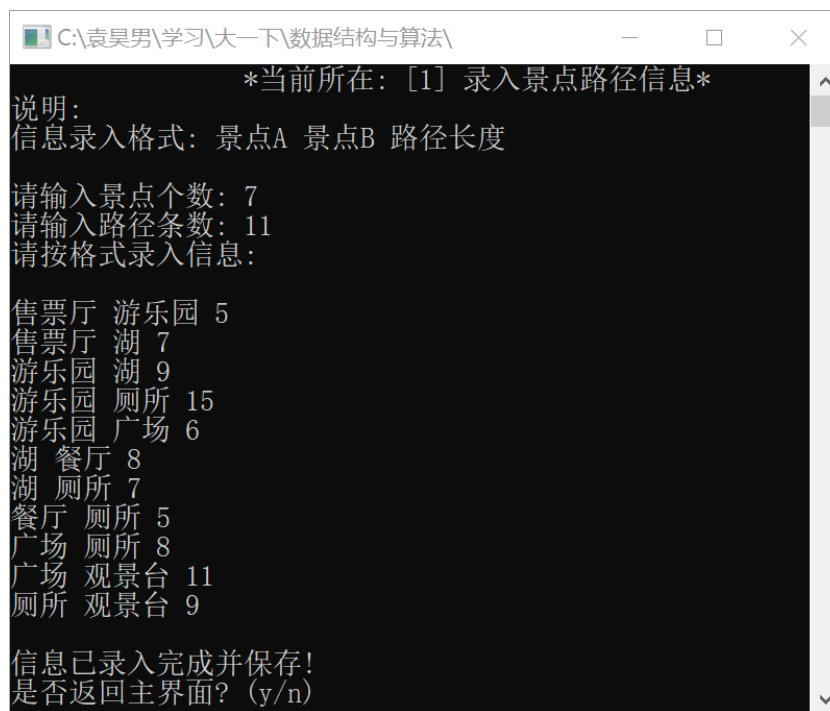


图 5.2.3 录入景点路径信息



图 5.2.4 路径信息保存至 info.dat 文件中

3. 设计算法，实现计算给定的两个旅游景点间的最短路径；

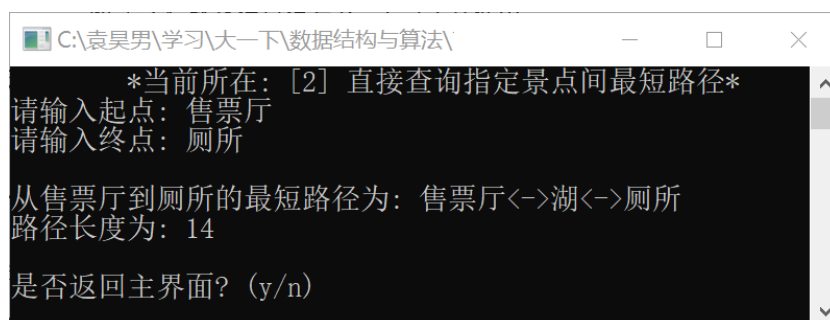


图 5.2.5 应用 Floyd 算法计算景点间最短路径

4. 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；

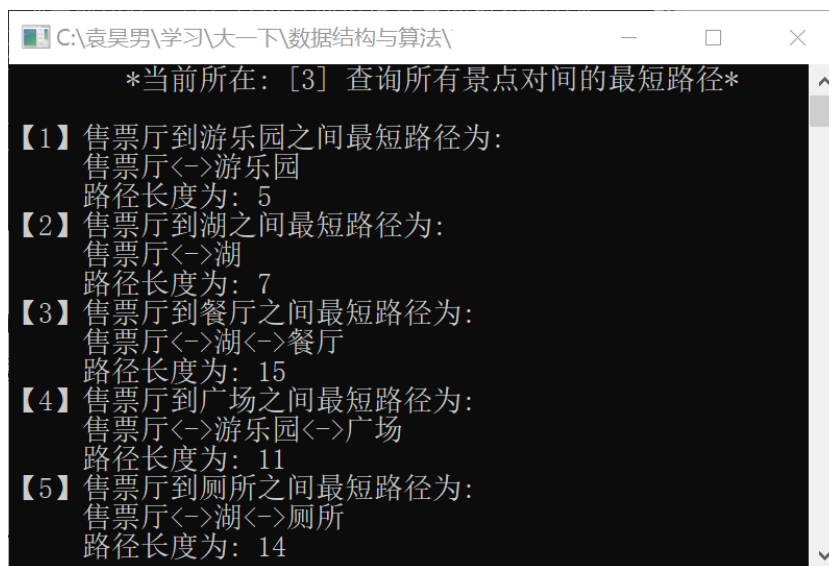


图 5.2.6 计算所有的景点对之间的最短路径

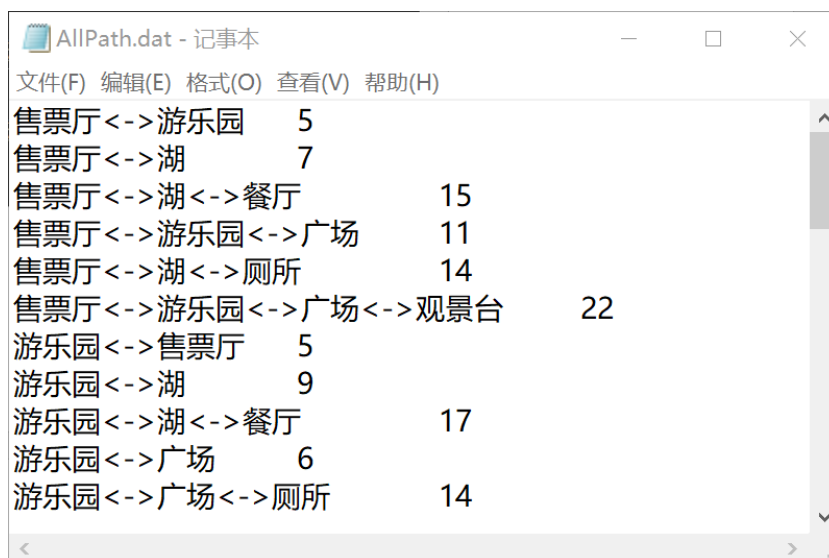


图 5.2.7 所有最短路径保存至 AllPath.dat 文件中

5. 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；

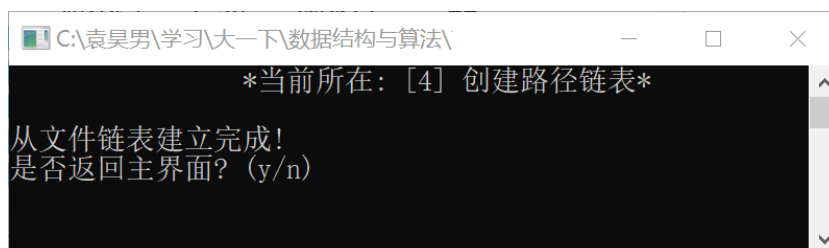


图 5.2.8 从文件中建立路径链表

6. 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；

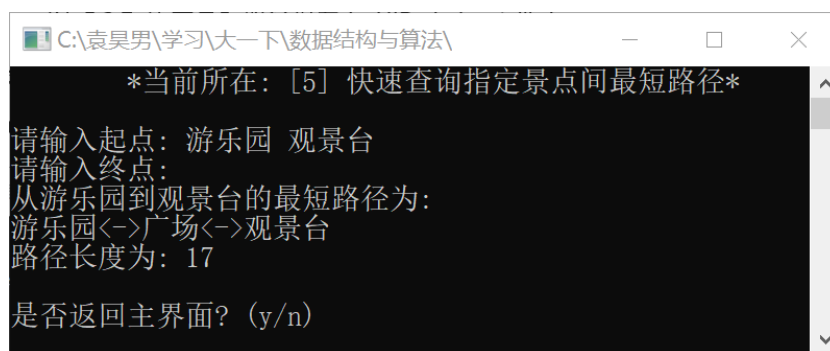


图 5.2.9 从链表快速查询景点间最短路径

六、 实验器材(设备、元器件):

个人电脑一台。

七、 实验步骤:

1. 分析程序需求、设计数据结构;
2. 编写二叉树递归前序遍历函数 PreorderTraverse、非递归中序遍历函数 InorderTraverse、非递归后序遍历函数 PostorderTraverse;
3. 编写从先序序列递归创建二叉树函数 CreateRecursive、非递归创建二叉树函数 CreateNonrecursive;
4. 编写图初始化函数 Graph_Init、最短路径函数 Floyd、保存最短路径函数 Dispath1 及 Dispath_All;
5. 编写链表初始化函数 Init_List、从文件中创建链表函数 Create_List;
6. 调试软件, 修改错误, 完善功能以达到实验目的要求;
7. 测试软件各功能是否正常;
8. 实验结果分析。

八、 实验结果与分析(含重要数据结果分析或核心代码流程分析)

1. 【第 1 部分】采用二叉链式存储创建二叉树

```
1. void InitBitree(Bitreeptr tree)
2. {
3.     tree->root = NULL;
4.     tree->depth = 0;
5.     tree->length = 0;
6. }
7. int CreateBitree(Binodeptr *root)
8. {
9.     type input;
```

```

10.  input = getchar();
11.  getchar();
12.  if (input == '#')
13.  {
14.      *root = NULL;
15.      return 0;
16.  }
17.  else
18.  {
19.      *root = (Binode*)malloc(sizeof(Binode));
20.      if (!root)
21.          printf("Fail to create a tree!\n");
22.      (*root)->data = input;
23.      CreateBitree(&((*root)->lchild));
24.      CreateBitree(&((*root)->rchild));
25.  }
26.}

```

代码分析: 首先 InitBitree 函数对二叉树 tree 进行初始化: 根置为 NULL、depth 及 length 置为 0; CreateBitree 函数采用递归方法根据输入的结点序列创建二叉树。注意在递归调用时, 子树参数前加上 “&” 取地址符号。

2. 【第 1 部分】采用先序序列化显示输出序列并存储到文件

```

1. void PreorderTraverse(Binodeptr root, char order[])
2. {
3.     if (root == NULL)
4.     {
5.         order[i++] = '#';
6.         return;
7.     }
8.     order[i++] = root->data;
9.     PreorderTraverse(root->lchild, order);
10.    PreorderTraverse(root->rchild, order);
11.}
12. void SaveToFile(char order[], FILE *fp)
13. {
14.     fp = fopen("preorder.dat", "w");
15.     if (fp == NULL)
16.     {
17.         printf("Fail to open file!");
18.         exit(0);
19.     }
20.     char *p = &order[0];
21.     while (*p != '\0')
22.     {
23.         fprintf(fp, "%c", *p);
24.         p++;
25.     }
26.     fclose(fp);

```

```
27. }
```

代码分析: PreorderTraverse 函数先序遍历二叉树: 当树为空时, 序列数组内存放“#”(其中 i 是全局变量); 当树不为空时, 把 root->data 值赋给 order 数组的对应下标单元内暂存。SaveToFile 函数采用 C 语言的文件读写操作打开新建的新建空白文件 preorder.dat 并将 order 数组里暂存的序列信息以 fprintf 函数格式化输出至文件, 注意最后加上 fclose(fp)关闭文件指针。

3. 【第 1 部分】从文件序列用反序列化的递归方法构造二叉树

```
1. void ReadFromFile(char order[], FILE *fp)
2. {
3.     if ((fp = fopen("preorder.dat", "r")) == NULL)
4.     {
5.         printf("Fail to open file!");
6.         exit(0);
7.     }
8.     fscanf(fp, "%s", order);
9. }
10. int k = 0;
11. void CreateRecursive(char order[], Binodeptr *root)
12. {
13.     type input;
14.     input = order[k++];
15.     if (input == '#')
16.     {
17.         *root = NULL;
18.         return 0;
19.     }
20.     else
21.     {
22.         *root = (Binode*)malloc(sizeof(Binode));
23.         if (!root)
24.             printf("Fail to create a tree!\n");
25.         (*root)->data = input;
26.         CreateRecursive(order, &((*root)->lchild));
27.         CreateRecursive(order, &((*root)->rchild));
28.     }
29. }
```

代码分析: ReadFromFile 函数使用 C 语言的文件读写操作打开 preorder.dat 文件, 并把文件里保存的结点序列读入到字符数组 order 中。CreateRecursive 函数递归地从字符数组 order 中存储的结点序列建立二叉树。注意在递归调用时, 子树参数前加上“&”取地址符号。

4. 【第 1 部分】从文件序列用反序列化的非递归方法构造二叉树

```
1. void CreateNonrecursive(char order[], Binodeptr *root)
```

```

2. {
3.     *root = (Binode*)malloc(sizeof(Binode));
4.     int j = 0;
5.     type input;
6.     Binodeptr back, origin;
7.     back = (Binode*)malloc(sizeof(Binode));
8.     origin = (Binode*)malloc(sizeof(Binode));
9.     origin = *root;
10.    Binodeptr s[MAXSIZE];
11.    int top = -1;
12.    input = order[j++];
13.    (*root)->data = input;
14.    s[++top] = *root;
15.    input = order[j++];
16.    Binodeptr p, q, r;
17.    p = (Binodeptr)malloc(sizeof(Binode));
18.    (*root)->lchild = p;
19.    *root = (*root)->lchild;
20.    while (top != -1)
21.    {
22.        if (input != '#')
23.        {
24.            (*root)->data = input;
25.            s[++top] = *root;
26.            q = (Binodeptr)malloc(sizeof(Binode));
27.            (*root)->lchild = q;
28.            *root = (*root)->lchild;
29.            input = order[j++];
30.        }
31.        else if (input == '#')
32.        {
33.            *root = NULL;
34.            r = (Binodeptr)malloc(sizeof(Binode));
35.            s[top]->rchild = r;
36.            *root = s[top--]->rchild;
37.            input = order[j++];
38.        }
39.    }
40.    s[++top] = origin;
41.    r = (Binodeptr)malloc(sizeof(Binode));
42.    origin->rchild = r;
43.    *root = origin->rchild;
44.    while (top != -1)
45.    {
46.        if (input != '#')
47.        {
48.            (*root)->data = input;
49.            s[++top] = *root;
50.            q = (Binodeptr)malloc(sizeof(Binode));
51.            (*root)->lchild = q;

```

```

52.         *root = (*root)->lchild;
53.         input = order[j++];
54.     }
55.     else if (input == '#')
56.     {
57.         *root = NULL;
58.         r = (Binodeptr)malloc(sizeof(Binode));
59.         s[top]->rchild = r;
60.         *root = s[top--]->rchild;
61.         input = order[j++];
62.     }
63. }
64. *root = origin;
65.}

```

代码分析：非递归地建立二叉树比递归方法要复杂得多。主要思想为借用栈和两个辅助指针 `back` 及 `origin` 来记录当前结点的上一个结点或下一个结点。首先初始化栈的指示变量 `top` 为 -1, `input` 读入第一个序列并将其存入根 `root` 中。首先建立左子树：当栈不空，即 `top != -1` 时，循环执行以下过程：如果读入的下一个序列为字符“#”，则当前所在元素结点为 `NULL`，并通过 `top` 的增减将结点指针移动至上一结点的右子树，继续循环直至栈空。然后以同样的循环过程建立右子树。最后注意将 `root` 指针重新指回最初的根结点。

5. 【第 1 部分】使用非递归方法输出二叉树中序遍历序列

```

1. void InorderTraverse(Binodeptr root, char order[])
2. {
3.     int i = 0;
4.     Binodeptr s[MAXSIZE];
5.     Binodeptr p = root;
6.     int top = -1;
7.     if (root == NULL)
8.         return;
9.     while (p != NULL || top != -1)
10.    {
11.        while (p != NULL)
12.        {
13.            if (top < MAXSIZE - 1)
14.                s[++top] = p;
15.            else
16.                printf("Overflow!\n");
17.            p = p->lchild;
18.        }
19.        if (top == -1)
20.            return;
21.        else
22.        {

```

```

23.         p = s[top--];
24.         if (p->lchild == NULL)
25.             order[i++] = '#';
26.         order[i++] = p->data;
27.         if (p->rchild == NULL)
28.             order[i++] = '#';
29.         p = p->rchild;
30.     }
31. }
32. }

```

代码分析：对于二叉树，其先序遍历、中序遍历和后序遍历的遍历路线都是从根结点开始的同一路线，只是各结点访问的时机不同而已。在不断深入和返回的过程中，返回结点的顺序和深入结点的顺序正好相反，即后深入先返回，这正是栈的特征，因此可以用栈来实现二叉树的非递归遍历算法。沿左子树深入时，深入一个结点就入栈一个结点，如果沿左子树无法继续深入时，就返回(即出栈)，同时访问此结点，然后从此结点的右子树继续深入。这样一直下去，最后从根结点的右子树返回时结束。以 `top` 的值作为判定栈空与否的条件及作为 `s` 数组下标值。

6. 【第 1 部分】使用非递归方法输出二叉树后序遍历序列

```

1. void PostorderTraverse(Binodeptr root, char order[])
2. {
3.     int i = 0;
4.     int flag = 0;
5.     Binodeptr s[MAXSIZE];
6.     Binodeptr r, t, p = root;
7.     int top = -1;
8.     do
9.     {
10.         if (p != NULL)
11.         {
12.             while (p != NULL)
13.             {
14.                 s[++top] = p;
15.                 p = p->lchild;
16.             }
17.             order[i++] = '#';
18.         }
19.         else
20.         {
21.             r = NULL;
22.             flag = 1;
23.             while (top != -1 && flag)
24.             {
25.                 p = s[top];
26.                 if (p->rchild == r)

```



```

27.         {
28.             if (p->rchild == NULL)
29.                 order[i++] = '#';
30.                 order[i++] = p->data;
31.                 r = p;
32.                 p = s[--top];
33.             }
34.         else
35.         {
36.             p = p->rchild;
37.             flag = 0;
38.         }
39.     }
40. }
41. } while (top != -1);
42. }

```

代码分析：二叉树的非递归后序遍历与非递归中序遍历类似，同样用栈来实现。沿左子树深入时，深入一个结点就入栈一个结点，如果沿左子树无法继续深入时，就返回(即出栈)，然后从此结点的右子树继续深入；当沿右子树无法继续深入时，就返回(即出栈)，同时访问此结点。这样一直下去，最后栈空时结束。以 top 的值作为判定栈空与否的条件及作为 s 数组下标值。

7. 【第 1 部分】销毁释放二叉树

```

1. void Destroy(Binodeptr root)
2. {
3.     if (root == NULL)
4.         return;
5.     Destroy(root->lchild);
6.     Destroy(root->rchild);
7.     free(root);
8. }

```

代码分析：Destroy 函数递归地销毁二叉树的每个结点。首先判空，如果该二叉树已经是一棵空树，则返回；若树不空，则递归地调用左右子树，并使用 free 函数回收每个结点占用的内存。

8. 【第 2 部分】输入景点信息并存到磁盘文件中

```

1. void Info_Input(int n, Info info[])
2. {
3.     int i = 0;
4.     for (i = 0; i < n; i++)
5.     {
6.         char ch[10];
7.         scanf("%s", &ch);

```

```

8.         strcpy(info[i].begin, &ch);
9.         scanf("%s", &ch);
10.        strcpy(info[i].end, &ch);
11.        scanf("%d", &info[i].weight);
12.    }
13.}
14.void Create_File()
15.{
16.    FILE *fp1, *fp2;
17.    fp1 = fopen("info.dat", "w");
18.    fp2 = fopen("AllPath.dat", "w");
19.    if ((!fp1)||(!fp2))
20.    {
21.        printf("Fail to create file!");
22.        exit(0);
23.    }
24.    fclose(fp1);
25.    fclose(fp2);
26.}
27.void Info_Output(int n, Info info[], FILE *fp)
28.{
29.    int i = 0;
30.    fp = fopen("info.dat", "w");
31.    if (fp == NULL)
32.    {
33.        printf("Fail to open file!");
34.        exit(0);
35.    }
36.    for (i = 0; i < n; i++)
37.    {
38.        fprintf(fp, "%s\t", info[i].begin);
39.        fprintf(fp, "%s\t", info[i].end);
40.        fprintf(fp, "%d\t", info[i].weight);
41.        if (i != n - 1)
42.            fprintf(fp, "\n");
43.    }
44.    fclose(fp);
45.}

```

代码分析：首先 Info_Input 函数循环读入输入的景点名称字符串 begin 及 end 及整型路径距离 weight，前者使用字符串赋值函数 strcpy 将对应字符串复制到结构体数组 info 对应成员中，后者直接赋值到 info 对应成员中。Create_File 函数创建空白信息文件 info.dat 和 AllPath.dat，注意在函数最后用 fclose 函数关闭文件指针。Info_Output 函数采用 C 语言的文件读写操作将结构体 info 中的信息使用 fprintf 函数格式化地输出到 info.dat 文件中，注意在输出最后一行信息时不要再多输出一个换行符“\n”，以免读入信息时产生不必要的错误。同样在函数结束时注意用 fclose 函数关闭指针。

9. 【第2部分】Floyd 算法

```

1. void Floyd(MatGraph *g, int A[MAXV][MAXV], int path[MAXV][MAXV])
2. {
3.     int i, j, k;
4.     for (i = 0; i < (*g).n; i++)
5.         for (j = 0; j < (*g).n; j++)
6.             {
7.                 A[i][j] = (*g).edges[i][j];
8.                 if (i != j && (*g).edges[i][j] < INF)
9.                     path[i][j] = i;
10.                else
11.                    path[i][j] = -1;
12.            }
13.     for (k = 0; k < (*g).n; k++)
14.     {
15.         for (i = 0; i < (*g).n; i++)
16.             for (j = 0; j < (*g).n; j++)
17.                 if (A[i][j] > A[i][k] + A[k][j])
18.                     {
19.                         A[i][j] = A[i][k] + A[k][j];
20.                         path[i][j] = path[k][j];
21.                     }
22.     }
23. }

```

代码分析：首先将邻接矩阵信息读入到二维数组 $A[i][j]$ 中，当顶点 i 到顶点 j 有边时，将二维数组 $path[i][j]$ 的值置为 i ，否则置为 -1。接下来依次考察所有顶点，当路径 $i \rightarrow j$ 的长度大于路径 $i \rightarrow k \rightarrow j$ 的长度时，修改 $i \rightarrow j$ 的最短路径长度、修改最短路径。

10. 【第2部分】输出多源最短路径

```

1. int Dispath_All(MatGraph *g, int A[][MAXV], int path[][MAXV], Path output[])
2. {
3.     for (i = 0; i < (*g).n; i++)
4.         for (j = 0; j < (*g).n; j++)
5.             {
6.                 if (A[i][j] != INF && i != j)
7.                     {
8.                         MatchName(g, i);
9.                         strcpy(begin1, temp1[0].name);
10.                        MatchName(g, j);
11.                        strcpy(end1, temp1[0].name);
12.                        printf("【%d】从%s到%s的最短路径\n", sum, begin1, end1);
13.                        strcpy(output[way].end, end1);
14.                        sum++;

```

```

15.         k = path[i][j];
16.         d = 0; apath[d] = j;
17.         while (k != -1 && k != i)
18.         {
19.             d++; apath[d] = k;
20.             k = path[i][k];
21.         }
22.         d++; apath[d] = i;
23.         printf("        %s", begin1);
24.         strcpy(output[way].path, begin1);
25.         strcpy(output[way].begin, begin1);
26.         for (s = d - 1; s >= 0; s--)
27.         {
28.             MatchName(g, apath[s]);
29.             strcpy(name1, temp1[0].name);
30.             printf("<->%s", name1);
31.             strcat(output[way].path, "<->");
32.             strcat(output[way].path, name1);
33.         }
34.         printf("\n");
35.         printf("        路径长度
为: %d\n", A[i][j]);
36.         output[way++].length = A[i][j];
37.         printf("\n");
38.     }
39. }
40. }

```

代码分析：定义整型数组 apath[MAXV]存放一条路径中间顶点，整型变量 d 存放该路径的顶点个数。使用两个嵌套的 for 循环语句，若顶点 *i* 和 *j* 之间存在路径，则先在路径上添加终点，当路径长度 $k \neq -1$ 且 $k \neq i$ 时，在路径上添加中间点；后在路径上添加起点并输出起点。最后使用 for 循环输出所有中间顶点完成一条最短路径的输出。

其中涉及到两个问题：将输入的景点名称转换为结点编号以及将结点编号转换为结点名称输出。为了解决这两个问题，编写了两个函数 MatchNo 和 MatchName 进行转换。

```

1. int MatchNo(MatGraph *g, char name[10])
2. {
3.     int i;
4.     for (i = 0; i < (*g).n; i++)
5.     {
6.         if (strcmp((*g).vexs[i].name, name) == 0)
7.             return (*g).vexs[i].no;
8.     }
9. }
10. void MatchName(MatGraph *g, int no)

```

```

11.{
12.    int i;
13.    for (i = 0; i < (*g).n; i++)
14.        if ((*g).vexs[i].no == no)
15.            strcpy(temp1[0].name, (*g).vexs[i].name);
16.}

```

代码分析：其中 MatchNo 函数返回值为 int 类型，使用一个简单的 for 循环在图 g 的顶点名称数组 vexs 中对应查找，如果查找成功(即 strcmp 函数值为 0)则返回改对应顶点的编号值。MatchName 函数因为返回 char 类型比较复杂，因此使用 void 返回类型，使用简单的 for 循环，当查找成功时，使用全局数组 temp1 来保存查找到的对应编号的景点名称。

11. 【第 2 部分】从路径信息文件创建链表

```

1. void Init_List(LinkListPtr list)
2. {
3.     Nodeptr p;        //头结点
4.     if (*list == NULL)
5.     {
6.         p = (Nodeptr)malloc(sizeof(Node));
7.         if (p == NULL)
8.             printf("Fatal!");
9.         *list = p;
10.        (*list)->next = NULL;
11.    }
12.}
13.void Create_List(int n, Path output[], LinkListPtr list)
14.{
15.    Nodeptr p, r;
16.    p = (Nodeptr)malloc(sizeof(Node));
17.    r = (Nodeptr)malloc(sizeof(Node));
18.    (*list)->next = p;
19.    (*list)->next = r;
20.    p->next = NULL;
21.    int i = 0;
22.    for (i = 0; i < n; i++)
23.    {
24.        Nodeptr q;
25.        q = (Nodeptr)malloc(sizeof(Node));
26.        strcpy(q->begin, output[i].begin);
27.        strcpy(q->end, output[i].end);
28.        strcpy(q->path, output[i].path);
29.        q->length = output[i].length;
30.        q->next = p;
31.        r->next = q;
32.        r = q;
33.    }
34.}

```

代码分析：首先 Init_List 函数对链表进行初始化：为头结点分配空间，并将 next 域置为 NULL；Create_List 函数采用尾插法，路径信息已经从文件中读取到了 output 结构体中，于是使用 for 语句，循环从 output 结构体中读取信息存入结点、建立链表。

12. 【第 2 部分】从链表快速查询景点间最短路径

```
1. void List_Query(LinkListPtr list)
2. {
3.     char begin[10];
4.     char end[10];
5.     printf("请输入起点: ");
6.     scanf("%s", begin);
7.     fflush(stdin);
8.     printf("请输入终点: ");
9.     scanf("%s", end);
10.    Nodeptr p;
11.    p = (Nodeptr)malloc(sizeof(Node));
12.    p = (*list)->next;
13.    while (p != NULL)
14.    {
15.        if (strcmp(p->begin, begin) == 0 && strcmp(p->end
16.        , end) == 0)
17.        {
18.            printf("\n");
19.            printf("从%s 到%s 的最短路径
20.            为: \n", begin, end);
21.            printf("%s\n", p->path);
22.            printf("路径长度为: %d\n", p->length);
23.            break;
24.        }
25.        p = p->next;
26.    }
27.}
```

代码分析：List_Query 函数能在路径信息链表中快速查询景点间最短路径，首先定义指针 p 指向第一个路径信息结点，通过 p = p->next 语句遍历链表，并比较起点与终点信息，当比较成功时(即 strcmp 函数返回值为 0)输出该条路径并退出循环。

九、 总结及心得体会：

本次实验的两个部分：二叉树的序列化及反序列化、公园景点最短路径查询程序考察了对数据结构中数组、结构体、链表、栈、二叉树、图的理解及操作；考察了对二叉树递归/非递归前序、中序、后续遍历算法、递归/非递归从序列建立二叉

树算法、最短路径 Dijkstra/Floyd 算法的应用及 C 语言文件读写的一般方法，既巩固了 C 语言的基本语法及操作，又对本学期数据结构的相关知识进行实际应用与操作。通过本次实验，我掌握了有关知识及其应用，熟悉了在 Visual Studio 环境下编写代码、测试代码的方法及编写程序的一般步骤，掌握了对算法的时间复杂度、正确性进行理论分析的能力。

十、 对本实验过程及方法、手段的改进建议：

本次实验内容主要集中于二叉树、图中的重点内容，能较好地考察学生对于该部分内容的掌握情况。美中不足的是，第一部分的考察内容过于陈旧、没有亮点，并且两部分内容之间的关联度不高。

改进建议：根据实际情况更合理设计实验内容，把这两部分的考察内容更紧密地联系起来，融合为一个综合性更强的实验。这样不仅能激发起学生解决问题的兴趣，还能巩固学生对该部分内容知识的掌握，提高实验效率。

报告评分：

指导教师签字：