项目文档

谈瑞

项目二：约瑟夫生者死者游戏

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 专业班级：软件工程4班  电话：18936361545 | 学号：1452775  电子邮件：tanrui106@163.com | 课目：数据结构课程设计  个人网站：http://guitoubing.top |
|  |  |  |

目录

[项目简介 3](#_Toc496957709)

[项目概要 3](#_Toc496957710)

[项目功能及要求 3](#_Toc496957711)

[项目结构 4](#_Toc496957712)

[项目类的实现 5](#_Toc496957713)

[Maze类 5](#_Toc496957714)

[Node类 6](#_Toc496957715)

[主要代码分析 7](#_Toc496957716)

[Maze.cpp 7](#_Toc496957717)

[Maze类默认构造函数 7](#_Toc496957718)

[Maze类默认析构函数 9](#_Toc496957719)

[findRoad函数 10](#_Toc496957720)

[运行测试 12](#_Toc496957721)

[几点不足 15](#_Toc496957722)

[代码不够精简 15](#_Toc496957723)

# 项目简介

## 项目概要

约瑟夫生者死者游戏的大意是：30个旅客同乘一条船，因为严重超载，加上风高浪大危险万分；因此船长告诉乘客，只有将全船一半的旅客投入海中，其余人才能幸免于难。无奈，大家只得统一这种方法，并议定30个人围成一圈，由第一个人开始，依次报数，数到第9人，便将他投入大海中，然后从他的下一个人数起，数到第9人，再将他投入大海，如此循环，直到剩下15个乘客为止。问哪些位置是将被扔下大海的位置。

## 项目功能及要求

本游戏的数学建模如下：假如N个旅客排成一个环形，依次顺序编号1, 2, …, N。从某个指定的第S号开始。沿环计数，每数到第M个人就让器出列，且从下一个人开始重新计数，继续进行下去。这个过程一直进行到剩下K个旅客为止。

本游戏要求用户输入的内容包括：

1.旅客的个数，也就是N的值；

2.离开旅客的间隔书，也就是M的值；

3.所有旅客的序号作为一组数据要求存放在某种数据结构中。

本游戏要求输出的内容是包括：

1.离开旅客的序号；

2.剩余旅客的序号。

# 项目结构

本项目采用单循环链表实现，基本上就是考察链表的插入、删除等基本操作，应该说写起来很顺畅，按照题设思路一步一步往下做就能写成功。

首先通过用户输入约瑟夫环的几个限制条件总人数、开始位置、死亡数字、剩余人数，同时判断输入是否合理，不合理则要求重新输入。而后通过这些数据构建链表完成游戏过程最后输出死亡的人以及剩余人的位置。

# 项目类的实现

## Josepth类

采用正交循环双向链表构建迷宫

private成员

public成员

|  |  |
| --- | --- |
| 类成员 | 作用 |
| Josepth (); | Josepth类的默认构造函数，此处用户输入约瑟夫环的限制条件 |
| ~Josepth (); | Josepth类的默认析构函数，在程序结束后释放链表中所有在游戏中存活下的结点 |
| void setNumbers(int, const int); | 设置限制条件，同时判断输入是否合理，若不合理则重新输入 |
| Passenger\* setJosepthNode(int); | 递归设置约瑟夫环结点，最后构建约瑟夫环 |
| Passenger\* setBeginNode(int); | 根据开始位置设置开始结点 |
| Passenger\* deleteNode(Passenger\*); | 删除结点，即杀人 |
| void startKilling(Passenger\*); | 游戏开始入口 |
| Passenger \*present, \*head, \*tail; | 头尾结点、当前结点 |
| int amount, firstIndex,  deathNumber, amountLeft; | 约瑟夫环的限制条件 |
|  |  |

# Passenger类

public成员

Passenger类存储了迷宫的结点

private成员

|  |  |
| --- | --- |
| 类成员 | 作用 |
| Passenger(); | Passenger类的默认构造函数 |
| Passenger(int, Passenger\*); | Passenger类的重载构造函数，通过参数number及next指针构造实例 |
| ~Passenger(); | Passenger类的默认析构函数 |
| int number; | 旅客的位置编号 |
| Passenger\* next; | 指向下一个旅客的指针 |
|  |  |

# 主要代码分析

## Maze.cpp

### Maze类默认构造函数

/\*

由于迷宫是从文件中读取，因此Maze类中数据不需要重载构造函数来初始化，用户要想改变迷宫形状只需要更改in\_file.txt文件即可。这里我将Maze类所有的操作函数定义为protected，因为对于用户来说只希望更改迷宫文件就能找到路径，而不需要其他的操作。这些操作函数全部在构造函数内被调用或者是间接调用。

\*/

Maze::Maze() {

head = present = NULL;

row = line = 0;

in\_file.open("in\_file.txt");

in\_file >> row >> line;

//这里的row和line是行号，而不是行数，要注意行号是比行数少1的

for (int i = 0; i <= row; i++){

//外侧循环读取列头，内侧循环从相应列头读取行

Node \*\_node = new Node();

in\_file >> \_node->data;

\_node->y\_index = i;

\_node->x\_index = 0;

if (\_node->data != '#') \_node->is\_road = true;

//如果遇到的不是"#"，则判定为该结点是路，可以添加到Maze正交链表中

present = \_node;

Node \*temp\_head1 = head;

if (i == 0) head = \_node;

else {

temp\_head1 = \_node;

Node \*temp\_head2 = head;

for (int k = 1; k < i; k++){

temp\_head2 = temp\_head2->down;

}

temp\_head2->down = temp\_head1;

temp\_head1->up = temp\_head2;

}//若是首列头结点，则直接赋值给head，否则要将其与之前的列头结点连在一起

if (i == row) {

temp\_head1->down = head;

head->up = temp\_head1;

}//首尾相连，完成首列循环

for (int j = 1; j <= line; j++){

//从列头下一个开始存储行

Node \*\_node = new Node();

in\_file >> \_node->data;

\_node->y\_index = i;

\_node->x\_index = j;

if (\_node->data != '#') \_node->is\_road = true;

if (i != 0 && j != 0) {

if (!\_node->is\_road){

//如果待插入结点不是可走的路，那要判断他是不是行尾，若是，则需要将前一个路结点与行头相连，否则直接跳过

if (j == line) {

Node \*temp\_head2 = head;

for (int k = 1; k <= i; k++) {

temp\_head2 = temp\_head2->down;

}

present->right = temp\_head2;

temp\_head2->left = present;

}

continue;

}

}

\_node->left = present;

present->right = \_node;

present = \_node;

if (j == line) {

//如果该节点已经插入，也要判断其是否为行尾，同上

Node \*temp\_head2 = head;

for (int k = 1; k <= i; k++) {

temp\_head2 = temp\_head2->down;

}

present->right = temp\_head2;

temp\_head2->left = present;

}

}

}

//上述操作仅将各行内结点、各行行头结点连在了一起，以下操作将出首列外的每一列连起来

Node \*temp\_head, \*temp\_line1 = head;

for (int i = 1; i <= line; i++){

temp\_head = head;

temp\_line1 = temp\_line1->right;

Node \*temp\_line2 = temp\_line1;

for (int j = 1; j <= row+1; j++){

temp\_head = temp\_head->down;

Node \*find\_node = findNodeInRow(i, temp\_head);

if (j == row + 1) {

temp\_line2->down = temp\_line1;

temp\_line1->up = temp\_line2;

continue;

}

if (find\_node != NULL) {

temp\_line2->down = find\_node;

find\_node->up = temp\_line2;

temp\_line2 = find\_node;

}

}

}

printMaze();

justGo();

}

### Maze类默认析构函数

/\*

由于Maze采用的是正交链表，因此析构时需要将每个结点空间全部释放，这是一种递归释放，但是这里把递归解开成非递归循环，逐行释放空间知道链表为空。

\*/

Maze::~Maze(){

if (head == NULL){

//空迷宫直接返回，无需析构

return;

}

//非空迷宫需从头结点开始遍历每一个元素，使得所有结点指针均被释放，最后留下头结点，删除之

Node \*temp\_head = head, \*temp\_line = head;

for (int i = 0; i <= row; i++){

for (int j = 0; j <= line; j++){

//这里删除采用和构造时一样的顺序，先找列头，而后从列头往后删除结点，直到所删除结点的right结点是列头，即停止，并删除列头

if (temp\_line->right == head) {

//此处判断是否到达列尾

delete temp\_line;

if (head->data != '#'){

break;

}

temp\_head = head;

head = head->down;

delete temp\_head;

j = line + 1;

}

else{

temp\_head = temp\_head->right;

temp\_line = temp\_head->right;

if (temp\_head == temp\_line){

break;

}

delete temp\_head;

temp\_head = temp\_line;

if (temp\_line == head) {

temp\_head = head;

head = head->down;

delete temp\_head;

j = line + 1;

}

}

}

temp\_head = head;

}

head = NULL;

//最后释放头结点，Maze为空

}

### findRoad函数

void Maze::findRoad(bool \_right, bool \_down, bool \_left, bool \_up, Node \*\_present) {

road\_vec.push\_back(\_present);

if (\_present->x\_index == line - 1 && \_present->y\_index == row - 1) {

isEnd = true;

return;

}//找到了终点就返回

if (\_present->x\_index == 1) \_left = false;

if (\_present->x\_index == line-1) \_right = false;

if (\_present->y\_index == 1) \_up = false;

if (\_present->y\_index == row-1) \_down = false;

//靠边界的结点就不再找边界结点了

if (\_right && \_present->right->x\_index == \_present->x\_index+1 && !isEnd) {

findRoad(true, true, false, true, \_present->right);

}

if (\_down && \_present->down->y\_index == \_present->y\_index+1 && !isEnd){

findRoad(true, true, true, false, \_present->down);

}

if (\_left && \_present->left->x\_index == \_present->x\_index - 1 && !isEnd) {

findRoad(false, true, true, true, \_present->left);

}

if (\_up && \_present->up->y\_index == \_present->y\_index - 1 && !isEnd) {

findRoad(true, false, true, true, \_present->up);

}

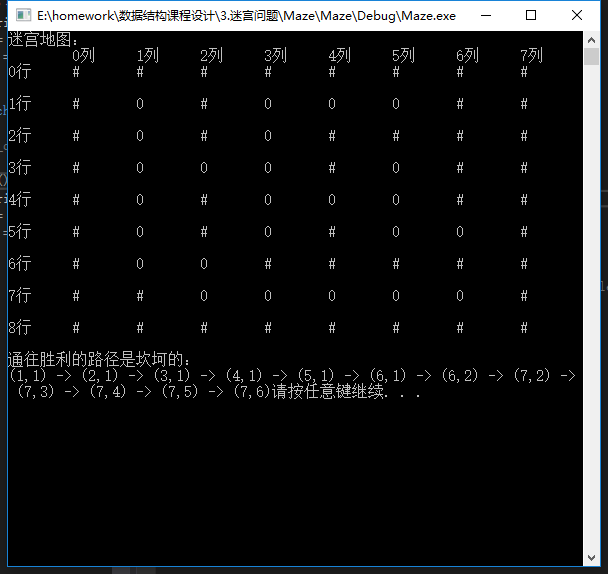
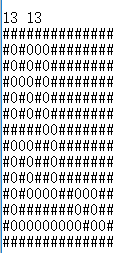
if (!isEnd) {

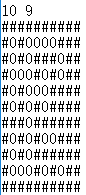
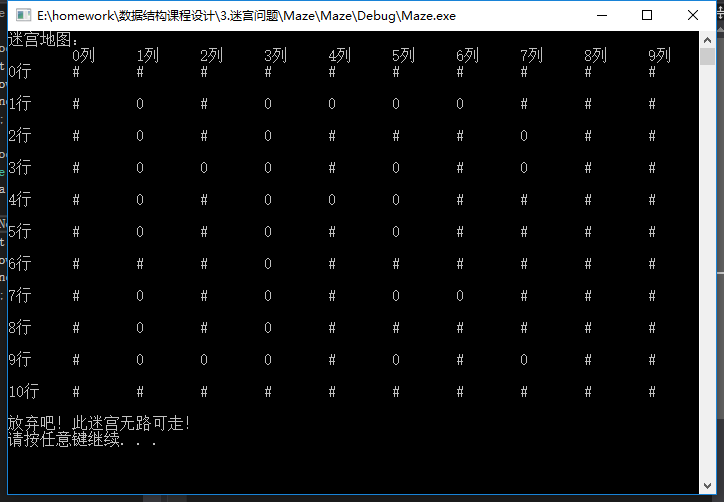
road\_vec.pop\_back();

}//按照右、下、左、上的顺序搜索下一个路结点，如果没有路了，那同时该结点必然不是终点，则弹出该结点

}

# 运行测试





# 几点不足

## 代码不够精简

由于是第一次使用正交双向循环链表，对其操作还是不够熟悉，因此有很多冗余代码，但是又不好删，只得留下保证程序基本运行。