

西安交通大学实验报告(1)

姓名： 刘沁宇

班级： 计算机 002

学号： 2203613019

课程名称： 数据结构 实验名称： 约瑟夫环仿真问题

学 院： 电信学部 实 验 日 期 2021 年 11 月 28 日

一、实验目的

1. 熟悉掌握线型表的基本操作在两种存储结构上的实现的，其中以各种链表的操作和应用作为重点。
2. 利用链式存储结构模拟此过程，按照出列的顺序输出各个数的编号。

二、实验环境

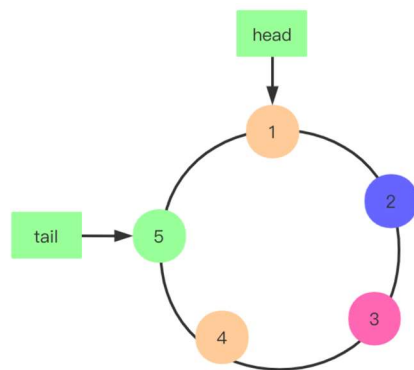
VC++6.0.

三、项目设计和实现

问题中以按照输入的序号标示某个人，所以结点的 num 域和 password 域设为一个整型类型的变量。

```
1. typedef struct LinkNode
2. {
3.     int num;           //序号
4.     int password;      //密码
5.     struct LinkNode* next;
6. }LinkNode;
```

当某人出圈后，报数的工作要从下一个人开始继续，剩下的人仍然围成一圈，可以使循环链表。由于出圈的人将不再属于圈内，意味着数据元素的删除。因此，算法中存有频繁的元素删除操作，存储结构宜采用链表。每个结点中既存储密码还存储初始位置，所以结点有两个数据域，一个指针域。另外，每个结点代表一个人。所以，可以令尾结点指针指向首元结点来进行循环，并利用头结点中的 password 域来记录当前循环链表中的结点数目。



四、实验结果

```
Microsoft Visual Studio 调试控制台
总人数:7
7个人的密码依次为:3 1 7 2 4 8 4
初始报数上限值:20
出列人的编号序列为:6 1 4 7 2 3 5

D:\visual studio\sorces\repos\约瑟夫环\Debug\约瑟夫环.exe
要在调试停止时自动关闭控制台，请启用“工具”->“调试”->“调试时自动关闭控制台”。
按任意键关闭此窗口. . .
```

五、实验总结

通过本次实验，我对链表的构造和删除等操作有了更为深刻的认识，在链表删除一个结点往往需要知道其前驱结点，但有时候可能前驱结点已经访问过了，无法再找到，因此在初始时，就可以再添加一个 p 的前驱结点 pre ，当 p 结点后移时， pre 也后移，这样就保证了 p 的前驱结点始终能够访问到，这就为删除接结点提供了极大的便利，这可以称之为双结点遍历法。同时本次实验也使我更加知道自己的不足和需要加强的地方，指针仍然是自己需要去加强巩固的地方。每次看到自己写的程序和网上的比较发现自己需要完善程序，而不只是编出而已，必须让自己的程序运行结果一目了然。数据结构让我感觉很有意思，让我没有了学习 $c++$ 的茫然和烦恼，实验也很贴近每章学习的内容，很有效的巩固了我们的知识。

西安交通大学实验报告（2）

姓名： 刘沁宇

班级： 计算机 002

学号： 2203613019

课程名称： 数据结构 实验名称： 农夫过河问题

学 院： 电信学部 实 验 日 期 2021 年 11 月 28 日

一、实验目的

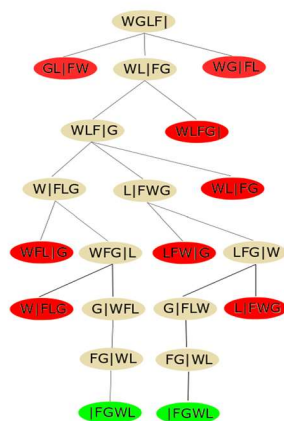
1. 掌握利用图论建模和解决实际问题的能力
2. 掌握图的基本操作（创建图，DFS 和 BFS）

二、实验环境

VC++6.0.

三、项目设计和实现

模拟农夫过河问题，首先需要选择一个对问题中每个角色的位置进行描述的方法，一个很方便的办法是用四位二进制数顺序分别表示农夫、狼、白菜和羊的位置。例如用 0 表示农夫或者某东西在河的南岸，1 表示在河的北岸。因此可以列举出 16 种情景，其中有 6 种情形是不安全的。从初始状态二进制 0000(全部在河的南岸) 出发，寻找一种全部由安全状态构成的状态序列，它以二进制 1111(全部到达河的北岸) 为最终目标，并且在序列中的每一个状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达，因此该模型可看成 16 个顶点的有向图，并采用广度优先或深度优先的搜索策略---得到从 0000 到 1111 的安全路径。



实现 DFS 的前提是建立图或邻接表进行存储。我们选择用邻接表存储，将问题转化成如何建立邻接表的节点并构建相邻节点。邻接表的每个元素表示一个可以安全到达的中间状态。另外还需要一个数据结构记录已被访问过的各个状态，以及已被发现的能够到达当前这个状态的路径。首先建立结点，包含农夫、狼、羊、白菜四个属性，最初状态均是 0。

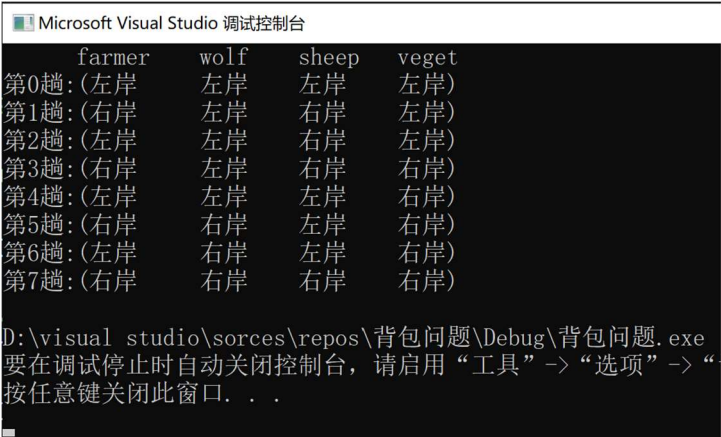
```
1. typedef struct VertexType //顶点的类型
2. {
3.     int farmer;
4.     int wolf;
5.     int sheep;
6.     int vegetable;
7. }VertexType;
```

设 visited 数组对已访问的顶点进行标记（图的遍历），visited 的每个分量初始化值均为 0，每当我们在队列中加入一个新状态时，就把顺序表中以该状态作下标的元素的值改为达到这个状态的路径上前一状态的下标值。visited 的一个元素具有非 0 值表示这个状态已访问过，或是正被考虑。最后我们可以利用 visited 顺序表元素的值建立起正确的状态路径。isSafe 函数是确定状态的安全性，通过位置分布的代码来判断当前状态是否安全，当农夫与羊不在一起时，狼与羊或羊与白菜在一起是不安全的，返回 false，否则返回 true。isConnect 函数是判断两点是否有边的函数，条件是农夫的状态改变，且狼、羊、白菜中最多有一个状态改变则两个点相互可达。

```
1. //判断情景是否安全
2. bool IsSafe(int farmer, int wolf, int sheep, int vegetable)
3. {
4.     //狼与羊或羊与白菜在一起且此时农夫不在场时是不安全的
5.     if (farmer != sheep && (wolf == sheep || sheep == vegetable))
6.         return false;
7.     else
8.         return true;
9. }
10. // 判断状态 i 与状态 j 之间是否可转换
11. bool isConnect(MatGraph G, int i, int j)
12. {
13.     int k = 0;
14.     if (G.Vertexs[i].sheep != G.Vertexs[j].sheep)
15.         k++;
16.     if (G.Vertexs[i].wolf != G.Vertexs[j].wolf)
17.         k++;
18.     if (G.Vertexs[i].vegetable != G.Vertexs[j].vegetable)
19.         k++;
20.     if (G.Vertexs[i].farmer != G.Vertexs[j].farmer && k <= 1) // 以上三个条件不同时满足两个且农夫状态改变时，返回真，也即农夫每次最多只能带一件东西过桥
21.         return true;
22.     else
23.         return false; }
```

搜索过程可利用深度优先搜索算法从初始状态二进制 0000（全部在河的左岸）出发，寻找一种全部由安全状态构成的状态序列，它以二进制 1111（全部到达河的右岸）为最终目标，并且在序列中的每一个状态都可以从前一个状态得到。为避免重复，要求在序列中不出现重复的状态。用数组 path 保存 DFS 搜索到的路径，即与某顶点到下一顶点的路径。

四、实验结果



```
Microsoft Visual Studio 调试控制台
farmer    wolf    sheep    veget
第0趟: (左岸    左岸    左岸    左岸)
第1趟: (右岸    左岸    右岸    左岸)
第2趟: (左岸    左岸    右岸    左岸)
第3趟: (右岸    左岸    右岸    右岸)
第4趟: (左岸    左岸    左岸    右岸)
第5趟: (右岸    右岸    左岸    右岸)
第6趟: (左岸    右岸    左岸    右岸)
第7趟: (右岸    右岸    右岸    右岸)

D:\visual studio\sorces\repos\背包问题\Debug\背包问题.exe
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“
按任意键关闭此窗口。 . . .
```

五、实验总结

面对农夫过河问题，我们可采用手算或者其他数学方法来描述怎么做才能达到目的，但如何采用计算机语言来实现过程是一个难点。而采用四元组形式的结点来表示图是十分巧妙的方法。0 和 1 代表每种生物不同的状态，它或者在北岸，或者在南岸，总共有 16 个结点，但我们还可以用 isSafe 函数每次判断这个结点的状态是不是安全的，因此就能在开始的时候直接筛选出安全状态的结点，不再考虑不安全的，把图的结点数直接降到 10 个。这道题提高了我利用图论和图的数据结构解决问题的能力，也让我对建立图，DFS 算法都有了进一步的认识。

西安交通大学实验报告（3）

姓名： 刘沁宇

班级： 计算机 002

学号： 2203613019

课程名称： 数据结构 实验名称： 二叉排序树与平衡二叉树

学 院： 电信学部 实 验 日 期 2021 年 11 月 28 日

一、 实验目的

1. 熟练掌握排序二叉树的查找，插入和删除方法。
2. 掌握顺序和链序二叉排序树的定义以及基本操作的实现算法。
3. 掌握二叉树平均查找长度的求解方法。
4. 熟练掌握平衡二叉树的构造方法，了解左旋、右旋的含义。

二、实验环境

VC++6.0.

三、项目设计和实现

(1). 以二叉链表作存储结构实现二叉排序树

a). 二叉排序树的构造及插入(采用迭代的方法实现)

- 若二叉排序树为空，则直接插入结点作为根结点；
- 若二叉排序树非空，当值小于根结点时，插入左子树；
- 若二叉排序树非空，当值等于根结点时，不予插入；
- 若二叉排序树非空，当值大于根结点时，插入右子树；

```
1. void L_InsertBSTnode(BSTnode*& BTree, KeyType k)
   //向排序二叉树中插入结点
2. {
3.     if (BTree == NULL)
4.     {
5.         BTree = (BSTnode*)malloc(sizeof(BSTnode));
6.         BTree->key = k;
7.         BTree->lchild = BTree->rchild = NULL;
8.     }
9.     else if (BTree->key == k)
10.         return;
```

```

11.     else if (k < BTree->key)
        //关键字小于当前结点的 key，继续向左子树中查找插入的位置
12.         L_InsertBSTnode(BTree->lchild, k);
13.     else if (k > BTree->key)
14.         L_InsertBSTnode(BTree->rchild, k);
        //关键字大于当前结点的 key，继续向右子树中查找插入的位置
15. }

```

b). 二叉排序树的查找

- 从根结点开始查找，若相等，则查找成功；
- 若查找的值小于根结点，则查找左子树；
- 若查找的值大于根结点，则查找右子树；

c). 二叉排序树的删除

- 若删除的结点为叶子结点，则直接删除；
- 若删除的结点 T 只有一棵子树，则让 T 的子树成为 T 父结点的子树（替代 T，独子继承父业）；
- 若被删除结点 T 有两棵子树，则让 T 的右子树中 key 最小的结点替代 T，再删除该结点；

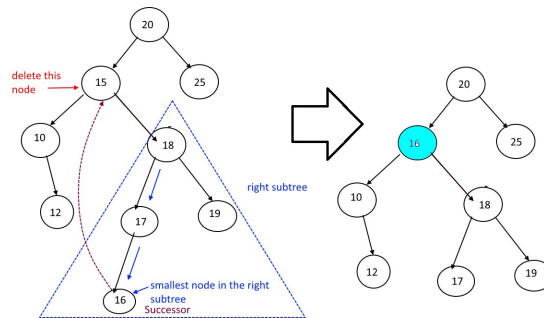


图 1 含有两颗子树的结点的删除过程

d). 计算查找成功的平均查找长度

查找第*i*层的结点关键字比较的次数为*i*，由此可归纳出平均查找长度为

$$ASL = \frac{\sum \text{本层高度} * \text{本层元素个数}}{\text{节点总数}}$$

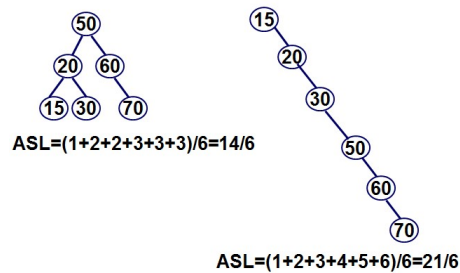


图 2 ASL 计算过程

(2). 以顺序表作存储结构实现二叉排序树

以顺序表(一维数组)作存储结构来实现二叉排序树, 需要先设置数组的长度为 **MaxSize**, 对于非空的下标为 i 的结点, 其左孩子结点的下标为 $2i + 1$, 右孩子结点的下标为 $2i + 2$, 并将空结点的值设为 **NULLKEY**, 并规定 **NULLKEY** 为 0. 以顺序表作存储结构的二叉排序树的插入, 查找, 求 ASL 等方法与二叉链表基本相同, 但是在判断时还需要确保下表不越界, 即不超过 **MaxSize**。

```
1. //向排序二叉树中插入结点
2. void S_InsertBSTNode(KeyType BTree[], KeyType key , int j )
3. {
4.
5.     if (BTree[j] == NULLKEY && j < MaxSize)
6.     {
7.         BTree[j] = key;
8.     }
9.     else if (BTree[j] == key )
10.        return;
11.    else if (key < BTree[j])
12.        S_InsertBSTNode(BTree, key, 2 * j + 1);
13.    else if (key > BTree[j])
14.        S_InsertBSTNode(BTree, key, 2 * j + 2);
15. }
```

(3). 用二叉链表作存储结构实现平衡的二叉排序树

a). LL 型调整: 这是因为在 A 结点的左孩子 (设为 B 结点) 的左子树上插入结点, 使得 A 结点的平衡因子由 1 变为 2 引起的不平衡。

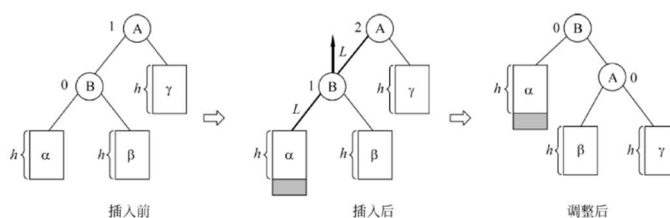


图 3 LL 型调整过程 (右旋)

b). RR 型调整: 这是因为在 A 结点的右孩子 (设为 B 结点) 的右子树上插入结点, 使得 A 结点的平衡因子由-1 变为-2 引起的不平衡。

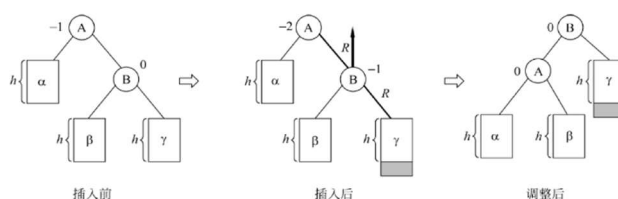


图 4 RR 型调整过程 (左旋)

c). LR 型调整：这是因为在 A 结点的左孩子（设为 B 结点）的右子树上插入结点，使得 A 结点的平衡因子由 1 变为 2 引起的不平衡。

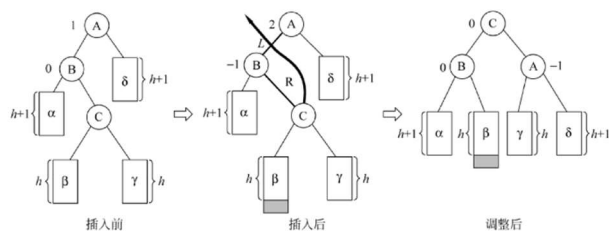


图 5 LR 型调整过程（先左旋再右旋）

d). RL 型调整：这是因为在 A 结点的右孩子（设为 B 结点）的左子树上插入结点，使得 A 结点的平衡因子由 -1 变为 -2 引起的不平衡。

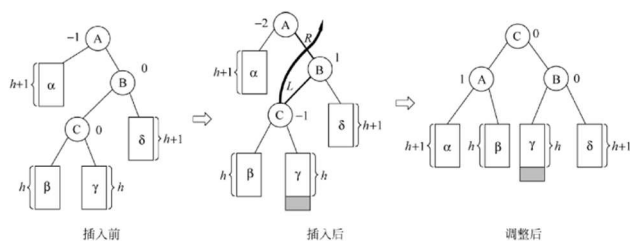


图 6 RL 型调整过程（先右旋再左旋）

四、实验结果

```
Microsoft Visual Studio 调试控制台
Input a sequence stored in linked-list BSTree:5 2 1 6 7 4 8 3 9
linked-list bstree midorder:1 2 3 4 5 6 7 8 9
Average search length :3.00
The key of the BTreeNode that needs to be deleted is 5
Deleting 5 successfully!
After deleting 5,Linked-List BSTree MidOrder:1 2 3 4 6 7 8 9
D:\visual studio\sorces\repos\7text\Debug\7text.exe (进程 4016) 已调
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->
按任意键关闭此窗口。...
```

图 7 以二叉链表作存储结构实现二叉排序树

```
Microsoft Visual Studio 调试控制台
Input a sequence stored in SqList BSTree:5 2 1 6 7 4 8 3 9
SqList-BSTree MidOrder:1 2 3 4 5 6 7 8 9
Average Search Length :3.00
The key of the BTreeNode that needs to be deleted is 5
Deleting 5 successfully!
After deleting 5,SqList-BSTree MidOrder:1 2 3 4 6 7 8 9
D:\visual studio\sorces\repos\7text\Debug\7text.exe (进程 710
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调
按任意键关闭此窗口。...
```

图 8 以顺序表作存储结构实现二叉排序树



```
Microsoft Visual Studio 调试控制台
Input a sequence stored in AVL:5 2 1 6 7 4 8 3 9
Average search length :2.778
D:\visual_studio\sources\repos\7text\Debug\7text.exe (进
要在调试停止时自动关闭控制台，请启用“工具”->“选项”
按任意键关闭此窗口. . .
```

图 9 以二叉链表作存储结构实现平衡二叉树

五、实验总结

本次排序二叉树和平衡二叉树实验，使我对二叉树的性质有了更为深刻的认识。对于需要频繁插入和删除结点的二叉树，适合采用二叉链表作为存储结构，以顺序标作为存储结构的二叉树适合用于已知结点数目的满二叉树或完全二叉树，能够节省存储空间。二叉树中插入和遍历等操作，往往可以通过迭代的方法来实现，通过本次实验，提高我对迭代方法认知，也让我能熟练运用迭代方法来解决其他有关树的问题。在实验过程中我对排序二叉树的用途也有了新的认识，比如对排序二叉树进行中序遍历，就能得到由小到的一组序列，且对以排序二叉树作为存储结构的数据进行查找具有很高的效率。

西安交通大学实验报告(4)

姓名： 刘沁宇

班级： 计算机 002

学号： 2203613019

课程名称： 数据结构 实验名称： 迷宫问题

学 院： 电信学部 实 验 日 期 2021 年 11 月 28 日

一、实验目的

1. 掌握利用 BFS 和 DFS 解决迷宫问题的方法。
2. 掌握 BFS 算法中的迭代思想。
3. 掌握队列在 DFS 算法中的应用。

二、实验环境

VC++6.0.

三、项目设计和实现

求解迷宫问题需要找到从起点到终点的一条路径，用图论知识来说，即起点与终点之间是连通的。首先可用矩阵 **maze** 存储迷宫，矩阵中为 **0** 的位置表明通路，为 **1** 的位置表明有障碍。

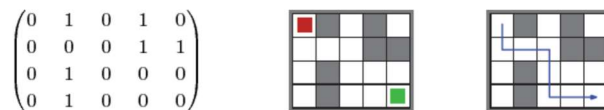


图 1 迷宫的矩阵形式

而移动方向可用二维数组 **direct** 表示， $direct = \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix}$ ，分别表示上、右、下、左四个方向。

(1) 利用 DFS 算法求解

用数组 **path** 存储可通过的路径，**step** 表示当前路径长度，**length** 表示最终路径长度，数组 **visited** 判断当前位置是否已经走过

实现步骤：

- (a). 如果起点和终点合法，将起点的 **visited** 置为 **true**

- (b). 按照向上、向右、向下、向左的顺序判断下一个位置是否为通路，若为通路，则将其 `visited` 置为 `true`，`step` 增加 1，并用 `path` 记录当前位置并再执行步骤(b)；若不为通路则返回上一位置，并接着按顺序判断。
- (c). 当到达通路的位置与出口相同时，表明找到了一条路径，若所有路径遍历完后仍未有通路与出口匹配，则表示路径不存在。

```
1. bool DFS(int r, int c, int x, int y, int step, int &length)
2. {
3.     int i, x1, y1;
4.     if (x == r && y == c)
5.     {
6.         length = step + 1;
7.         //length 记录路径长度，由于 step 初值为 0，因此长度为 step+1
8.         return true;
9.     }
10.    visited[x][y] = true;
11.    //用 path 记录路径，当结点可访问时，step 增 1
12.    step++;
13.    //step 要放在循环外，否则当遇到路口时，step 增量可能会超过 1
14.    for (i = 0; i < 4; i++)
15.    {
16.        x1 = x + direct[i][0];
17.        y1 = y + direct[i][1];
18.        if (0 <= x1 && x1 <= r && 0 <= y1 && y1 <= c)
19.        //判断该位置是否超过迷宫边界
20.        {
21.            if (visited[x1][y1] != true && maze[x1][y1] != 1)
22.            //若该位置还未访问且不为障碍物
23.            {
24.                path[step][0] = x1;
25.                path[step][1] = y1;
26.                if(DFS(r, c, x1, y1, step, length))
27.                //当下一个位置可达时返回 true
28.                return true;
29.            }
30.        }
31.    }
32.    return false;
33.    //若没找到路径（或所有方向均到达不了）则返回 false
34. }
```

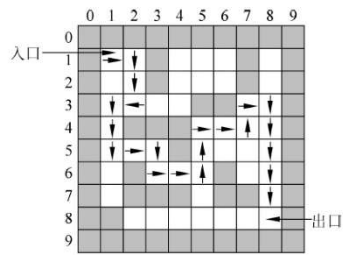


图 2 DFS 求解的路径

利用 DFS 能够简单快速地求解迷宫问题的一个解，但是找到的第一条可行路径不一定是最短路径，如果需要找到最短路径，那么需要找出所有可行路径后，再逐一比较，求出最短路径。因此可考虑用 BFS 来求解最短路径。

(2).利用 BFS 算法求解

迷宫中的每个位置用结点 **node** 表示，且 **node** 中包含 5 个成员

```
1. struct node //构建迷宫的结点
2. {
3.     int x;
4.     int y;
5.     int flag; //flag 为 0 表示可通过
6.     bool visited; //判断是否访问
7.     node* pre; //用 pre 记录该结点的前驱结点（类似树的存储结构）
8. }
```

实现方法:

首先将入口 (x,y) 进队，在队列 Queue 不空时循环，出队一个方块 e 。然后查找 e 的所有相邻可走方块，假设为 e_1 和 e_2 两个方块，将他们进队，并且将他们对应的 **pre** 均设置为 **front**（因为在迷宫路径上 e_1 和 e_2 两个方块的前一个方块都是 e ），如图 2

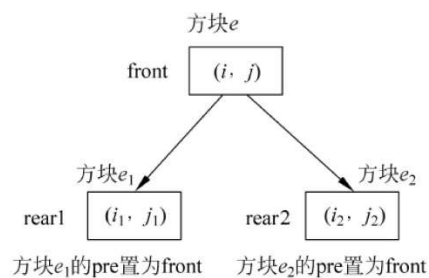


图 3 设置相邻方块的 pre

当找到出口时，通过出口方块的 **pre** 值前推找到出口，所有经过的中间方块构成一条迷宫路径。实际上，上述过程是从入口 (x,y) 开始，利用队列的特点，一层一层向外扩展查找可走的方块，知道找到出口为止。

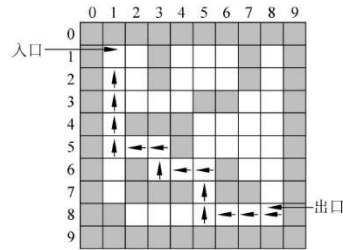


图 4 BFS 求解的路径

四、 实验结果

```
Microsoft Visual Studio 调试控制台
maze:
  0 1 2 3 4 5 6 7
0  0 0 1 0 0 0 1 0
1  0 0 1 0 0 0 1 0
2  0 0 0 0 1 1 0 0
3  0 1 1 1 0 0 0 0
4  0 0 0 1 0 0 0 0
5  0 1 0 0 0 1 0 0
6  0 1 1 1 0 1 1 0
7  1 0 0 0 0 0 0 0

DFS_Path: (0,0) (0,1) (1,1) (1,0) (2,0) (3,0) (4,0) (4,1) (4,2) (5,2) (5,3) (5,4) (4,4) (4,5) (4,6) (4,7) (5,7) (6,7) (7,7)
D:\visual_studio\sorces\repos\迷宫问题\Debug\迷宫问题.exe (进程 18604) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 5 DFS 实验结果

```
Microsoft Visual Studio 调试控制台
maze:
  0 1 2 3 4 5 6 7
0  0 0 1 0 0 0 1 0
1  0 0 1 0 0 0 1 0
2  0 0 0 0 1 1 0 0
3  0 1 1 1 0 0 0 0
4  0 0 0 1 0 0 0 0
5  0 1 0 0 0 1 0 0
6  0 1 1 1 0 1 1 0
7  1 0 0 0 0 0 0 0

BFS_Path: (0,0) (1,0) (2,0) (3,0) (4,0) (4,1) (4,2) (5,2) (5,3) (5,4) (6,4) (7,4) (7,5) (7,6) (7,7)
D:\visual_studio\sorces\repos\迷宫问题\Debug\迷宫问题.exe (进程 25692) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 6 BFS 实验结果

五、 实验总结

(1).在解决迷宫问题时, 对于已经探索过的路径, 可以再加一个 bool 类型的参数 visited 来判断, 这在 DFS 和 BFS 算法中经常会用到。

(2).DFS 算法通常需要用到迭代的思想, 通过本次实验, 我对迭代和回溯算法有了更进一步的认识。

(3).BFS 算法可用于寻找最短路径问题, 因为是一层一层呈辐射型向外遍历的, 但是利用 BFS 算法时通常需要用到队列, 在一定程度上会增加代码的复杂度。因此是用 BFS 算法还是 DFS 算法需要根据实际问题来选择。

代码附录

(1) 约瑟夫环实验

```
1. #include<iostream>
2. #include<cstdlib>
3. #include<vector>
4. #include<fstream>
5. using namespace std;
6. typedef struct LinkNode
7. {
8.     int num;           //序号
9.     int password;      //密码
10.    struct LinkNode* next;
11. }LinkNode;
12.
13. LinkNode* CreatNode(int num, int password)    //结点构造函数
14. {
15.     LinkNode* p;
16.     p = (LinkNode*)malloc(sizeof(LinkNode));
17.     p->num = num;
18.     p->password = password;
19.     return p;
20. }
21.
22. void LinkAssign(LinkNode*& List, int &Initial_Limit)    //链表拼接函数
23. {
24.
25.     LinkNode* p;
26.     int password, n;
27.
28.     //以读模式打开文件
29.     ifstream infile;
30.     infile.open("data.txt");
31.     infile >> n;           //先读取人数
32.
33.     List = (LinkNode*)malloc(sizeof(LinkNode));
34.     List->password = n;     //用头结点的 password 域记录循环链表内的结点总数
35.
36.     cout << "总人数:" << n << endl;
37.     cout << n << "个人的密码依次为:";
38.     LinkNode* end = List;
39.     for (int i = 1; i <= n; i++)
40.     {
41.         infile >> password;
```

```

42.     printf("%d ", password);
43.     p = CreatNode(i, password);
44.     end->next = p;
45.     end = p;
46. }
47. printf("\n");
48. end->next = List->next;    //这里在构造循环链表时, 并没有将尾结点的指针域指向
    头结点, 而是指向首结点(头结点的后继节点), 使得在循环访问链表更方便
49.     infile >> Initial_Limit;
50.     printf("初始报数上限值:%d\n", Initial_Limit);
51.     infile.close();
52. }
53.
54.
55. void GetList(int N, LinkNode*& L, int Initial_Limit, int List[]) //用 List 存储
    出队的序列
56. {
57.     int limit = Initial_Limit;
58.     int password;
59.     int No = 0;
60.     int i;
61.     LinkNode* cur = L->next;
62.     LinkNode* pre = L;
63.     while (N--)                                //当循环链表
        中结点数不为 0 时
64.     {
65.         for (i = 0; i < limit - 1; i++)
66.         {
67.             pre = cur;
68.             cur = cur->next;
69.         }                                        //找到需要被
        删除结点的前驱结点
70.         List[No++] = cur->num;
71.         limit = cur->password;
72.         pre->next = cur->next;
73.         free(cur);
74.         if (pre != NULL)
75.             cur = pre->next;
76.     }
77. }
78.
79. void PrintList(int List[], int NodeNum)          //输出 List 中的序列
80. {
81.     cout << "出列人的编号序列为:";

```



```
82.     for (int i = 0; i < NodeNum; i++)  
83.         cout << List[i] << " ";  
84.     cout << endl;  
85. }  
  
86.  
87.  
88. int main()  
89. {  
90.     LinkNode* Ring;  
91.     int Initial_Limit, NodeNum;  
92.     int List[50];  
93.     LinkAssign(Ring,Initial_Limit);           //构造循环链表  
94.     NodeNum = Ring->password;    //记录初始编号总数  
95.     GetList(NodeNum, Ring, Initial_Limit, List);  
96.     PrintList(List, Ring->password);  
97.     free(Ring);                          //注意头结点不要忘记释放  
98.     return 0;  
99. }
```

(2) 农夫过河实验

```
1. #include<iostream>
2. #include<string>
3. #define MAXV 16
4. using namespace std;
5.
6. typedef struct VertexType //顶点的类型
7. {
8.     int farmer;
9.     int wolf;
10.    int sheep;
11.    int vegetable;
12. }VertexType;
13.
14. typedef struct MatGraph
15. {
16.     int n; //边数和定点数
17.     int edges[MAXV][MAXV]; //邻阶矩阵
18.     VertexType Vertexs[MAXV]; //存放顶点信息
19. }MatGraph; // 定义图的邻接矩阵存储结构
20.
21.
22. int visited[MAXV] = { 0 };
23.
24. //判断情景是否安全
25. bool IsSafe(int farmer, int wolf, int sheep, int vegetable)
26. {
27.     //狼与羊或羊与白菜在一起且此时农夫不在场时是不安全的
28.     if (farmer != sheep && (wolf == sheep || sheep == vegetable))
29.         return false;
30.     else
31.         return true;
32. }
33.
34. //根据位置确定结点序号
35. int locate(MatGraph G, int farmer, int wolf, int sheep, int vegetable)
36. {
37.     for (int i = 0; i < G.n; i++)
38.     {
39.         if (G.Vertexs[i].farmer == farmer && G.Vertexs[i].wolf == wolf && G.Vertexs[i].sheep == sh
40.             return i; //返回当前位置
41.     }
42.     return -1;
43. }
```

```

44.
45. // 判断状态 i 与状态 j 之间是否可转换
46. bool isConnect(MatGraph G, int i, int j)
47. {
48.     int k = 0;
49.     if (G.Vertexs[i].sheep != G.Vertexs[j].sheep)
50.         k++;
51.     if (G.Vertexs[i].wolf != G.Vertexs[j].wolf)
52.         k++;
53.     if (G.Vertexs[i].vegetable != G.Vertexs[j].vegetable)
54.         k++;
55.     if (G.Vertexs[i].farmer != G.Vertexs[j].farmer && k <= 1) // 以上三个条件不同时满足两个且农夫未死
56.         return true;
57.     else
58.         return false;
59. }
60.
61. // 创建连接图
62. void CreateMatGraph(MatGraph &G)
63. {
64.     int i = 0; int j = 0;
65.     for (int farmer = 0; farmer <= 1; farmer++)
66.     {
67.         for (int wolf = 0; wolf <= 1; wolf++)
68.         {
69.             for (int sheep = 0; sheep <= 1; sheep++)
70.             {
71.                 for (int vegetable = 0; vegetable <= 1; vegetable++)
72.                 {
73.                     if (IsSafe(farmer, wolf, sheep, vegetable)) // 生成所有安全的图的顶点
74.                     {
75.                         G.Vertexs[i].farmer = farmer;
76.                         G.Vertexs[i].wolf = wolf;
77.                         G.Vertexs[i].sheep = sheep;
78.                         G.Vertexs[i].vegetable = vegetable;
79.                         i++;
80.                     }
81.                 }
82.             }
83.         }
84.     }
85.     // 邻接矩阵初始化即建立邻接矩阵
86.     G.n = i; // 图中安全结点数量赋值为 i
87.     for (i = 0; i < G.n; i++)

```

```

88.     {
89.         for (j = 0; j < G.n; j++)
90.         {
91.             // 状态 i 与状态 j 之间可转化, 初始化为 1, 否则为 0
92.             if (isConnect(G, i, j) )
93.                 G.edges[i][j] = 1;
94.             else
95.                 G.edges[i][j] = 0;
96.         }
97.     }
98. }
99.
100.     // 判断在河的哪一边
101.     string judgement(int state)
102.     {
103.         if (state == 0)
104.             return "左岸";
105.         else
106.             return "右岸";
107.     }
108.
109.     //输出 path 中的结点信息
110.     void printPath(MatGraph G, int d,int path[])        //d 为路径长度
111.     {
112.         int i,j;
113.         cout << "        farmer\t" << "wolf\t" << "sheep\t" << "veget\t" << endl;
114.         for(i=0;i<=d;i++)
115.         {
116.             j = path[i];
117.             cout <<"第"<<i<<"趟:"<< "(" << judgement(G.Vertexts[j].farmer)<<"\t"<<
118.                 judgement(G.Vertexts[j].wolf) << "\t" << judgement(G.Vertexts[j].sheep) << "\t"
119.                 judgement(G.Vertexts[j].vegetable) << ")";
120.             cout << endl;
121.         }
122.     }
123.
124.     //基于深度优先输出路径
125.     void FindPath(MatGraph G, int start, int end, int path[], int d)
126.     {
127.         int i;
128.         d++;
129.         path[d] = start;
130.         visited[start] = 1;
131.         if (start == end)

```

```

132.         {
133.             printPath(G, d, path);
134.             return;
135.         }
136.         for (i = 0; i < G.n; i++)
137.         {
138.             if (G.edges[start][i] == 1 && visited[i] == 0)
139.             {
140.                 FindPath(G, i, end, path, d);
141.             }
142.         }
143.     }
144.
145.
146.     int main()
147.     {
148.         MatGraph Graph;
149.         CreateMatGraph(Graph);
150.         int start = locate(Graph, 0, 0, 0, 0); //确定初始状态和最终状态对应的图结点的编号
151.         int end = locate(Graph, 1, 1, 1, 1);
152.         int d = -1; //路径长度初始值置为-1
153.         int path[MAXV] = { -1 };
154.         FindPath(Graph, start, end, path, d);
155.         return 0;
156.     }

```

(3). 二叉排序树和平衡二叉树(这里仅附上二叉链表存储的二叉排序树的操作)

```
1. #include<iostream>
2. #include<cstdlib>
3. #include<fstream>
4. #include<sstream>
5. #include<string>
6. #include<math.h>
7. using namespace std;
8. typedef int KeyType;
9. typedef struct BSTnode
10. {
11.     KeyType key;
12.     BSTnode* lchild;
13.     BSTnode* rchild;
14. }BSTnode;
15.
16. void L_InsertBSTnode(BSTnode*& BTree, KeyType k)           //向排序二叉树
    中插入结点
17. {
18.     if (BTree == NULL)
19.     {
20.         BTree = (BSTnode*)malloc(sizeof(BSTnode));
21.         BTree->key = k;
22.         BTree->lchild = BTree->rchild = NULL;
23.     }
24.     else if (BTree->key == k)
25.         return;
26.     else if (k < BTree->key)           //关键字小于当
        前结点的 key, 继续向左子树中查找插入的位置
27.         L_InsertBSTnode(BTree->lchild, k);
28.     else if (k > BTree->key)
29.         L_InsertBSTnode(BTree->rchild, k);           //关键字大于
        当前结点的 key, 继续向右子树中查找插入的位置
30. }
31.
32. BSTnode* L_CreateBSTree(ifstream& infile, int& n)           //读取文件中的数据构
    造一棵排序二叉树
33. {
34.     BSTnode* BTree;
35.     n = 0;           //n 用于记录二叉树中结点数量
36.     BTree = NULL;
37.     string numberline;
38.     int keynum;
39.     getline(infile, numberline);           //读取一行序列
```

```

40.     istream is(numberline);                                //将读出的一行转
    成数据流进行操作
41.     printf("Input a sequence stored in linked-list BSTree:");
42.     while (!is.eof())
43.     {
44.         is >> keynum;
45.         n++;
46.         printf("%d ", keynum);
47.         L_InsertBSTnode(BTree, keynum);
48.     }
49.     return BTree;
50. }
51.
52. void L_MidOrder(BSTnode* btree)                            //输出中序遍历
    结果
53. {
54.     if (btree != NULL)
55.     {
56.         L_MidOrder(btree->lchild);
57.         printf("%d ", btree->key);
58.         L_MidOrder(btree->rchild);
59.     }
60. }
61.
62. bool L_DeleteBSTnode(BSTnode*& BTnode, KeyType k)
    //删除值为 k 的结点
63. {
64.     BSTnode* p;
65.     if (BTnode != NULL)
66.     {
67.         if (k == BTnode->key)
68.         {
69.             if (BTnode->lchild == NULL && BTnode->rchild == NULL)
                //若是叶子结点，则直接删除
70.             {
71.                 p = BTnode;
72.                 BTnode = NULL;
                //注意这里不能直接释放 btree，还要将 btree 的双亲结点的孩子结点设置为 null
73.                 free(p);
74.                 return true;
75.             }
76.             else if (BTnode->lchild != NULL && BTnode->rchild == NULL)
                //若只有左子树，无右子树，则将被删除节点的左孩子结点来代替
77.             {

```

```

78.         p = BTreeNode;
79.         BTreeNode = BTreeNode->lchild;
80.         free(p);
81.         return true;
82.     }
83.     else if (BTreeNode->lchild == NULL && BTreeNode->rchild != NULL)
        //若只有右子树，无左子树，则将被删除节点的右孩子结点来代替
84.     {
85.         p = BTreeNode;
86.         BTreeNode = BTreeNode->rchild;
87.         free(p);
88.         return true;
89.     }
90.     else if (BTreeNode->lchild != NULL && BTreeNode->rchild != NULL)
        //若左右子树都有，则将左子树中 key 最大的结点来代替（即被删除结点的最右下结点，此结点
        //无右子树，但可能有左子树）
91.     {
92.         BSTnode* pre = BTreeNode;
93.         p = BTreeNode->lchild;           //p 指向被删除结点的左子树
94.         while (p->rchild != NULL)
95.         {
96.             pre = p;
97.             p = p->rchild;
98.         }
99.         BTreeNode->key = p->key;
100.        pre->rchild = p->lchild;
101.        free(p);
102.        return true;
103.    }
104. }
105. else if (k < BTreeNode->key)
106.     return L_DeleteBSTnode(BTreeNode->lchild, k);
107. else if (k > BTreeNode->key)
108.     return L_DeleteBSTnode(BTreeNode->rchild, k);
109. }
110. else
111.     return false;
112. }
113.
114.
115. void L_isDeleted(BSTnode*& btnode, ifstream& infile)           //判断是否
    删除成功
116. {
117.     string numberline;

```



```

118.     int x;
119.     getline(infile, numberline);           //读取一行序
        列
120.     istringstream is(numberline);         //将读出的一行
        转成数据流进行操作
121.     is >> x;
122.     printf("The key of the BTreeNode that needs to be deleted is %d\n", x);

123.     if (L_DeleteBSTnode(btnode, x) == true)
124.     {
125.         printf("Deleting %d successfully!\nAfter deleting %d,Linked-
        List BStree MidOrder:", x, x);
126.         L_MidOrder(btnode);
127.     }
128.     else
129.         printf("%d doesn't exist", x);
130. }
131.
132.
133. int L_BTreeHeight(BSTnode* btree)           //求树高
134. {
135.     int left_height, right_height;
136.     if (btree == NULL)
137.         return 0;
138.     else
139.     {
140.         left_height = L_BTreeHeight(btree->lchild);
141.         right_height = L_BTreeHeight(btree->rchild);
142.         return (left_height > right_height) ? (left_height + 1) : (right_he
        ight + 1);           //返回左右子树中高度最大的值
143.     }
144. }
145.
146. void L_CountNode(BSTnode* btree, int h, int k, int& count) //求第 h 层
        的结点数
147. {
148.     if (btree == NULL)
149.         return;
150.     else
151.     {
152.         if (h == k)
153.             count++;
154.         if (k < h)
155.         {

```

```
156.         L_CountNode(btree->lchild, h, k + 1, count);
157.         L_CountNode(btree->rchild, h, k + 1, count);
158.     }
159. }
160. }
161.
162. void L_AS_L(BSTnode* btree, int n)
163. {
164.     int height = L_BTreeHeight(btree);
165.     int i;
166.     float sum = 0;
167.     int nodenum;
168.     for (i = 1; i <= height; i++)
169.     {
170.         nodenum = 0;
171.         L_CountNode(btree, i, 1, nodenum);
172.         sum = i * nodenum + sum;
173.     }
174.     printf("Average search length :%0.2f\n", sum / n);
175. }
176.
177. int main()
178. {
179.     BSTnode* btree;
180.     ifstream infile;
181.     int nodenum;
182.     //打开文件
183.     infile.open("data.txt");
184.     btree = L_CreateBSTree(infile, nodenum);
185.     printf("\nlinked-list btree midorder:");
186.     L_MidOrder(btree);
187.     printf("\n");
188.     L_AS_L(btree, nodenum);
189.     L_isDeleted(btree, infile);
190.     infile.close();
191.     return 0;
192. }
```

(4) 迷宫问题

(a) .DFS 算法

```
1. #include<iostream>
2. #include<string>
3. #include<sstream>
4. #include<fstream>
5. using namespace std;
6. #define MaxSize 50
7. int maze[MaxSize][MaxSize];
8. bool visited[MaxSize][MaxSize];
9. int path[MaxSize][2];
10. int direct[4][2] = { 0,1,0,-1,1,0,-1,0 };
11. void CreateMaze(int &r,int &c)
12. {
13.     ifstream infile;
14.     infile.open("maze.txt");
15.     infile >> r;
16.     infile >> c;
17.     for (int i = 0; i < r; i++)
18.         for (int j = 0; j < c; j++)
19.             infile >> maze[i][j];
20.     printf("maze:\n");
21.     for (int i = 0; i < c; i++)
22.         printf("%d ", i);
23.     printf("\n\n");
24.     for (int j = 0; j < r; j++)
25.     {
26.         printf("%d ", j);
27.         for (int k = 0; k < c; k++)
28.             printf("%d ", maze[j][k]);
29.         printf("\n");
30.     }
31.     infile.close();
32. }
33.
34. bool DFS(int r, int c, int x, int y,int step,int &length)
35. {
36.     int i, x1, y1;
37.     if (x == r && y == c )
38.     {
39.         length = step + 1; //length 记录路
                               径长度，由于 step 初值为 0，因此长度为 step+1
40.         return true;
41.     }
```

```

42.     visited[x][y] = true;                                //用 path 记录路
    径，当结点可访问时，step 增 1
43.     step++;                                              //step 要放在循
    环外，否则当遇到路口时，step 增量可能会超过 1
44.     for (i = 0; i < 4; i++)
45.     {
46.         x1 = x + direct[i][0];
47.         y1 = y + direct[i][1];
48.         if (0 <= x1 && x1 <= r && 0 <= y1 && y1 <= c)    //判断该位置是
    否超过迷宫边界
49.         {
50.             if (visited[x1][y1] != true && maze[x1][y1] != 1) //若该位置还未
    访问且不为障碍物
51.             {
52.                 path[step][0] = x1;
53.                 path[step][1] = y1;
54.                 if(DFS(r, c, x1, y1, step, length))        //当下一个位置
    可达时返回 true
55.                     return true;
56.             }
57.         }
58.     }
59.     return false;                                        //若没找到路
    径（或所有方向均到达不了）则返回 false
60. }
61.
62.
63. int main()
64. {
65.     int step = 0, length = 0;
66.     int r, c;                                           //定义行和列
67.     path[0][0] = 0;
68.     path[0][0] = 0;
69.     CreateMaze(r, c);
70.     if (DFS(r-1, c-1, 0, 0, step, length))
71.     {
72.         printf("\nDFS_Path: ");
73.         for (int i = 0; i < length; i++)
74.         {
75.             printf("(%d,%d) ", path[i][0], path[i][1]);
76.         }
77.     }
78.     else
79.         printf("Path not found");

```

```
80.     return 0;
81. }
```

(b). BFS 算法

```
1. #include<iostream>
2. #include<string>
3. #include<sstream>
4. #include<fstream>
5. #include<queue>
6. using namespace std;
7. #define MaxSize 50
8. typedef struct node                //构建迷宫的结点
9. {
10.     int x;
11.     int y;
12.     int flag;                      //flag 为 0 表示可通过
13.     bool visited;                  //判断是否访问
14.     node* pre;                    //用 pre 记录该结点的前驱结点
    (类似树的存储结构)
15. }node;
16. int direct[4][2] = { 1,0,-1,0,0,-1,0,1 };
17. node maze[MaxSize][MaxSize];
18. void CreateMaze(int& r, int& c)    //创建图
19. {
20.     ifstream infile;
21.     infile.open("maze.txt");
22.     infile >> r;
23.     infile >> c;
24.     for (int i = 0; i < r; i++)
25.         for (int j = 0; j < c; j++)
26.             {
27.                 infile >> maze[i][j].flag;
28.                 maze[i][j].x = i;
29.                 maze[i][j].y = j;
30.             }
31.     printf("maze:\n ");
32.     for (int i = 0; i < c; i++)
33.         printf("%d ", i);
34.     printf("\n\n");
35.     for (int j = 0; j < r; j++)
36.         {
37.             printf("%d ", j);
```

```

38.     for (int k = 0; k < c; k++)
39.         printf("%d ", maze[j][k].flag);
40.     printf("\n");
41. }
42.     infile.close();
43. }
44.
45. bool BFS(int r, int c, int x, int y)           //BFS 算法
46. {
47.     node front;
48.     queue<node> Queue;                          //定义队列
49.     int x1, y1;
50.     if (maze[x][y].flag != 0)
51.         return false;
52.     maze[x][y].visited = true;
53.     maze[x][y].pre = NULL;                      //将入口的前驱结点置为 NULL
54.     Queue.push(maze[x][y]);                    //将入口入队
55.     while (!Queue.empty())                     //当队不空时循环
56.     {
57.         front = Queue.front();
58.         Queue.pop();                            //将队首元素出队
59.         for (int i = 0; i < 4; i++)
60.         {
61.             x1 = front.x + direct[i][0];
62.             y1 = front.y + direct[i][1];
63.             if (0 <= x1 && x1 <= r && 0 <= y1 && y1 <= c)
64.             {
65.                 if (maze[x1][y1].visited!=true && maze[x1][y1].flag != 1)
66.                 {
67.                     maze[x1][y1].pre = &maze[front.x][front.y];
68.                     maze[x1][y1].visited = true;
69.                     if (x1 == r && y1 == c ) //到达终点时返回 true
70.                         return true;
71.                     Queue.push(maze[x1][y1]); //将新的结点入栈
72.                 }
73.             }
74.         }
75.
76.
77.     }
78.     return false;
79. }
80. void Print(node p)                             //利用迭代输出路径
81. {

```

```
82.     if (p.pre == NULL)
83.         printf("(%d,%d)", p.x, p.y);
84.     else
85.     {
86.         Print(*(p.pre));
87.         printf("(%d,%d)", p.x, p.y);
88.     }
89. }
90. int main()
91. {
92.     int r, c;
93.     CreateMaze(r, c);
94.     if (BFS(r - 1, c - 1, 0, 0))
95.     {
96.         printf("\nBFS_Path: ");
97.         Print(maze[r - 1][c - 1]);
98.     }
99.     else
100.         printf("Path not found");
101. }
```