项目文档

谈瑞

项目七：牧场栅栏问题

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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 |
|  |  |  |

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

农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要N块木头，每块木头长度为整数Li个长度单位，于是他购买了一个很长的，能锯成N块的木头，即该木头的长度是Li的总和。

但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨就设酬金等于所锯木头的长度。例如，要将长度为20的木头锯成长度为8，7和5的三段，第一次锯木头将木头锯成12和8，花费20；第二次锯木头将长度为12的木头锯成7和5花费12，总花费32元。如果第一次将木头锯成15和5，则第二次将木头锯成7和8，那么总的花费是35（大于32）.

## 项目功能及要求

此问题的数学建模如下，即对于一个正整数数列，每次选取2个数相加，将和添加到数列中去，然后删除这2个数，如此循环知道数列中最后剩下一个数。而此题就要求出使得最后一个数最小的情况。很容易想到需要构建Huffman树来实现此功能。

具体做法是：先将输入的N（N<=104）个整数排序，使其按照从小到大的顺序排列（目的是使数组满足Huffman树的基本条件），实际上，构成Huffman树的元素并不是输入的整数数组，而应是锯成N块木头的前一个状态，此时的ceil（N/2）（ceil指向上取整）个元素才是Huffman树的真正的叶子结点。要得到这样的叶子结点，即是将N个整数两两相加（这里要注意N若是奇数则会多出来一个数，应对措施是先从总价钱cost中减去这个数，然后将其视作叶子结点，将其按照大小放置于正确位置即对ceil（N/2）个整数再排序），此时用得到的ceil（N/2）个整数构建Huffman树，而后遍历Huffman树，每遍历一个结点将其中的长度作为价格加到cost上，遍历完成即得到最终的花费，而此花费也正是各个叶子结点的权值与其到根结点的路径乘积的和，某结点到根结点的路径长度即可理解为农夫从整块木头到第一次砍出来这一块木头的次数。

此项目支持控制台输入及文件输入，初始文档中已经有1000个随机数，初测能得出正确花费。

# 项目结构

本项目采用Huffman树结构，依照Huffman树的“带权路径长度最小的二叉树应是权值大的外结点离根结点最近的扩充二叉树特性”，将木头长度视作结点权值，构建Huffman树，而后遍历树得到结果。

此题特殊点在于，构建Huffman树的基础数组并非用户输入的数组，具体原因在于，Huffman树在求解带权路径长度时是将叶子结点的权值乘以其到根结点的路径，若采用输入的整数数组作为基数组，则会求解时会多加一遍这个数组，当然若这样做之后将总结果减去基数组的总和也是不可取的，应该使最后存储的结构在去掉基数组中的木头后成为一个标准的Huffman树，才能得到正确的结果。

当然，为何排了序后的输入数组两两相加就能得到想要的正确的基数组呢，这是数学证明的范畴了，我这里也是假设以及通过对比验证默认认为此种方法得出的基数组而生成的Huffman树在所有可能的Huffman树中的总花费最小。

# 项目类的实现

## Woods类

采用Huffman树数据结构

private成员

public成员

|  |  |
| --- | --- |
| 类成员 | 作用 |
| struct Wood | Woods类的结点，包含当前结点（即木头）的长度，以及左右子节点 |
| static long cost = 0; | 砍出所需木头的总花费，设置为静态变量，非类成员 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 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);

}

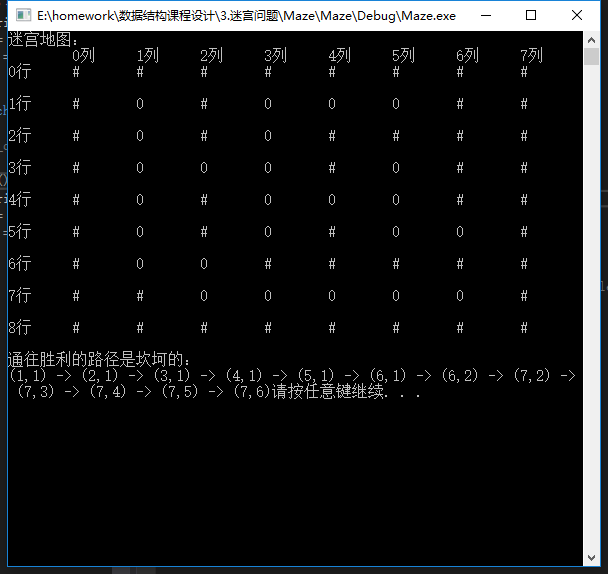
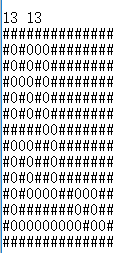
if (!isEnd) {

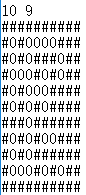
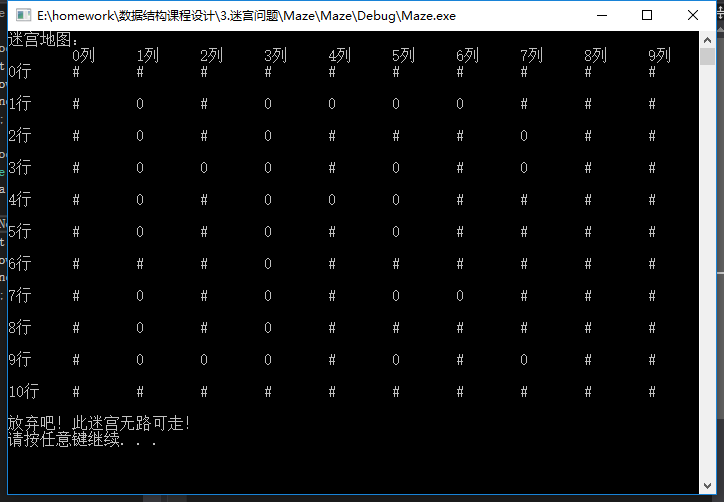
road\_vec.pop\_back();

}//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。