武汉纺织大学

数据结构课程设计

**迷宫 & 停车场管理 & 哈夫曼编码**

**学 院： 计算机与人工智能学院**

**班 级： 计科12002班**

**姓 名： 刘逸风**

**学 号： 2004240423**

**指导老师：**

**成 绩：**

**完成日期： 2021年12月20日**

目 录

[1. 需求分析 3](#_Toc95471722)

[1.1问题描述 3](#_Toc95471723)

[1.2基本要求 3](#_Toc95471724)

[1.2.1输入的形式和输入值的范围 3](#_Toc95471725)

[1.2.2输出的形式 4](#_Toc95471726)

[1.2.3程序所能达到的功能 4](#_Toc95471727)

[2. 概要设计 5](#_Toc95471728)

[2.1数据结构 5](#_Toc95471729)

[2.2程序模块 5](#_Toc95471730)

[2.3各模块之间的调用关系以及算法设计 6](#_Toc95471731)

[3. 详细设计 8](#_Toc95471732)

[4. 测试与分析 28](#_Toc95471733)

[5. 总结 34](#_Toc95471734)

[6. 附录 35](#_Toc95471735)

# 需求分析

## 1.1问题描述

* 迷宫
  + 以一个m\*n的长方阵表示迷宫，0和1分别表示迷宫中的通路和障碍。设计一个程序，对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。
* 停车场管理
  + 停车场内只有一个可停放n辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序，依次由北向南排列（大门在最南端，最先到达的第一辆车停放在停车场的最北端），若车场内已停满n辆汽车，则后来的汽车只能在门外的便道上等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内某辆车要离开时，在它之后开入的车辆必须先退出车场为它让路，待该辆车开出大门外，其它车辆再按原次序进入车场，每辆停放在车场的车在它离开停车场时必须按它停留的时间长短交纳费用。试为停车场编制按上述要求进行管理的模拟程序。
* 哈夫曼编码
  + 利用哈夫曼编码进行通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码（解码）。对于双工信道（即可以双向传输信息的信道），每端都需要一个完整的编/译码系统。试为这样的信息收发站设计一个哈夫曼编译码系统。

## 1.2基本要求

### 1.2.1输入的形式和输入值的范围

* 迷宫
  + 输入形式：以文件的方式读入m行n列的长方阵表示迷宫 ，入口或出口的坐标索引从1开始
  + 输入值的范围：迷宫的长和宽都要大于2，且迷宫只由数字0和数字1构成，0为通路，1为障碍。
* 停车场管理
  + 输入形式：从终端输入，每一组输入数据包括三个数据项：汽车“到达”或“离去”信息，车牌号码以及“到达”或“离去”的时刻。时间信息的输入形式为: xxxx-xx-xx-xx-xx，采用24小时进制，精确到分钟
  + 输入值的范围：第一个数据项只能为“到达”或“离去”，车牌号码为任意字符串，时间信息要求年份要大于1900年，月份范围为1-12，天数范围为1-31，小时范围为0-23，分钟范围为0-59。
* 哈夫曼编码
  + 输入形式：以文件方式输入，构建哈夫曼树的数据文件为DataFile.data， 需要编码的文件为ToBeTran.data，需要译码的文件为CodeFile.data。
  + 输入值的范围：只能包括ASCII表的前127个字符，但不能包括‘@’字符

### 1.2.2输出的形式

* 迷宫
  + 以三元组（i，j，k）的形式输出，其中（i，j）表示坐标，k表示方向，下上右左分别用0，1，2，3表示
  + 以方阵的形式输出迷宫以及所有的通路，其中障碍物‘1’以黄色背景填充，通路的前进方向“上下左右”分别用U，D，L，R来代替，并将通路标红。
* 停车场管理
  + 若输入为到达，则输出汽车的车牌号以及汽车在停车场或便道的停车位置上。
  + 若输入为离去，则输出汽车的车牌号以及汽车在停车场停留的时间以及需要缴纳的费用(默认5元/小时)。若在便道上则不收费，可直接离去
* 哈夫曼编码
  + - 1. 显示DataFile.data中出现的字符以及出现频度(次数)
      2. 显示构建的哈夫曼树的层次遍历结果
      3. 显示字符编码对照表
      4. 输出ToBeTran.data及其报文Code.txt
      5. 输出CodeFile.data及其TextFile.txt

### 1.2.3程序所能达到的功能

* 迷宫: 满足所有要求
* 停车场管理：若为输入到达，只能输出汽车在停车场的位置，没有对便道进行编号
* 哈夫曼编码：满足所有的要求

# 概要设计

## 2.1数据结构

* 迷宫：栈，图
* 停车场管理：栈和队列
* 哈夫曼编码：二叉树和优先队列和图

## 2.2程序模块

* 迷宫：

1. Data：迷宫问题测试数据.txt——存储着m\*n的迷宫数据。
2. Lib：Stack\_2.h——用链表创建的链式栈。
3. Src：maze.cpp——源码文件，提供递归和非递归两种解谜宫的方法，分别用三元组，方阵来输出结果。

* 停车场管理：

1. Lib：
   1. Car.h——定义了汽车的结构体，且提供了读入数据和计算费用的功能。
   2. Queue.h——用链表创建的队列，重载了[]运算符方便查找，提供了erase函数，可以删除第一个指定值的元素。
   3. Stack\_1.h——用数组创建的顺式栈，重载了[]运算符方便查找。
2. Src：administration.cpp——源码文件，根据终端输入输出对应的结果。

* 哈夫曼编码：

1. Data：
   1. Code.txt——存储ToBeTran.data的编码结果。
   2. CodeFile.data——需要译码的文件。
   3. DataFile.data——构建哈夫曼树所需要的数据。
   4. TextFile.txt——存储CodeFile.data的译码结果。
   5. ToBeTran.data——需要编码的文件。
2. Lib：
   1. node.h——定义了哈夫曼树的节点信息的类，提供了三种构造函数，重载了‘>’比较运算符，方便排序。
   2. tree.h——用于构建哈夫曼树，含有存储DataFile.dada的记录结果的优先队列，存储哈夫曼树层次遍历结果的二维向量以及哈夫曼字符编码对照。分别提供了优先队列，层次遍历结果，字符编码对照表的输出函数，同时提供了构建哈夫曼树的函数（返回哈夫曼树的根节点）。
3. Src：huffman.cpp——源码文件，提供“初始化”，“编码”，“译码”，“输出”四个功能模块。

## 2.3各模块之间的调用关系以及算法设计

* 迷宫

1. 调用关系
   1. maze.cpp的get\_data函数可以获取“迷宫问题测试数据.txt”的数据以保存在以二维向量martix中，Stack\_2.h提供的栈则用于采用非递归的方式以三元组输出结果
2. 算法设计
   1. 递归算法采用深度优先搜索，从起点开始，向4个方向深度搜索，如果遇到障碍或超出边界就回溯到前一个位置从下一个方向重新搜索，将路径坐标保存在一维向量soulution中，最后通过保存的路径坐标计算出每一次的方向数据，以方针形式输出即可
   2. 非递归算法采用数据结构栈来模拟，先将起点压入栈中，当栈不为空则进入循环，弹出栈顶元素，如果栈顶元素是终点就以三元组的形式输出结果，若不是终点，则把四个方向的合法坐标压入栈中，同时使用图pre记录当前坐标和前驱坐标。循环结束后，从终点开始找前驱坐标直到起点，记录路径，最后把路径反转后就是从起点到终点的路径了。

* 停车场管理

1. 调用关系
   1. car.h为administration.cpp提供了car类，并提供了数据输入函数，重载了‘()’运算符来计算费用，Stack\_1.h提供栈来模拟停车场，Queue.h提供队列来模拟便道。
2. 算法设计
   1. 关于停留时间的计算采用time.h库提供的mktime函数和difftime函数。
   2. 在有汽车到达时，若栈的容量没到指定的最大值，则将汽车入栈，若栈的容量以及到了最大值，则将汽车入队；有汽车离去时，若汽车在便道上，则将汽车从队列里删除，若汽车在栈中，先将该汽车后进栈的元素全部弹出，保存在一个新队列中，将该汽车弹出后，再将新队列中的元素依次入栈，此时判断便道是否为空，若不为空，则便道队列队头元素出队进栈并开始计时。

* 哈夫曼编码

1. 调用关系
   1. DataFile.data，ToBeTran.data，CodeFile.data分别提供数据用于构建哈夫曼树，编码，译码。Code.txt，TextFile.txt分别用于存储编码和译码结果。
   2. node.h定义了了节点类HTNode，并提供了三种构造方法，且重载了比较运算符’>’便于之后的排序。
   3. tree.h含有存储DataFile.dada的记录结果的优先队列，存储哈夫曼树层次遍历结果的二维向量以及哈夫曼字符编码对照。分别提供了优先队列，层次遍历结果，字符编码对照表的输出函数，同时提供了构建哈夫曼树的函数（返回哈夫曼树的根节点）。
2. 算法设计
   1. 采用优先队列存储树的节点，比较方式为节点字符的频度，遍历数据文件的每一个字符(包括空格)，以哈希的方式记录次数，然后遍历哈希表，若哈希表的这一项不为0，我们调用节点的构造函数构造一个节点，将此节点入队即可。
   2. 创建哈夫曼树时，我们利用存储节点的优先队列，若优先队列内的元素大于2，我们取出最小的两个元素出队，再调用节点的构造函数生成一个中间节点，使其的左右孩子为这两个节点，将生成的中间节点入队，循环这个过程直到最后队列中只有一个节点，次节点即为构建的哈夫曼树的根节点。
   3. 层次遍历哈夫曼树时，我们创建一个队列，将根节点入队，当队列不为空的时候进行循环，每次循环记录当前层的节点数量，依次将当前层的左右孩子入队，把每一层的结果用vector来保存，最终结果用二维向量来表示。
   4. 生成字符编码对照表时，我们采用递归的算法，每次把路径编码（左0右1）添加到该节点到根节点的路径编码，如果该节点是中间节点，则递归其左孩子和右孩子，若为叶子节点则将结果记录到字符编码对照表中。
   5. 编码时，将需要编码的字符通过编码表对应的编码进行替换即可
   6. 译码时，遍历编码，按左0右1的遍历方式从哈夫曼树的根节点往根节点遍历，如果遍历到根节点就记录这个字符，在重新返回根节点，以同样的方式遍历下一串编码

# 详细设计

* 迷宫

1. Stack\_2.h

//链式栈类

#include<iostream>

#include<list>

using namespace std;

template<typename T>

class stack {

public:

    stack() {

    }

    stack(const stack& s) {

        this->l = s.l;

    }

    void push(T a) {

        l.emplace\_back(a);

    }

    void pop() {

        l.pop\_back();

    }

    bool empty() {

        return l.empty();

    }

    T top() {

        return l.back();

    }

private:

    list<T> l;

};

1. maze.cpp

//迷宫问题

#include <iostream>

#include <cstring>

#include <vector>

#include <fstream>

#include <map>

#include <algorithm>

#include "../lib/Stack\_2.h" //链式栈

#include <windows.h>    //改变路径的显示颜色

using namespace std;

int dir[4][2]={{1,0},{-1,0},{0,1},{0,-1}}; //方向数组 ,分别为下，上，右，左 ，输出分别设为0, 1, 2, 3

vector<vector<int>> matrix; //迷宫矩阵

vector<vector<int>> visited;    //访问记录(0:未访问, 1:已访问)

vector<pair<int, int>> solution;    //记录节点结果

vector<vector<string>> run; //记录方向的输出结果

map<pair<int, int>, pair<int, int>> pre;    //记录前驱节点

int m, n;   //矩阵的行和列

int start\_x, start\_y, end\_x, end\_y; //起点坐标和终点坐标

int flag = 0;   //是否有通路

void init\_run() {   //初始化结果列表

    run = vector<vector<string>>(m, vector<string>(n, " "));

    for(int i = 0; i < m; i++)

        for(int j = 0; j < n; j++)

            run[i][j] = matrix[i][j] + '0';

}

void get\_data() {

    ifstream in("../data/迷宫问题测试数据.txt");    //读入测试数据

    string s;

    while(getline(in, s)) {

        vector<int> t;

        for(char c : s) {

            if(c != ' ')

                t.emplace\_back(c - '0');

        }

        matrix.emplace\_back(t);

    }

    in.close();

    m = matrix.size();      //行

    n = matrix[0].size();   //列

    visited = vector<vector<int>>(m, vector<int>(n));

    init\_run();

}

bool check(int x, int y) {  //检查该坐标点是否有障碍或已经被访问过

    if(x < 0 || x >= m || y < 0 || y >= n)

        return false;

    else

        return matrix[x][y] == 0 && visited[x][y] == 0;

}

int direction(pair<int, int>& a, pair<int, int>& b) {   //返回a -> b的方向参数

    for(int i = 0; i < 4; i++) {

        if(b.first == a.first + dir[i][0] && b.second == a.second + dir[i][1])

            return i;

    }

    return  -1;

}

void ouput1() {     //以三元组形式输出结果

    pair<int, int> p = {end\_x, end\_y};

    while(!(p.first == start\_x && p.second == start\_y)) {

        solution.push\_back({p.first, p.second});

        p = pre[{p.first, p.second}];

    }

    solution.push\_back({start\_x, start\_y});

    reverse(solution.begin(), solution.end());

    for(int i = 0; i < solution.size() - 1; i++) {

        cout << '(' << solution[i].first + 1 << ", " << solution[i].second + 1;

        int d = direction(solution[i], solution[i + 1]);

        cout << ", " << d << ')' << endl;

    }

    cout << '(' << end\_x + 1 << ", " << end\_y + 1 << ", 终)" << endl;

}

void solve1() {     //问题一，非递归

    init\_run();

    visited = vector<vector<int>>(m, vector<int>(n));

    solution.clear();

    stack<pair<int, int>> s;    //链栈

    s.push({start\_x, start\_y}); //将起点压栈

    visited[start\_x][start\_y] = 1;

    while(!s.empty()) {

        pair<int, int> a = s.top();

        s.pop();

        if(a.first == end\_x && a.second == end\_y) {

            ouput1();

            return;

        }

        for(int i = 0; i < 4; i++) {

            int x = a.first + dir[i][0];

            int y = a.second + dir[i][1];

            if(!check(x, y)) {

                continue;

            }

            visited[x][y] = 1;

            pre[{x, y}] = a;

            s.push({x, y});

        }

    }

    cout << "无通路" << endl;

}

void output2() {    //以方阵的形式输出所有通路结果

    for(int i = 0; i < solution.size() - 1; i++) {

        int x = solution[i].first;

        int y = solution[i].second;

        int d = direction(solution[i], solution[i + 1]);

        if(d==0) run[x][y] = "D";

        if(d==1) run[x][y] = "U";

        if(d==2) run[x][y] = "R";

        if(d==3) run[x][y] = "L";

    }

    flag = 1;

    run[end\_x][end\_y] = "终";

    HANDLE hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);   //获取标准输出的句柄

    for(auto i : run) {

        for(auto j : i) {

            if(j == "D" || j == "U" || j == "R" || j == "L") {

                SetConsoleTextAttribute(hStdout, 0x0c); //改为红色

                cout << j << ' ';

            }

            else if(j == "终") {

                SetConsoleTextAttribute(hStdout, 0x0c); //改为红色

                cout << j << ' ';

                SetConsoleTextAttribute(hStdout, 0x0F); //恢复白色

            }

            else if(j == "1") {

                SetConsoleTextAttribute(hStdout, 0x60); //改为黄色背景

                cout << j << ' ';

                SetConsoleTextAttribute(hStdout, 0x0F); //恢复白色

            }

            else {

                SetConsoleTextAttribute(hStdout, 0x0F); //恢复白色

                cout << j << ' ';

            }

        }

        cout << endl;

    }

    cout << endl;

    init\_run();     //每次输出结果后初始化结果列表

}

void dfs(int x, int y) {    //dfs求所有通路, (x, y)为当前起点坐标

    solution.push\_back({x, y});

    visited[x][y] = 1;

    if(x == end\_x && y == end\_y) {  //如果到达终点

        output2();

    }

    else {

        for(int i = 0; i < 4; i++) {    //四个方向依次遍历

            int xx = x + dir[i][0]; //下个方向的横坐标

            int yy = y + dir[i][1]; //下个方向的纵坐标

            if(check(xx,yy)) {

                dfs(xx, yy);

            }

            else    //下一个点不可行

                continue;

        }

    }

    visited[x][y] = 0;  //回溯

    solution.pop\_back();    //回溯

}

void solve2() { //递归

    init\_run();

    visited = vector<vector<int>>(m, vector<int>(n));

    solution.clear();

    flag = 0;

    dfs(start\_x, start\_y);

    if(!flag) {

        cout << "无通路" << endl;

    }

}

int main()

{

    get\_data();

    cout << "请输入起点坐标: " << endl;

    cin >> start\_x >> start\_y;

    start\_x--;  start\_y--;  //转换为下标索引

    cout << "请输入终点坐标: " << endl;

    cin >> end\_x >> end\_y;

    end\_x--;    end\_y--;    //转换为下标索引

    int choose;

    while(true) {

        cout << "请选择任务:\n" << endl;

        cout << "1: 使用非递归方法(栈),以三元组的形式输出一条通路" << endl;

        cout << "2: 使用递归方法(dfs),求出所有通路以方阵的形式输出通路" << endl;

        cout << "0: 结束程序" << endl;

        cin >> choose;

        if(choose == 1)

            solve1();

        else if(choose == 2)

            solve2();

        else if(choose == 0)

            return 0;

        else

            cout << "输入错误" << endl;

    }

    return 0;

}

* 停车场

1. Car.h

//停车场管理

#include<iostream>

#include<cstdio>

#include<string>

#include<cmath>

#include<ctime> //计算时间差

using namespace std;

const double price = 5; //5元每小时

struct car {    //定义结构体存储汽车信息

    tm arrive\_time; //到达时间(格式xxxx-xx-xx-xx-xx)(24进制)精确到分钟

    tm leave\_time;  //离去时间(格式xxxx-xx-xx-xx-xx)(24进制)精确到分钟

    string ID;      //车牌号

    int number;     //停车位编号(从1开始取)

    int pos;        //为0表示在停车场，为1表示在便道

    double operator()(){    //返回停留时间(单位: 秒)

        time\_t time\_1 = mktime(&arrive\_time);

        time\_t time\_2 = mktime(&leave\_time);

        return difftime(time\_2, time\_1);

    }

    bool operator == (car& c){

        if(this->ID == c.ID)

            return true;

        else

            return false;

    }

};

tm add\_time() {

    tm info = {0};

    cout << "请输入时间信息: (格式xxxx-xx-xx-xx-xx)(24进制)精确到分钟: " << endl;

    scanf("%d-%d-%d-%d-%d",&info.tm\_year, &info.tm\_mon, &info.tm\_mday, &info.tm\_hour, &info.tm\_min);

    info.tm\_min -= 1;   info.tm\_hour -= 1;  info.tm\_mon -= 1;   //标准化

    info.tm\_year -= 1900;   //标准化

    return info;

}

int add\_data(car& c) {  //返回0为到达，返回1为离去

    string information; cin >> information;

    cout << "请输入车牌号: " << endl;

    cin >> c.ID;

    if(information == "到达")

        c.arrive\_time = add\_time();

    else

        c.leave\_time = add\_time();

    return information == "离去";

}

double need\_money(car& c) {

    return price \* ceil(c() / 3600.0);

}

1. Queue.h

//链式队列类

#include<iostream>

#include<list>

using namespace std;

template <typename T>   //T为数据类型

class queue {

public:

    queue() {   //默认构造函数

    }

    queue (const queue& q) {    //拷贝构造函数

        this->l = q.l;

    }

    void push(T a) {    //入队

        l.emplace\_back(a);

    }

    void pop() {        //出队

        l.pop\_front();

    }

    bool empty() {      //是否为空

        return l.empty();

    }

    T front() {         //队头元素

        return l.front();

    }

    T back() {          //队尾元素

        return l.back();

    }

    int size() {

        return l.size();

    }

    T operator[](int n) {

        auto i = l.begin();

        while(n--) i++;

        return \*i;

    }

    void erase(T& t) {

        for(auto i = l.begin(); i != l.end(); i++) {

            if(\*i == t) {

                l.erase(i);

                return;

            }

        }

    }

private:

    list<T> l;

};

1. Stack\_1.h

//顺式栈类

#include<iostream>

#include<vector>

using namespace std;

template<typename T>

class stack {

public:

    stack() {

    }

    stack(const stack& s) {

        this->v = s.v;

    }

    void push(T a) {

        v.emplace\_back(a);

    }

    void pop() {

        v.pop\_back();

    }

    bool empty() {

        return v.empty();

    }

    T top() {

        return v.back();

    }

    int size() {

        return v.size();

    }

    T operator[] (int n) {

        return v[n];

    }

private:

    vector<T> v;

};

1. Administration.cpp

//停车场管理

#include <iostream>

#include "../lib/Car.h"         //定义车辆结构体

#include <string>

#include "../lib/Stack\_1.h"     //顺式栈(重载了[]运算符，方便查找)

#include "../lib/Queue.h"       //链式队列(重载了[]运算符，方便查找)

using namespace std;

int n;

stack<car> st;

queue<car> que;

void arrive(car& c) {   //到达操作

    if(st.size() < n) { //停车场没满

        c.number = st.size() + 1;   //编号

        c.pos = 0;  //位置在停车场

        st.push(c);

    }

    else {  //停车场满了

        c.pos = 1;  //位置在便道

        que.push(c);

    }

}

void leave(car& c) {

    for(int i = 0; i < st.size(); i++) {    //找停车场

        if(st[i].ID == c.ID) {

            tm tmp = c.leave\_time;

            c = st[i];  c.leave\_time = tmp;

        }

    }

    for(int i = 0; i < que.size(); i++) {   //找便道

        if(que[i].ID == c.ID) {

            tm tmp = c.leave\_time;

            c = que[i]; c.leave\_time = tmp;

        }

    }

    if(c.pos) { //如果在便道上, 直接离开即可

        que.erase(c);

        cout << "该车(" << "车牌号为: " << c.ID <<  ")离开便道, 不收费" << endl;

    }

    else {  //在停车场

        if(c.number == st.size()) { //停车场最后面

            st.pop();

            cout << "该车("<< "车牌号为: " << c.ID <<")在停车场停留" << ceil(c()/60.0) << "分钟" << endl;

            cout << "应缴纳" << need\_money(c) << "元" << endl;

        }

        else {  //在停车场中间

            queue<car> t;   //辅助队列

            while(st.size() != c.number) {  //后面的车让路

                car x = st.top();

                x.number -= 1; //后面的车依次前进一个车位

                st.pop();   t.push(x);

            }

            st.pop();

            cout << "该车("<< "车牌号为: " << c.ID <<")在停车场停留" << ceil(c()/60.0) << "分钟" << endl;

            cout << "应缴纳" << need\_money(c) << "元" << endl;

            while(!t.empty()) { //让路的车重新进入停车场

                car x = t.front();

                st.push(x); t.pop();

            }

            if(st.size() < n && !que.empty()) { //如果停车场有空位并且便道有车等待

                car x = que.front();

                que.pop();

                x.arrive\_time = c.leave\_time;   //开始计算收费

                x.number = st.size() + 1;       //车位编号

                x.pos = 0;                      //进入停车场

                st.push(x);

            }

        }

    }

}

int main()

{

    cout << "输入狭长通道最大容量: " << endl;

    cin >> n;

    car c;

    while(true) {

        cout << "\n请输入操作(到达或离去)" << endl;

        int flag = add\_data(c);

        if(!flag) { //到达

            arrive(c);

            if(!c.pos)

                cout << "汽车("<< "车牌号为: " << c.ID <<")在停车场的位置为: " << c.number << endl;

            else

                cout << "汽车("<< "车牌号为: " << c.ID <<")停在便道"<< endl;

        }

        else {  //离开

            leave(c);

        }

    }

    return 0;

}

* 哈夫曼编码

1. node.h

//定义节点

#include <iostream>

using namespace std;

class HTNode {

public:

    char ch;        //字符

    int weight;     //权重

    HTNode\* left;   //左孩子

    HTNode\* right;  //右孩子

    //三个构造函数

    HTNode(int w) : weight(w) {}

    HTNode(char c, int w) : ch(c), weight(w), left(nullptr), right(nullptr) {}

    HTNode(char c, int w, HTNode\* l, HTNode\* r) : ch(c), weight(w), left(l), right(r) {}

    bool operator > (const HTNode& node) const {    //重载比较运算符方便排序

        return weight > node.weight;

    }

};

1. tree.h

//完成哈夫曼树的构建

#include "node.h"   //定义节点的头文件

#include <map>      //用map保存编码对照表

#include <fstream>  //读写文件

#include <queue>    //需要使用优先队列

#define prior\_que priority\_queue<HTNode, vector<HTNode>, greater<HTNode>>   //宏定义一下优先队列类型, 大顶堆, 小的数据先出队

using namespace std;

int a[128] = {0};           //哈希法处理ASCII字符

//通过字符串统计字符的频率, 返回记录结果的优先队列

prior\_que count(string PATH) {

    ifstream in(PATH);

    string t;

    prior\_que ans;

    while (getline(in, t)) {

        for (int i = 0; i < t.size(); i++)  {

            try {   //异常处理

                if(t[i] < 0 || t[i] >= 128) {

                    string y;

                    int x = i;

                    y += t[i];

                    throw x;

                    throw y;

                }

                else

                    a[t[i]]++;

            }

            catch(const int e) {

                cerr << e << endl;

            }

            catch(const string e) {

                cerr << e << endl;

            }

        }

    }

    for(int i = 0; i < 128; i++) {

        if(a[i]) {  //如果字符存在

            HTNode \* node = new HTNode(char(i), a[i]);

            ans.push(\*node);

            delete node;

        }

    }

    in.close();

    return ans;

}

prior\_que statistic\_queue;  //记录字符及其频度的优先队列

/\*\*

 \* 创建哈夫曼树

 \* @param 无

 \* @return 哈夫曼树的根节点

\*/

HTNode Create\_Tree() {

    prior\_que que = statistic\_queue;

    while(que.size() >= 2) {

        HTNode\* left = new HTNode(que.top());   que.pop();

        HTNode\* right = new HTNode(que.top());  que.pop();

        HTNode parent('@', left->weight + right->weight, left, right);  //@字符用于代替中间的节点字符

        que.push(parent);

    }

    return que.top();

}

/\*\*

 \* 输出字符及其频度

 \* @param que 存储字符及其频度的优先队列

\*/

void show\_que(prior\_que que) {

    while(!que.empty()) {

        HTNode node = que.top();

        que.pop();

        cout << node.ch << " : " << node.weight << endl;

    }

}

/\*\*

 \* 层次遍历哈夫曼树

 \* @param root 哈夫曼树的根节点

 \* @return 一个vector<vector<pair<char, int>>>类型的结果

\*/

vector<vector<pair<char, int>>> levelorder(HTNode\* root) {  //层次遍历哈夫曼树

    queue<HTNode\*> que;

    if(root != nullptr)

        que.push(root);

    vector<vector<pair<char, int>>> ans;

    while(!que.empty()) {

        int size = que.size();

        vector<pair<char, int>> vec;    //记录每一层的结果

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

            HTNode \* node = que.front();

            que.pop();

            vec.push\_back({node->ch, node->weight});

            if(node->left)      //左孩子入队

                que.push(node->left);

            if (node->right)    //右孩子入队

                que.push(node->right);

        }

        ans.push\_back(vec);

    }

    return ans;

}

vector<vector<pair<char, int>>> order\_res;  //哈夫曼树的层次遍历结果

void show\_vec(vector<vector<pair<char, int>>> res) {

    for(auto i : res) {

        for(auto j : i) {

            cout << "{" <<  j.first << " : " << j.second << "}" << " ";

        }

        cout << endl;

    }

}

map<char, string> code\_map;     //编码对照表

/\*\*

 \* 获取编码表，使用map存入结果: map<char, string> code\_map;

 \* @param root 哈夫曼树的根节点

 \* @param code 路径编码, 左0右1

 \* @param path 到根节点的路径

\*/

void get\_CodeMap(HTNode \* root, string code, string& path) {

    string s = path;

    s += code;

    if(root != nullptr) {

        if(root->ch == '@') {   //是中间节点

            get\_CodeMap(root->left, "0", s);    //左0

            get\_CodeMap(root->right, "1", s);   //右1

        }

        else {  //编码节点

            code\_map[root->ch] = s; //记录结果

        }

    }

}

void show\_map(map<char, string> Code\_Map)

{

    for(auto i : Code\_Map) {

        cout << i.first << "\t:\t" << i.second << endl;

    }

}

1. Huffman.cpp

//哈夫曼编码与译码

#include "../lib/tree.h"        //生成哈夫曼树的头文件

#include <windows.h>    //进度条改色用

using namespace std;

HTNode root(0); //哈夫曼树的根节点

//从数据文件中读入字符及每个字符的权值，建立哈夫曼树HuffTree

void Initialzation(string PATH) {

    statistic\_queue = count(PATH);  //存储字符频度的优先队列

    root = Create\_Tree();

    order\_res = levelorder(&root);

    string t = "";

    get\_CodeMap(&root, "", t);      //生成编码对照表code\_map

}

//用已建好的哈夫曼树，对from文件中的文本进行编码形成报文，将报文写在文件to中

void Encode(string from, string to) {

    ifstream in(from);

    ofstream out(to, ios::app);

    string data = "";   //记录每一行的明文

    while(getline(in, data)) {

        string t = "";  //记录每一行的编码结果

        for(int i = 0; i < data.size(); i++) {

            t += code\_map[data[i]];

        }

        out << t << endl;

    }

    in.close();

    out.close();

}

//利用已建好的哈夫曼树，对文件CodeFile.data中的代码进行解码形成原文，结果存入文件Textfile.txt中

void Decode(string from, string to) {

    ifstream in(from);

    ofstream out(to, ios::app);

    string t;   //记录每行的编码文

    string ans; //记录每行的解码结果

    while(getline(in, t)) {

        ans = "";

        HTNode copy\_root = root;

        HTNode \* PHead = &copy\_root;

        for(int i = 0; i < t.size(); i++) {

            if (t[i] == '0')

                PHead = PHead->left;    //往左寻找

            else if(t[i] == '1')

                PHead = PHead->right;   //往右寻找

            if(PHead->ch != '@') {  //找到字符节点

                ans += PHead->ch;

                PHead = &copy\_root; //重新从根节点向下寻找下一个字符

            }

        }

        out << ans << endl;

    }

    in.close();

    out.close();

}

//输出DataFile.data中出现的字符以及各字符出现的频度（或概率）；输出ToBeTran.data及其报文Code.txt；输出CodeFile.data及其原文Textfile.txt

void Output() {

    int choose;

    while(true) {

        // system("cls");

        cout << "\n1. 显示DataFile.data中出现的字符以及各字符出现的频度" << endl;

        cout << "2. 显示构建的哈夫曼树的层次遍历结果" << endl;

        cout << "3. 显示字符编码对照表" << endl;

        cout << "4. 输出ToBeTran.data及其报文Code.txt" << endl;

        cout << "5. 输出CodeFile.data及其原文Textfile.txt" << endl;

        cout << "0. 返回主菜单" << endl;

        cout << "\n请输入0——5" << endl;

        cin >> choose;

        cout << endl;

        switch (choose)

        {

        case 1: {

            show\_que(statistic\_queue);

            // system("pause");

            break;

        }

        case 2: {

            show\_vec(order\_res);

            // system("pause");

            break;

        }

        case 3: {

            show\_map(code\_map);

            // system("pause");

            break;

        }

        case 4: {

            string t;

            ifstream in("../data/ToBeTran.data");

            cout << "ToBeTran.data的明文的内容为: \n" << endl;

            while(getline(in, t)) {

                cout << t << endl;

            }

            in.close();

            ifstream i("../data/Code.txt");

            cout << "\n明文ToBeTran.data加密后的报文Code.txt的内容为: \n" << endl;

            while(getline(i, t)) {

                cout << t << endl;

            }

            cout << endl;

            i.close();

            // system("pause");

            break;

        }

        case 5: {

            string t;

            ifstream in("../data/CodeFile.data");

            cout << "CodeFile.data加密报文的内容为: \n" << endl;

            while(getline(in, t)) {

                cout << t << endl;

            }

            ifstream i("../data/Textfile.txt");

            cout << "\nCodeFile.data加密报文的原文Textfile.txt的内容为: \n" << endl;

            while(getline(i, t)) {

                cout << t << endl;

            }

            cout << endl;

            i.close();

            // system("pause");

            break;

        }

        case 0:

            return;

        default:

            break;

        }

    }

}

//改色函数

void SetColor(unsigned short ForeColor, unsigned short BackGroundColor) {

    HANDLE hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);   //获取标准输出的句柄

    SetConsoleTextAttribute(hStdout, (ForeColor) | (BackGroundColor << 4)); //改色

}

//进度条函数

void progress\_bar() {

    for (int i = 1; i <= 50; i++) {

        SetColor(0, 14);

        printf(" ");

        printf("%d%%", i << 1);

        Sleep(51 - i);

        printf("\b\b\b");

    }

    SetColor(15, 0);

}

//主函数

int main()

{

    int choose;

    while(true) {

        // system("cls");

        cout << "\n1. 构建哈夫曼树" << endl;

        cout << "2. 编码明文" << endl;

        cout << "3. 解码报文" << endl;

        cout << "4. 输出" << endl;

        cout << "0. 结束程序" << endl;

        cout << "\n请选择0——4" << endl;

        cin >> choose;

        switch (choose)

        {

        case 1: {

            Initialzation("../data/DataFile.data");

            progress\_bar();

            cout << "\n哈夫曼树构建完成" << endl;

            // system("pause");

            break;

        }

        case 2: {

            Encode("../data/ToBeTran.data", "../data/Code.txt");

            progress\_bar();

            cout << "\n编码完成, 结果已存入文件中，可选择“4. 输出”进行查看" << endl;

            // system("pause");

            break;

        }

        case 3: {

            Decode("../data/CodeFile.data", "../data/Textfile.txt");

            progress\_bar();

            cout << "\n解码完成, 结果已存入文件中，可选择“4. 输出”进行查看" << endl;

            // system("pause");

            break;

        }

        case 4: {

            Output();

            break;

        }

        case 0:

            exit(EXIT\_SUCCESS);

            break;

        default:

            break;

        }

    }

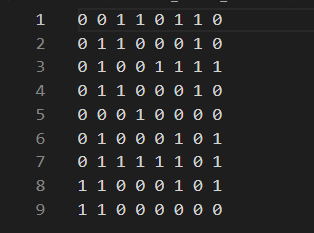
    return 0;

}

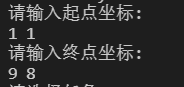
# 测试与分析

* 迷宫

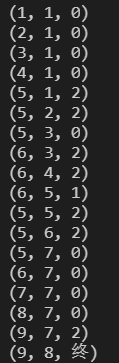
测试数据如下：9\*8的方针

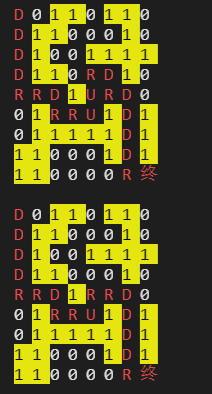


测试起点和终点如下：



测试结果：

非递归：

递归：

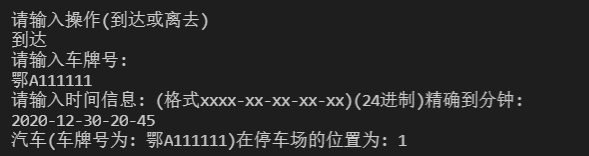
结果符合预期。

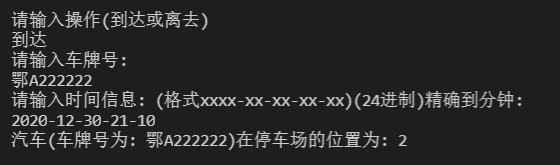
* 停车场

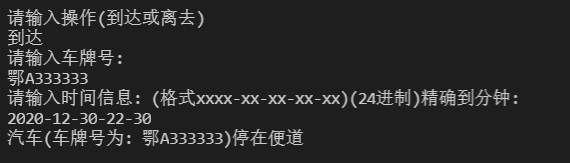
测试数据中，停车场的容量为2

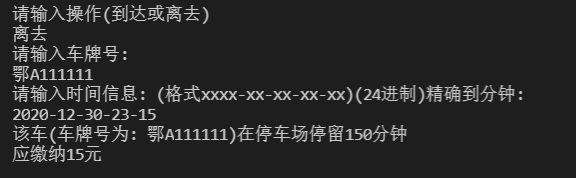


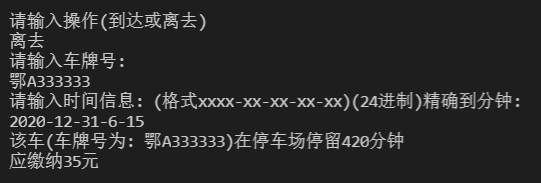
测试顺序和对应结果如下：







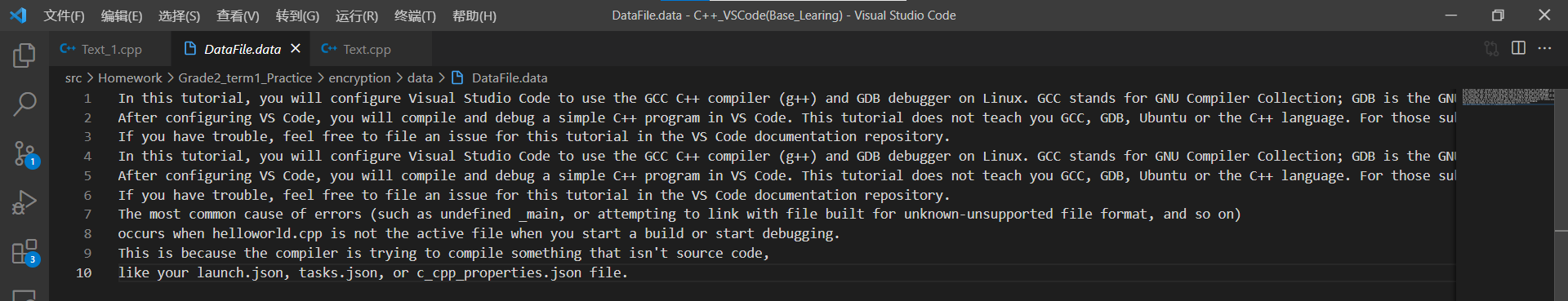


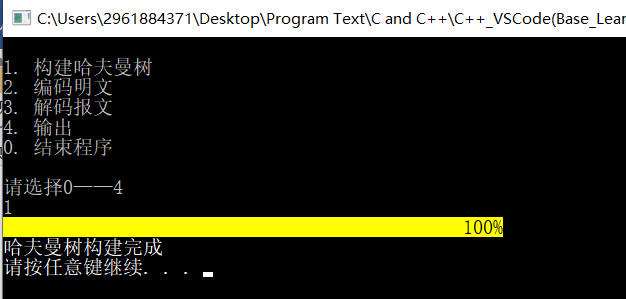


结果符合预期。

* 哈夫曼编码

构建哈夫曼树的测试文件内容如下：（选自vscode官方文档）

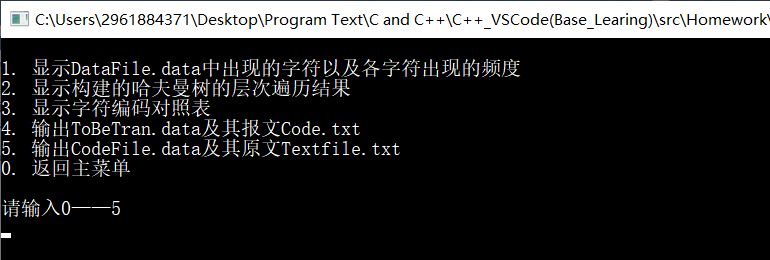








输出目录如下：

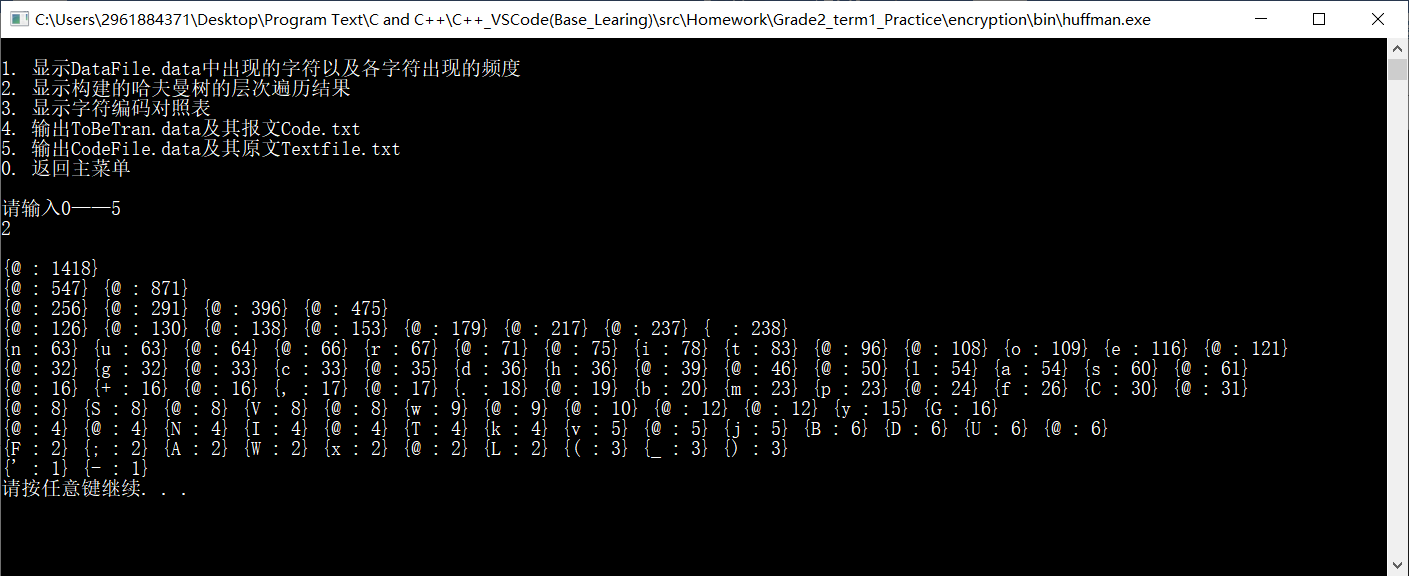


输出结果如下：

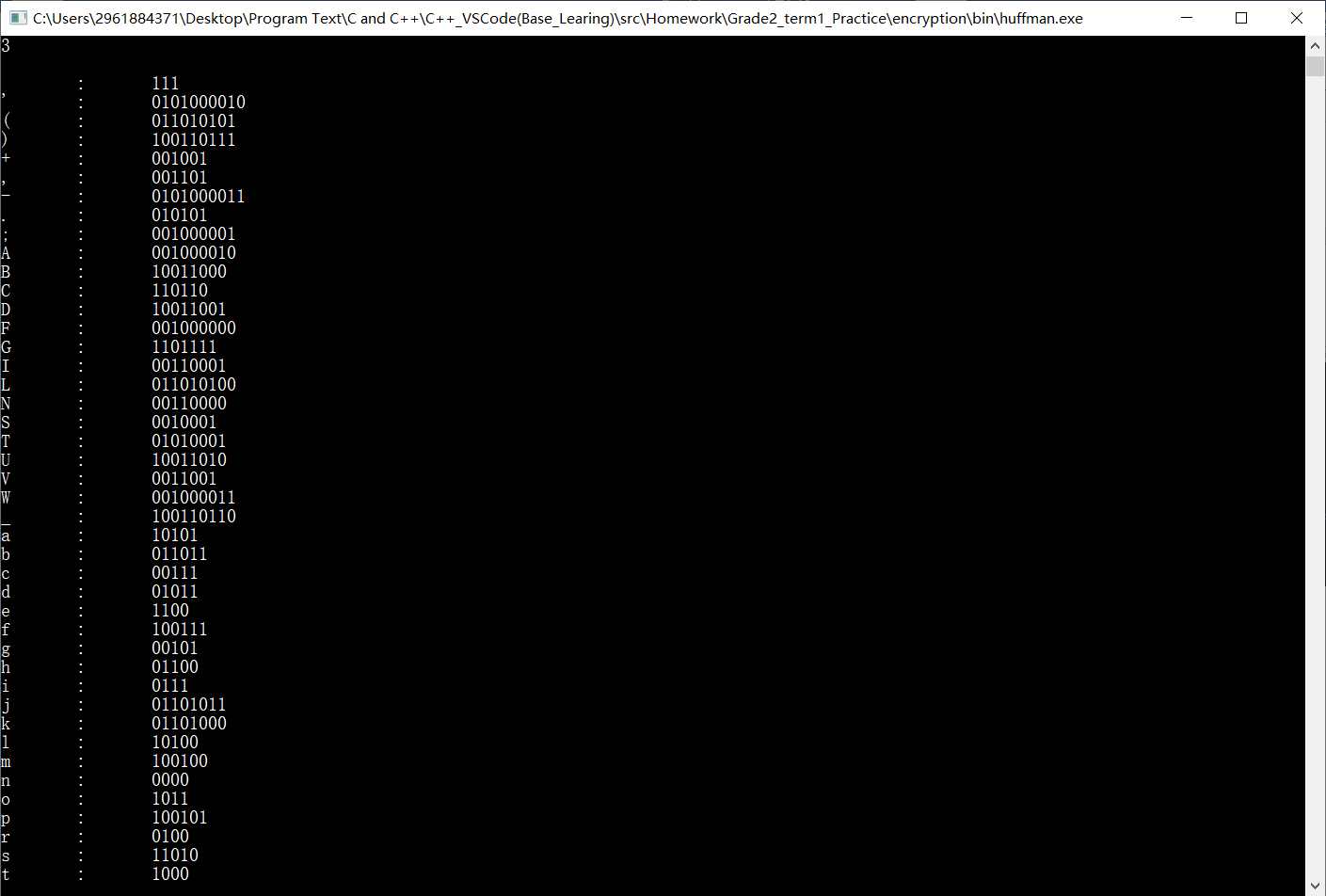
* + 1. 出现的字符以及其频度



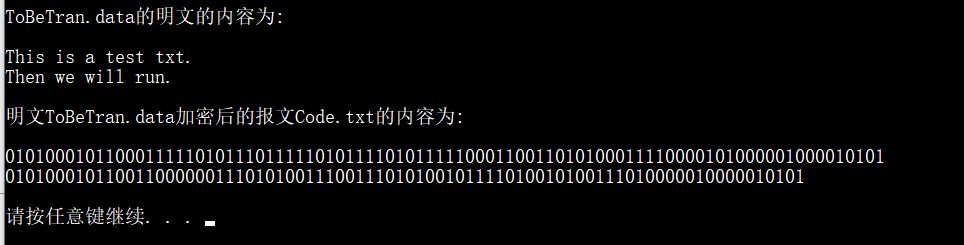
* + 1. 哈夫曼树的层次遍历结果



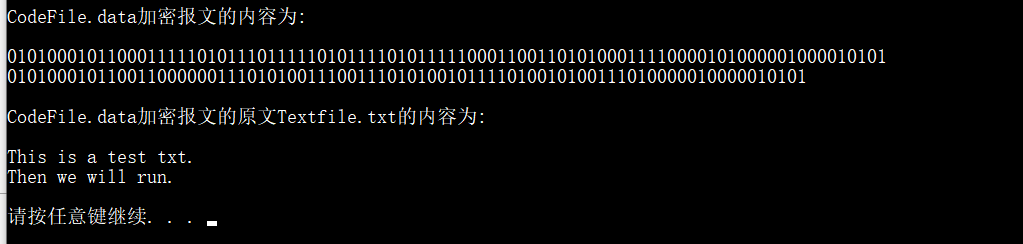
* + 1. 字符编码对照表



* + 1. 编码结果



* + 1. 译码结果



结果符合预期。

# 总结

* 迷宫：强化了对于dfs回溯的掌握，在尝试用栈完成非递归的方法时在网上查询了很多参考资料，最后采用记录前驱节点的方法，提高了我对栈的掌握和利用，也让我更加熟练的掌握不同数据结构之间的相互使用
* 停车场管理：在写关于时间的计算时，我去学习了c语言库文件time.h中对于时间数据结构clock\_t和tm的使用，也掌握了一些时间函数的使用，如mktime函数可以将tm结构转为对应的时间戳clock\_t结构，difftime可以返回两个时间戳之间所隔的秒数；在设计停车场的管理逻辑时，也加强了我对栈和队列的掌握。
* 哈夫曼编码：写node.h头文件的时候，复习了构造函数的知识；写tree.h的时候，加强了我对stl库中优先队列prior\_queue的使用；在构造哈夫曼树时，复习了数据结构课程中的二叉树和哈夫曼编码原理的知识；在译码时，利用了哈夫曼编码的互不干扰的特性，加强了我对树上递归的使用。

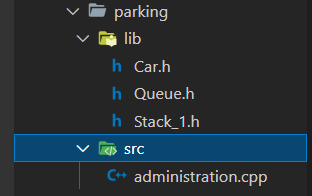
# 附录

源程序清单如下：

* 迷宫



* 停车场管理



* 哈夫曼编码

