



**数据结构实验报告**

题目名称 电梯模拟

学生学院 计算机

专业班级 软件工程17(4)班

学 号 3117004958

学生姓名 章世楠

指导教师 曾孜

2019 年 1 月 2 日

目录

[1. 需求分析 1](#_Toc2726)

[1.1. 题目描述 1](#_Toc27667)

[1.2. 题目要求 1](#_Toc3340)

[1.3. 问题分析 2](#_Toc11091)

[1.4. 程序的输入输出 4](#_Toc6680)

[1.5. 实现的功能 4](#_Toc18792)

[2. 概要设计 4](#_Toc32590)

[2.1. 数据类型的定义 4](#_Toc11786)

[2.2. 主程序流程 6](#_Toc7358)

[2.3. 模块之间的调用关系 8](#_Toc23385)

[3. 详细设计 9](#_Toc19728)

[3.1. 定义的数据类型 9](#_Toc17985)

[3.1.1. 系统相关宏定义 9](#_Toc10974)

[3.1.2. 系统数据类型定义的实现 11](#_Toc6885)

[3.2. 电梯调度的实现 13](#_Toc1514)

[3.3. 乘客进出电梯的实现 14](#_Toc5506)

[3.4. 电梯行为状态改变的实现 15](#_Toc12016)

[4. 调试分析 17](#_Toc7030)

[4.1. 设计过程的回顾分析 17](#_Toc14118)

[4.2. 算法的时空分析 18](#_Toc23870)

[4.3. 经验与体会 20](#_Toc26797)

[5. 用户使用说明 21](#_Toc20909)

[6. 测试结果 22](#_Toc10100)

[7. 附录 24](#_Toc7092)

[7.1. 头文件 24](#_Toc9832)

[7.2. 主函数 29](#_Toc14178)

[7.3. 电梯与乘客操作函数 34](#_Toc31455)

[7.4. 基本存储类型操作文件 55](#_Toc28692)

# 需求分析

## 题目描述

设计一个电梯模拟系统。这是一个离散的模拟程序，因为电梯系统是乘客和电梯等“活动体”构成的集合，虽然它们彼此交互作用，但它们的行为是基本独立的。在离散的模拟中，以模拟时钟决定每个活动体的动作发生的时刻和顺序，系统在某个模拟瞬间处理有待完成的各种事情，然后把模拟时钟推进到某个动作预定要发生的下一个时刻。

## 题目要求

（1）模拟某校五层教学楼的电梯系统。该楼有一个自动电梯，能在每层停留。五个楼层由下至上依次称为地下层、第一层、第二层、第三层和第四层，其中第一层是大楼的进出层，即是电梯的“本垒层”，电梯“空闲”时，将来到该层候命。

（2）乘客可随机地进出于任何层。对每个人来说，他有一个能容忍的最长等待时间，一旦等候电梯时间过长，他将放弃。

（3）模拟时钟从0开始，时间单位为0.1秒。人和电梯的各种动作均要耗费一定的时间单位（简记为t)，比如：

有人进出时，电梯每隔40t测试一次，若无人进出，则关门；

关门和开门各需要20t；

每个人进出电梯均需要25t；

如果电梯在某层静止时间超过300t，则驶回1层候命。

1. 按时序显示系统状态的变化过程：发生的全部人和电梯的动作序列。

**[选做内容]**

（1）增加电梯数量，模拟多梯系统。

（2）某高校的一座30层住宅楼有三部自动电梯，每梯最多载客15人。大楼每层八户，每户平均3.5人，每天早晨平均每户有3人必须在7时之前离开大楼去上班或上学。模拟该电梯系统，并分析分别在一梯、二梯和三梯运行情况下，下楼高峰期间各层的住户应提前多少时间候梯下楼？研究多梯运行最佳策略。

## 问题分析

此次的任务完成主要分为四部分：对乘客建模、对电梯建模、多电梯调度算法，电梯的可视化设计。

**一、乘客建模**

每个人从进入系统到离开称为该人在系统中的存在周期。在此周期内，他有六种可能发生的动作：

**M1.**[进入系统，为下一人的出现作准备] 产生以下数值：

InFloor -- 该人进入哪层楼；

OutFloor -- 他要去哪层楼；

GiveupTime -- 他能容忍的等候时间；

InterTime -- 下一人出现的时间间隔，据此系统预置下一人进入系统的时刻。为了保证数据的多样性，上述的四个变量的数据来源都是通过随机数来获得。

**M2.**[按电钮并等候] 此时应对以下不同情况作不同的处理：

1）Floor=InFloor且电梯的下一个活动是E6（电梯在本层，但正在关门）；

2）Floor=InFloor且D3≠0（电梯在本层，正有人进出）；

3）其它情况，可能D2=0或电梯处于活动E1（在1层停候）。

**M3.**[进入排队] 在等候队列Queue[InFloor]末尾插入该人，并预置在GiveupTime个t之后他若仍在队列中将实施动作M4。

**M4.**[放弃] 如果Floor≠InFloor或D1=0，则从Queue[InFloor]和系统删除该人。如果Floor=InFloor且D1≠0，他就继续等候（他知道马上就可进入电梯）。

**M5.**[进入电梯] 从Queue[InFloor]删除该人，并把他插入到Elevator（电梯）栈中。置CallCar[OutFloor]为1。

**M6.**[离去] 从Elevator和系统删除该人。

**二、电梯建模**

电梯有三种状态：GoingUp(向上)，GoingDown（向下），Idle（等候）.

电梯的活动有九种：

**E1.**[在1层停候] 若有人按下一个按钮，则调用Controler将电梯转入活动E3或E6。

**E2.**[要改变状态？] 如果电梯处于GoingUp（或GoingDown）状态，但该方向的楼层却无人等待，并且高层(底层)没有人响应。则要看反方向楼层是否有人等候，而决定置State为GoingDown（或GoingUp），如果上下都没有响应则进入Idle等待状态。

**E3.**[开门] 开门关门都需要20t的时间。

**E4.**[让人出入] 如果Elevator不空且有人的OutFloor=Floor，则按进入的倒序每隔25个t让这类人立即转到他们的动作M6。Elevator中不再有要离开的人时，如果Queue[Floor]不空，则以25个t的速度让他们依次转到M5。

**E5.**[关门] 每隔40 个t检查是否有人出入。如果有，则开门，否则并预置电梯再20个t后启动活动E6（再关门期间，若有人到来，则如M2所述，门再次打开）。

**E6.**[准备移动] 置CallCar[Floor]为0，而且若State≠GoingDown则置CallUp[Floor]为0，若State≠GoingUp则置CallDown[Floor]为0。调用Controler函数。

如果State=Idle，则即使已经执行了Controler，亦转到E1。最后，如果State=GoingUp，则预置15个t后（电梯加速）转到E7；如果State=GoingDown，则预置15个t后（电梯加速）转到E8。

**E7.**[上升一层] 置Floor加1并等候51个t。如果现在CallCar[Floor]=1或CallUp[Floor]=1，或者如果（（Floor=1或CallDown[Floor]=1）且CallUp[j]=CallDown[j]= CallCar[j]=0对于所有j>Floor），则预置14个t后（减速）转到E2。否则重复E7。

**E8.**[下降一层] 除了方向相反之外，与E7类似，但那里的51和14个t此时分别改为61和23个t（电梯下降比上升慢）。

**E9.**[置不活动指示器] 调用Controler函数（E9是由E3预置的，但几乎总是被E6取消了）。

需要对电梯的三种状态以及九种动作进行建模设计。

**三、多电梯调度算法**

为了更好地发挥每台电梯的性能，可能会造成多台电梯抢占一层乘客资源的情况，因此在多系统中需要设计电梯调度算法，合理有效地去管理电梯系统。保证俩台电梯不出现一起竞争同一层乘客，以及能够做到快速高效地响应用户的所有请求，电梯调度算法关系着整个系统的性能，是整个电梯系统的核心控制。

**四、界面的显示**

如果直观有效地将复杂的电梯显示出来，是本系统的关键要素，如果你设计了电梯模拟系统，但是无法让用户进行直观地进行观察，分析，则该模拟系统的意义不大。而本系统是在用C语言实现的，实现图形界面难度大，所以如果在控制台实现生动、简洁的界面成为了一个研究重点。

## 程序的输入输出

程序有一个模拟时钟Time，模拟时钟Time的初值为0，终值可在500~10000范围内逐步增加。程序通过输入模拟时间的终值来控制程序的结束，因此其输入为终止时间，数值为500-10000，输出为电梯的变化情况，以及乘客的信息，通过控制台界面显示。

## 实现的功能

|  |  |
| --- | --- |
| 基本功能的实现 | 选做功能或亮点 |
| 1. 电梯三种状态的改变，电梯通过判断是否有高层(底层)请求，来决定是否要改变方向 | 1. 电梯模拟系统的界面实现，通过信息的可视化可以清晰直观地看到电梯的工作情况(亮点) |
| 1. 电梯9种动作的实现 | 7.多电梯系统的实现(选做) |
| 1. 乘客6种动作的实现 | 8.电梯调度算法的实现，这是选做内容，也是本系统的亮点设计。 |
| 1. 通过产生随机数来构造乘客 | 9.对每层的电梯都设置了向上、向下按钮、以及响应事件。(亮点) |
| 1. 时序管理系统的模拟，通过设置模拟时钟以及延时操作来实现同步模拟(亮点) |  |
| 总结：   1. 基本功能全部实现 2. 选做了多电梯系统 3. 扩展实现了电梯调度算法 | |

# 概要设计

## 数据类型的定义

乘客数据类型定义：

ADT(乘客)

{

乘客编号、乘客的目标楼层、乘客所在楼层、乘客等待放弃时间

}

乘客等待队列：

每一层都有俩个等待队列，一个用来存储该层向上的等待乘客，一个用来存储向下的等待乘客

ADT（等待队列）

{

队列的头指针

队列的未指针

该队列中元素的数量

(每个队列的节点都包含：数据域->指向信息数据，next指针->指向下一节点)

}

乘客栈：是电梯结构体的成员，共有5个，分别用来存储5层楼的乘客

ADT(乘客栈)

{

栈底指针

栈顶指针

栈的存储空间

每个栈的节点都包含了 数据域->指向信息数据，next指针->指向下一节点

}

电梯状态跟动作的定义，共有3中状态、9种动作：

//状态的定义

enum ElevatorMove { Opening, Opened, Closing, Closed, Moving, SlowDown, Waiting, Accelerate }; //电梯的8种动作

enum ElevatorState{ GoingUp, GoingDown,Idle }; //电梯的状态

还有俩个全局数据，CallUp[5] CallDown[5]用来模拟每层电梯的按钮。

电梯数据类型定义：

ADT（电梯）

{

乘客数量：包括乘客的所以基本信息

乘客栈：每一层都需要建立一个乘客栈，方便模拟乘客出电梯时(先进后出)的操作

乘客ID数组：通过该数组能够清楚地知道在该电梯中存在哪些乘客

当前所在楼层：用来判断楼层位于何处

目标楼层数组：存放电梯行动的目的楼层

电梯的行为：电梯共有9种行为，需要根据当前行为做出下一行为的判断

电梯的状态：电梯有向上、向下、等候三种状态。

}

通过定义等待队列，能够很好地模拟等待，进入电梯的过程，同时电梯内有乘客栈，所以每到了一层就判断该层的乘客栈是否为空，如果不为空，则让其出栈，有了乘客栈可以轻松模拟乘客出栈。

电梯内设有一个目标楼层数组，用来存放乘客的目标楼层，可以根据此目标楼层来决定是否要继续往上或者往下行动。

而ClientID则主要是用来输出用的，通过该数组能够快速地输出电梯内所以乘客的编号，而不用去乘客栈搜索，相当于空间换时间，但是是非常划算的。

## 主程序流程

整个程序的流程为：在用户输入终止时间后

初始化：初始化有几个步骤，对等待队列的初始化，对每个楼层的电梯按键的初始化，对电梯进行初始化(包括对电梯内乘客栈的初始化)，显示电梯的框架、电梯的主体。

1. 判断是否到达产生下一乘客的时间，若是，产生新的乘客。
2. 检测是否有乘客想要放弃乘坐电梯。
3. 调用电梯调度函数，该函数的工作为：判断电梯是否处于等待状态，如果是，则将距离当前电梯楼层最近的乘客资源分配给该电梯，如果没有乘客的请求，则继续等待。
4. 判断电梯是否处于开门状态，若是，则判断是否到满足了进出时间间隔(25t)，如果满足了，就让乘客进出。否则进出时间+1.
5. 判断当前行为的时间周期是否到了，如果是的话，就调用状态改变函数，去改变其状态。
6. 如果时间到了，结束，否则继续。

这就是整个程序的流程，下面是该主程序的流程图。

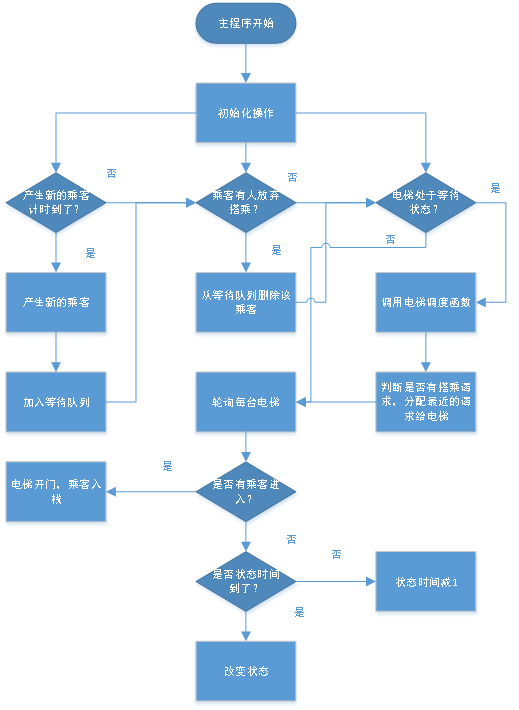


图 1 主程序流程图

## 模块之间的调用关系

整个系统可以分为以下模块：

主程序模块(运行时的主函数)：是整个系统的最高层，用来全局控制电梯

电梯调度模块：该模块在电梯处于等待时状态时，通过电梯调度算法来判断电梯的下一个方向，是电梯运行的决策模块。

电梯状态行为改变模块：如果状态计时到了，需要改变电梯的状态，这个是电梯的核心模块，也是整个系统的核心模块，通过该模块实现电梯9种动作、3种状态的改变。

乘客进出队列模块：这个模块是实现乘客进出电梯的逻辑，属于关键模块，需要涉及到乘客出等待队列以及乘客加入到乘客栈这2种操作，以及需要等待队列模块以及乘客栈模块作为存储支持。

等待队列模块：等待队列用来存放处于等待状态的用户的信息，每一层都要俩个等待队列，一个向上，一个向下。

电梯的乘客栈模块：电梯的乘客栈用来存放进入乘客的乘客，共有5个栈，对呀5个楼层，例如2楼进来的想去3楼的乘客，是放到3楼的乘客栈中，这样模拟乘客出去就直接让其出栈即可。

模块调用结构图如下：

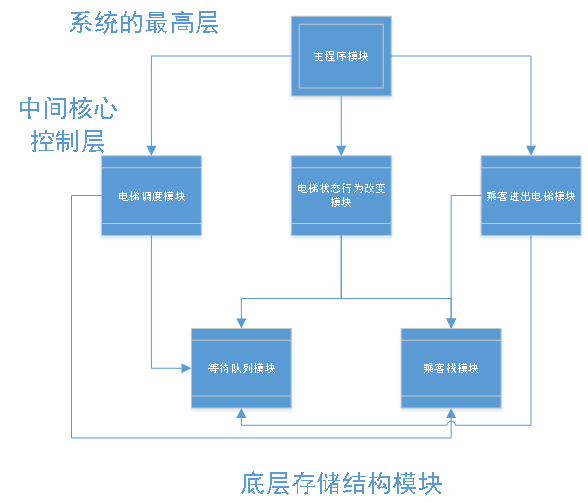


图 2 系统的模块调用图

# 详细设计

## 定义的数据类型

### 系统相关宏定义

//所有的宏定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define STACK\_INIT\_SIZE 8 //存储空间初始分配量

#define STACKINCREMENT 5 //存储空间分配增量

#define Run 1;

//#define Debug 1

#ifdef Run

#define CloseTestTime 40 //电梯关门测试时间,有人进出时，电梯每隔40t测试一次，若无人进出，则关门

#define OverTime 300 //电梯停候超时时间,如果电梯在某层静止时间超过300t，则驶回1层候命

#define AccelerateTime 15 //加速时间

#define UpTime 51 //上升时间

#define DownTime 61 //下降时间

#define UpSlowTime 14 //上升减速

#define DownSlowTime 23 //下降减速

#define DoorTime 20 //开门关门时间,关门和开门各需要20t

#define InOutTime 25 //进出电梯时间,每个人进出电梯均需要25t

#endif

#define MaxGiveUpTime 100 //乘客最长忍耐时间

#define MaxInterTime 100 //下一个乘客最长的出现时间

#define MaxWaitNumInFloor 7 //每一层的最大等待人数

#define MaxFloor 4 //最高层

#define MinFloor 0 //最低层

#define MaxElevator 10

#define Empty 0

#define FloorHight 3 //楼层高度

#define LeftElevatorX 6 //左边电梯的x起始坐标

#define LeftElevatorY 12 //左边电梯的y起始坐标

#define RightElevatorX 34 //右边电梯的x起始坐标

#define RightElevatorY 12 //右边电梯的y起始坐标

宏定义的类型分为3类：

第一种是栈、队列等基本数据结构操作函数需要用到的宏定义，像TRUE FALSE 这些的定义，都是为了表示函数操作的结果。

第二种是：对电梯、乘客建模需要用到的宏定义，像乘客的忍耐时间、电梯下降时间这些，通过宏定义能够做到高效地调整、修改整个程序的代码。

第三种是界面实现的宏定义：定义每一层楼的高度以及左右电梯的边界，通过宏定义，修改起来更加方便快捷。

### 系统数据类型定义的实现

乘客数据类型定义：

//乘客相关的结构体定义

typedef struct ClientNode {

int ClientID; //乘客的ID

int InFloor; //进入的楼层

int OutFloor; //目标楼层

int GiveupTime; //放弃时间

}\*Client;

乘客等待队列：

每一层都有俩个等待队列，一个用来存储该层向上的等待乘客，一个用来存储向下的等待乘客

//乘客队列

typedef struct ClientQueueNode {

Client data;

ClientQueueNode \* next;

}\* ClientQueuePtr;

//乘客的等待队列

typedef struct WaitQueueNode {

ClientQueuePtr front;

ClientQueuePtr rear;

int WaitClientNum;

}WaitQueue;

乘客栈：是电梯结构体的成员，共有5个，分别用来存储5层楼的乘客

//乘客栈

typedef struct ClientStackNode {

ElemType \*base; //栈底指针

ElemType \*top; //栈顶指针

int stacksize; //当前已分配存储空间，以元素为单位

}ClientStack;

电梯状态跟动作的定义，共有3中状态、9种动作：

//状态的定义

enum ElevatorMove { Opening, Opened, Closing, Closed, Moving, SlowDown, Waiting, Accelerate }; //电梯的8种动作

enum ElevatorState{ GoingUp, GoingDown,Idle }; //电梯的状态

还有俩个全局数据，CallUp[5] CallDown[5]用来模拟每层电梯的按钮。

电梯数据类型定义：

//电梯相关的结构体定义

typedef struct ElevatorNode {

int ClientNum; //乘客数量

ElevatorState State; //电梯的状态

int StateCount; //当前状态的计时

int floor; //当前所在的楼层

ElevatorMove move; //当前电梯的动作

int CallCar[5]; //目标楼层数据

ClientStack Stack[5]; //乘客栈

int ClientID[8]; //该数组用来存储电梯内的乘客编号

}\*Elevator;

## 电梯调度的实现

**实现思想：**

在多电梯系统中，如何避免电梯在资源的抢占而造成系统性能下降成为了一个重点研究问题，试想如果俩台电梯都处于等待状态，在3楼有1人按下按钮，如果同时出动俩台电梯，岂不是浪费资源。

因此，本系统引进操作系统中资源锁的概念以及自己设计的最近请求响应算法去进行设计，保证俩台电梯不发生资源抢占，尽可能少的消耗电梯的资源的前提下，能够做到快速响应用户的请求。

在本系统每层楼都有俩个电梯按钮，由全局数组CallUp[5],CallDown[5]实现的，一个表示向上，一个表示向下，初始化的值为0，当有人按下时，其值发生改变为1。资源锁就是通过该数组实现的，当有按下时，其对应楼层对应方向的元素值被设置为1，如果此时电梯想要响应它的请求，就把其值设置为-1，锁定它。设置为-1时，其他电梯无法响应它的请求，通过这样避免电梯抢占同一乘客资源。

电梯的最近响应算法：就是先找到电梯当前所在的楼层，判断距离其最近的楼层是否有搭乘请求，由CallUp、CallDown数组的值来判断，如果其中有元素值为1，则设置资源锁，把该楼层加入目标楼层，然后开始加速出发，前往该楼层。

**实现过程：**

先判断是否处于等待状态->找到最近待响应的目标楼层->设置资源锁，把该层的资源锁住->把该层加入目标楼层，驱动前往

**实现函数：**

void Controler(Elevator E[]) 用来轮询控制电梯的下一系统方向，是控制函数的主函数

int HasUpOrDownRequires(Elevator e) 被 Controler调用，用来判断是否向下或者向上的请求，返回最近请求的所在楼层，如果该楼层小于0或者大于最大楼层，则没有响应

**流程图如下所示：**

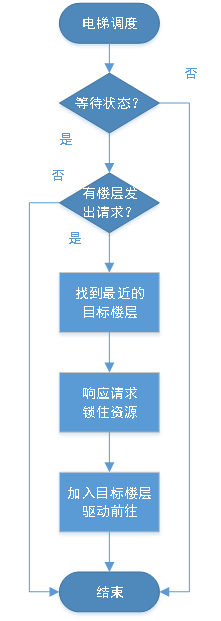


图 3 电梯调度流程图

## 乘客进出电梯的实现

**实现思想：**

首先判断是否电梯处于开门状态，让用户出电梯，出电梯的思想是根据日常情况得来的，就是先进后去，因为我们认为一般先进来的会站在电梯比较靠内的位置，所以比较晚出去。电梯出完后，就需要判断是否有人要进入电梯，需要注意的是，每个25t才能进出一个人，所以要做好计时。

**实现流程：**

判断电梯是否开门了(如果没开门返回)

->该层的乘客栈有人要出去(如果没有直接跳到乘客进入的步骤)

->若有人要出，则每个25t把一个乘客从乘客栈中Pop出来，直到没人人出栈

->根据自身的运动方向判断是否还有乘客要搭乘该电梯，向上的电梯只能接收向上的乘客，不会在此处改变方向，除非电梯已经到了目标楼层，如果电梯到了目标楼层，并且高层没有请求，而该层又刚好有反方向的乘客，则电梯才会改变方向。

->每隔25t从等待队列中删除一个乘客，加入到乘客栈中，直到等待队列为空。

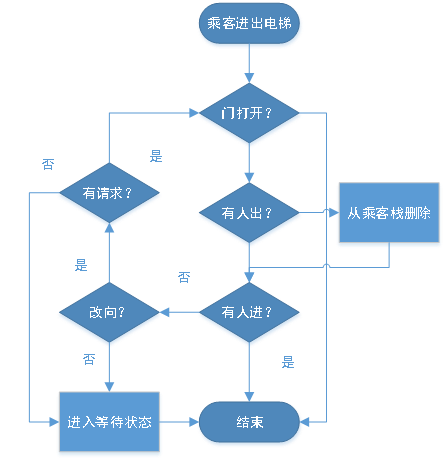


图 4 乘客进出电梯流程图

## 电梯行为状态改变的实现

**实现思想：**

通过一个状态计时函数，来判断当前的动作周期是否已经到了，若是，则开始执行状态行为改变函数，进行下一动作得判断。

**实现流程：**

1. 若电梯Opening状态动作结束，则转至Opened状态。

2. 若电梯Opened状态动作结束，若无人进出，则转至Closing状态。

3. 若电梯Closing状态动作结束，需要判断在Closing期间是否有乘客进出电梯，如果有得话，就重新开门，否则则转至Closed状态。

4. 若电梯Closed状态动作结束，需要根据情况判断，如果还有目标楼层没有到达，需要前往目标楼层，如果还有该方向得请求，例如电梯向上，如果有高层得向上得请求，则需要继续向上去响应请求，否则，进入等待状态，由电梯调度函数进行分配用户资源。

5. 若电梯在Accelerate状态动作结束，转至Moving状态.

6. 若电梯在Moving状态动作结束，需要判断下一楼层是否是目标层，如是，则转至SlowDown状态。否则，继续移动。

1. 若电梯在SlowDown状态动作结束，判断其是否要转向，如果还有目标楼层没有到达，需要前往目标楼层，如果目标楼层已经到了并且没有再继续该方向得请求，则转向。无论是否转向，都设定电梯时期，并转至Opening状态。
2. 若电梯在Waiting状态动作结束，进入电梯调度函数。

**流程图如下:**

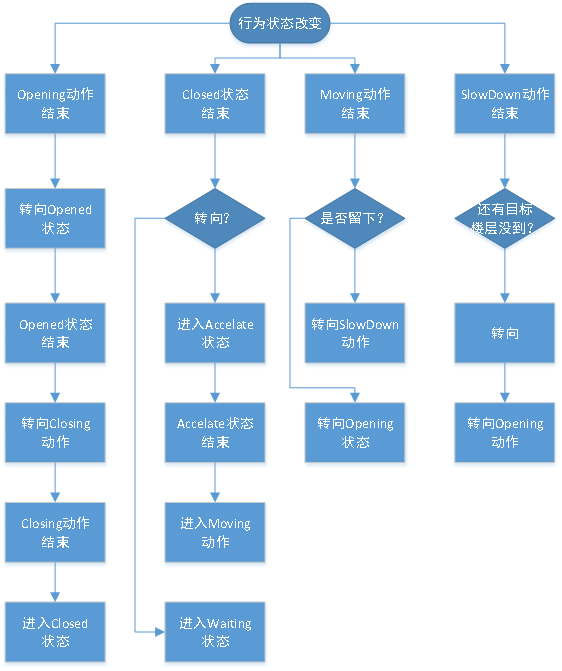


图 5 状态转换流程图

# 调试分析

## 设计过程的回顾分析

在整个电梯模拟系统的设计过程中，我遇到很多困难、挑战，我把它归结类三类，每一类都存在着若干问题。

第一类问题，数据结构的设计问题，这个问题在一开始接到题目时就遇到，当时对于模拟乘客出入的数据结构想了很久，因为自己的思维局限在用一个栈、一个队列去实现整个电梯的乘客进出管理，后来经过师兄的提醒，原来应该每一层都设置一个队列，每一层都设置一个栈才恍然大悟，犹如醍醐灌顶。接着便继续对其他电梯操作进行了分析，画图，网上查找资料，经过2天才最终敲定整个电梯系统的数据类型。但是再写出这个数据结构后依然是不够完善，后面再修修补补，总的来说就是前俩天的分析基本都能把需要的数据结构写出来，但是考虑的还不够周全，导致后面再进行了1天时间的修改。

第二类问题是，代码出错问题，这一类问题是在整个运行过程中经常会出现的，直到写文档前还出现了一个错误，原因是忘了加 &引用符号，导致删除后，内存读取出错。这样的例子数不胜数，只有有较强的代码调试能力才能游刃有余。通常数据结构的操作都是很容易造成栈溢出、内存读取失败这些问题，所以需要去设置断点去进行调式，在编写该程序期间，我设置断点进行调试的时间可能不比写程序的时间少。造成这类问题的原因可以归结为：粗心。

第三类问题是：算法时间空间设计不合理，一开始并没有写电梯调度算法，导致俩台电梯都竞争同一台电梯，造成极大的资源损耗，然后我想了很多办法去设计，加了很多代码，结果发现相当于拆东墙补西墙，整个程序缝缝补补，逻辑不清晰，于是我痛定思痛，进行理性分析，最终发现可以借用资源锁的思想进行设计，只是借用简单的资源锁数组这一数据结构就能实现复杂的算法。让我领略到了数据结构+算法设计的魅力。

整个程序的开发过程就是不断地循环出现上述三类问题，不断地解决上诉三类问题，一开始便是遇到数据设计问题，但是差点想要放弃这个题目，因为需求之多令人望而生畏，但最后还是咬咬牙坚持了下来。再设计完一个初期的数据类型之后，再开发过程中发现电梯没有设置专属的电梯栈，无法简单模拟乘客出入栈，于是便再进行乘客栈的设计。一开始只设计了一个乘客栈，后来发现不同层的乘客进出栈会造成混乱，又再进行修改，设计成每个楼层都拥有一个栈。后面的数据结构设计也是类似这样不断地更新修改。

在整个程序开发过程中，尤其是在一开始界面还没完成时，我就先写了电梯调度算法，结果当界面写完后，才发现出现了很多问题，于是便不断地设置断点进行修改，改了一整夜，后来发现问题出现最多的就是把判断条件 ==写成等于号=，造成判断失败，所以好的代码习惯相当重要。

## 算法的时空分析

**栈基本的操作分析：**

设栈的容量为N,且扩容的容量为M,栈内的元素个数为n

则进行以下时间空间的分析：

|  |  |  |
| --- | --- | --- |
| 操作名称 | 时间复杂度 | 空间复杂度 |
| Status InitStack(ClientStack &S) 初始化栈 | O(1) | O(N) |
| Status DestroyStack(ClientStack &S) 销毁栈 | O（n） | O(N) |
| Status StackEmpty(ClientStack S) 判空 | O（1） | 0 |
| Status Push(ClientStack &S, ElemType e) 入栈 | O（1） | O（1） |
| Status Pop(ClientStack &S, ElemType &e) 出栈 | O（1） | O（1） |

**队列的基本操作：**

设队列的元素有n个

|  |  |  |
| --- | --- | --- |
| 操作名称 | 时间复杂度 | 空间复杂度 |
| Status InitQueue(WaitQueue &Q)构建空队列 | O(1) | O(1) |
| Status DestroyQueue(WaitQueue &Q)销毁队列 | O(n) | O（n） |
| Status EnterQueue(WaitQueue &Q, ElemType e)入队 | O(1) | O(1) |
| Status DeleteQueue(WaitQueue &Q,ElemType &e)出队 | O(1) | O(1) |
| Status IsQueueEmpty(WaitQueue Q)判空 | O(1) |  |
| void PrintQueue(WaitQueue Q)输出队列 | O(n) |  |

**乘客的基本操作：**

设乘客的人数为n，在电梯的人数为e,在等待的人数为f

|  |  |  |
| --- | --- | --- |
| 基本操作 | 时间复杂度 | 空间复杂度 |
| void NewClient() 产生新的乘客 | O(1) | O(1) |
| Status ClientOut(Elevator &e, int k) 乘客出电梯 | O(1) | O(1) |
| Status ClientIn(Elevator &e, int k)乘客入电梯 | O(1) | O(1) |
| Status ClientGiveUp(WaitQueue &Q,Elevator E[],int floor)乘客放弃搭乘电梯 | O(f) |  |
| Status NobodyInOrOut(Elevator e)是否有人要进出电梯 | O（n） |  |
| void ShowWaitingClient(WaitQueue Q, int flag,int num)显示等待中的乘客 | O(f) | O(f) |

电梯操作函数：

设电梯内的乘客数量为n,电梯总的数量为i,楼层高度为f

|  |  |  |
| --- | --- | --- |
| 基本操作 | 时间复杂度 | 空间复杂度 |
| Status StopNextFloor(Elevator e)  是否要在下一层停留 | O（1） | O（1） |
| void ElevatorChangeMove(Elevator e,int k)  改变电梯的行为状态 | O(1)-O(f) | O(1)-O(f) |
| void Controler(Elevator E[])电梯调度函数 | O(1)-O(i\*f) | O(1)-O(i\*f) |
| PrintElevatorFrame();输出电梯框架 | O(f) | O(f) |
| void PrintfElevatorMove(Elevator e,int k)  输出电梯的行为状态 | O(1) | O(1) |
| void ShowElevatorBody(Elevator e, int k,int x)显示电梯主体，包括电梯内的乘客信息 | O(n) | O(1) |

## 经验与体会

在本次程序的设计过程中，要求熟悉掌握栈、队列的基本操作，以及设计数据结构的能力还有多个模块之间的调用，相互协作的能力。电梯系统中乘客于电梯看似是一个离散的关系，实则息息相关，需要设计数据结构去进行表示，同时要对真实的电梯进行一个建模模拟，所以这一次的数据机构设计是一个很大的挑战。

在完成这个挑战过程中遇到了很多难题，诸如上诉的那三类问题，但是在每次解决完一个问题，我都会觉得自己有了一些收获，在不断地出错Debug过程中，我发现自己设置断点调试的能力在不断提升，以前可能需要思考很久的错误，现在能够快速定位到问题的根源并纠错。同时在设计数据结构的过程中，领略到了数据结构的魅力，了解到了一个好的数据结构事半功倍，在开发过程中， 应该先想好数据结构再进行操作，算法于数据结构要相辅相成。还有就是养成了一个比较良好的代码习惯，用了很多的宏定义以及枚举变量，使得整个代码移植性好，修改容易。

总的来说整个设计过程出现上述的三类问题我觉得作为一名学生很正常，毕竟第一次接触 设计难度这么高的复杂系统，并且也是刚刚掌握数据结构。在整个解决过程中，我有过多次想更换题目的想法，毕竟这个题目是真的难，但是最终自己还是迎难而上，每天近8小时的代码编写，算法数据结构设计，虽然遇到了很多困难，但还是一一解决，自己也是比较满意这次的表现。经过此次设计我觉得个人的代码调试能力大大增强，同时我的数据结构设计思想以及算法设计思想都大有提升。

# 用户使用说明

本设计为模拟电梯系统，电梯的行动取决于乘客，而乘客的信息都是随机产生的，所以在程序运行过程中不需要进行操作。乘客的表示形式是通过序号1,2,3来表示。在运行的过程中会动态的生成电梯图像，包括电梯的轮廓，乘载的乘客，以及电梯行为信息、乘客信息、时间等各种信息。该程序只需要在刚开始的时候设置最大运行时间，此间程序会一直运行到开始时输入的运行时间，此时整个程序运行结束，按任意键退出。下面为设置是需要输入的相关设置以及设置完成时候程序的界面，如下图所示：

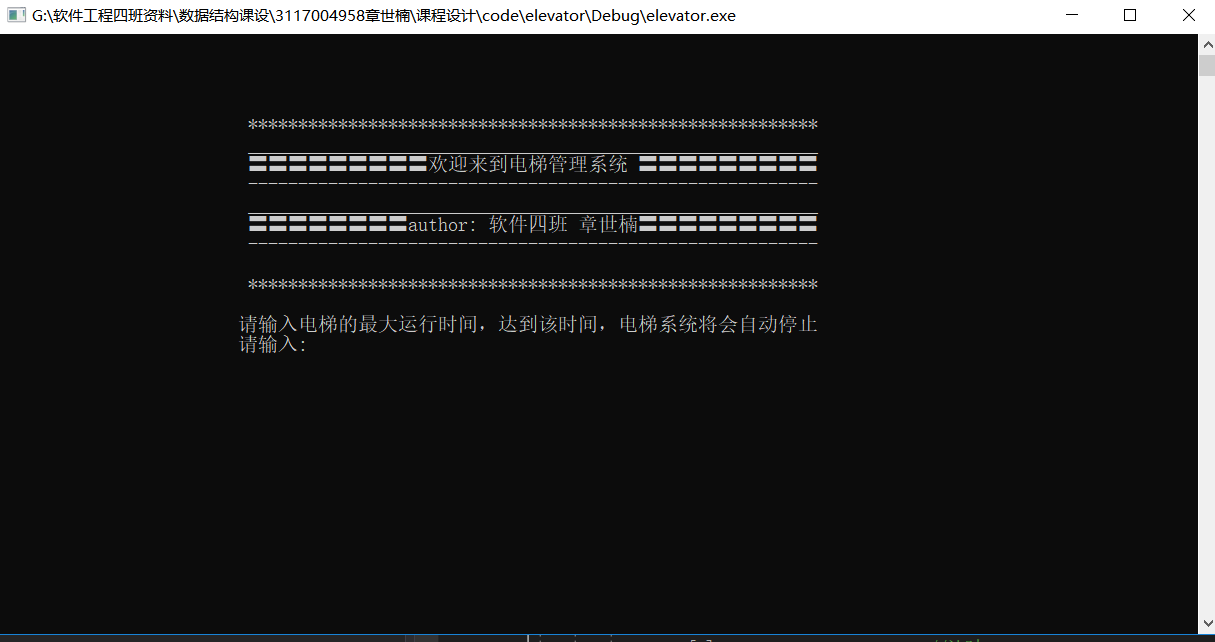


图 6 初始化界面

一开始的界面如图所示，输入时间后开始运行

# 测试结果

这是一开始的情况，1号乘客在3楼等待，准备向下前往1楼，0号电梯响应后再在移动，向上移动一层时间51t



图 7 初始图

可以看到在50t的时间向上上升一层，2号乘客此时出现在0层。并且要向1层移动，此时1号电梯做出响应，通过电梯调度函数，能够实现高效响应用户的请求。

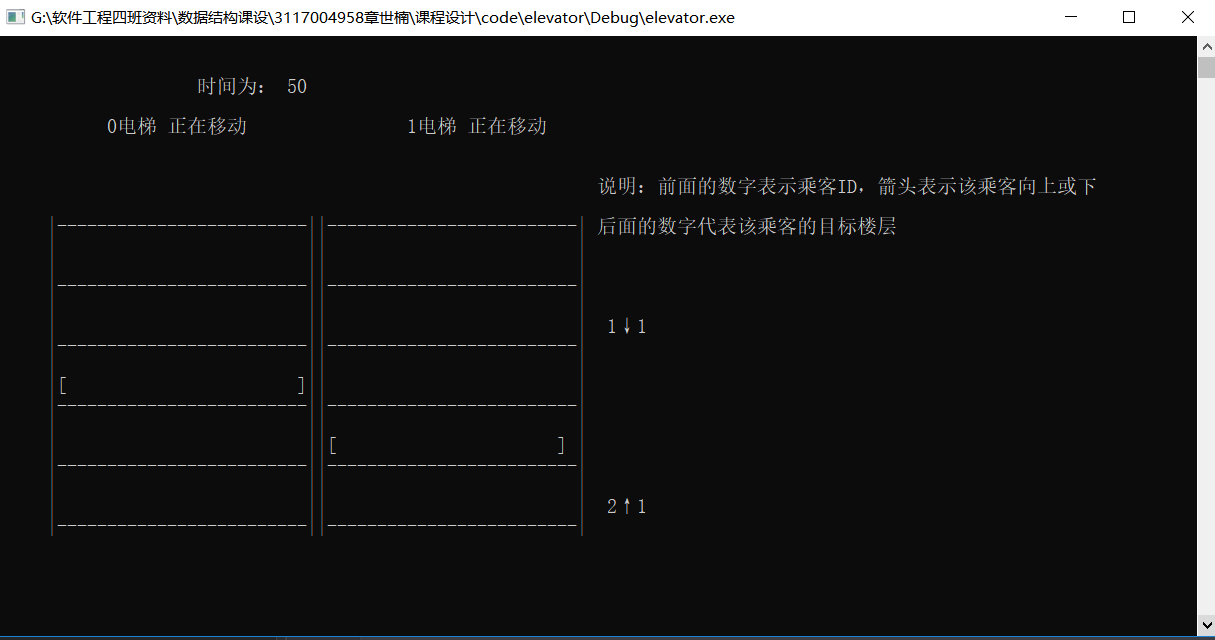


图 8 电梯移动图

因为我自己截图比较慢，所以没有截图到，1号乘客放弃电梯这一操作，但是在电梯上到第二层快到第三层时，1号乘客放弃了搭乘电梯，而3号乘客此时又刚好在3楼所以3号乘客进入电梯。同时2号乘客进入电梯1

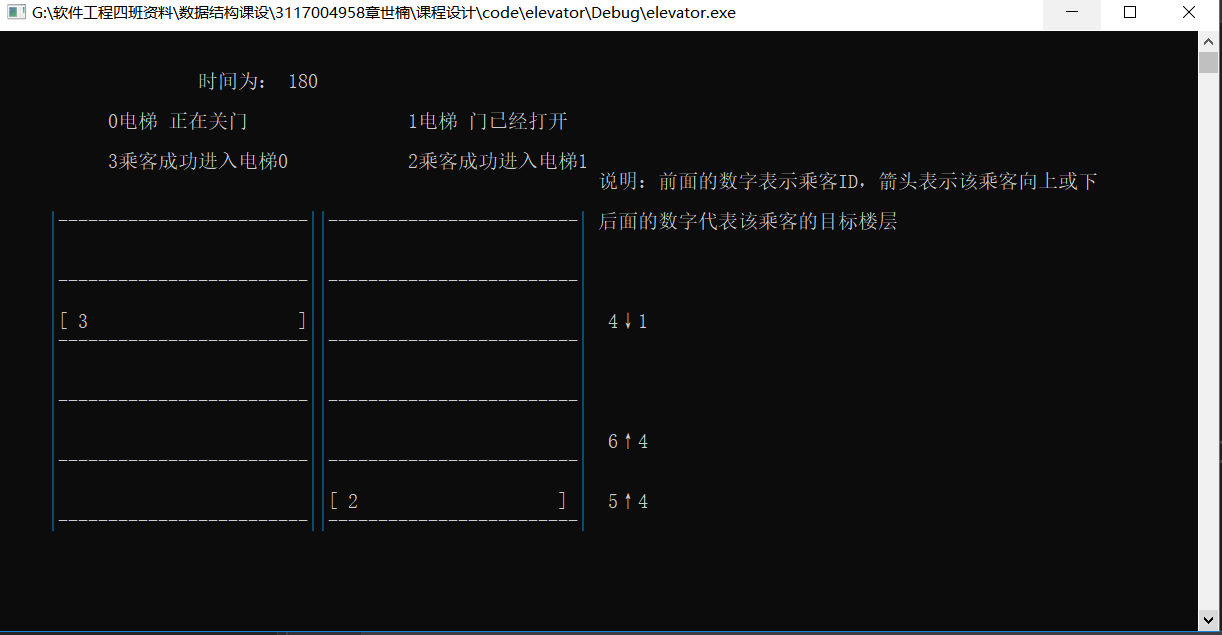


图 9 乘客进入电梯

因为5于2是同一个方向，所以一起进入电梯，而3，4是不同方向，所以不一起进入电梯。



图 10 乘客进入电梯

3号是要往4层，所以在到达4层时，让3号出电梯。从截图可以看到4号乘客此时正因忍耐时间到了，离开电梯。



图 11 3号出电梯

# 附录

共有7个文件

## 头文件

此次程序的头文件为elevator.h,其用来定义数据类型，宏定义，以及声明函数。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*elevator.h\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef ELEVATOR\_H\_INCLUDED

#define ELEVATOR\_H\_INCLUDED

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <windows.h>

#include <iostream>

using namespace std;

#include <iomanip>

#include <time.h>

//所有的宏定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define STACK\_INIT\_SIZE 8 //存储空间初始分配量

#define STACKINCREMENT 5 //存储空间分配增量

#define Run 1;

//#define Debug 1

#ifdef Run

#define CloseTestTime 40 //电梯关门测试时间,有人进出时，电梯每隔40t测试一次，若无人进出，则关门

#define OverTime 300 //电梯停候超时时间,如果电梯在某层静止时间超过300t，则驶回1层候命

#define AccelerateTime 15 //加速时间

#define UpTime 51 //上升时间

#define DownTime 61 //下降时间

#define UpSlowTime 14 //上升减速

#define DownSlowTime 23 //下降减速

#define DoorTime 20 //开门关门时间,关门和开门各需要20t

#define InOutTime 25 //进出电梯时间,每个人进出电梯均需要25t

#endif

#define MaxGiveUpTime 100 //乘客最长忍耐时间

#define MaxInterTime 100 //下一个乘客最长的出现时间

#define MaxWaitNumInFloor 7 //每一层的最大等待人数

#define MaxFloor 4 //最高层

#define MinFloor 0 //最低层

#define MaxElevator 10

#define Empty 0

#define FloorHight 3 //楼层高度

#define LeftElevatorX 6 //左边电梯的x起始坐标

#define LeftElevatorY 12 //左边电梯的y起始坐标

#define RightElevatorX 34 //右边电梯的x起始坐标

#define RightElevatorY 12 //右边电梯的y起始坐标

#ifdef Debug

//调试时设置时间短一点

#define CloseTestTime 3 //电梯关门测试时间,有人进出时，电梯每隔40t测试一次，若无人进出，则关门

#define OverTime 300 //电梯停候超时时间,如果电梯在某层静止时间超过300t，则驶回1层候命

#define AccelerateTime 1 //加速时间

#define UpTime 1 //上升时间

#define DownTime 1 //下降时间

#define UpSlowTime 1 //上升减速

#define DownSlowTime 1 //下降减速

#define DoorTime 1 //开门关门时间,关门和开门各需要20t

#define InOutTime 1 //进出电梯时间,每个人进出电梯均需要25t

#endif // DEBUG

//------------------------------------------------------

//状态的定义

enum ElevatorMove { Opening, Opened, Closing, Closed, Moving, SlowDown, Waiting, Accelerate }; //电梯的8种动作

enum ElevatorState{ GoingUp, GoingDown,Idle }; //电梯的状态

//-------------------------------------------------------------------------------------

//乘客相关的结构体定义

typedef struct ClientNode {

int ClientID; //乘客的ID

int InFloor; //进入的楼层

int OutFloor; //目标楼层

int GiveupTime; //放弃时间

}\*Client;

typedef int Status;

typedef Client ElemType;

//乘客队列

typedef struct ClientQueueNode {

Client data;

ClientQueueNode \* next;

}\* ClientQueuePtr;

//乘客的等待队列

typedef struct WaitQueueNode {

ClientQueuePtr front;

ClientQueuePtr rear;

int WaitClientNum;

}WaitQueue;

//乘客栈

typedef struct ClientStackNode {

ElemType \*base; //栈底指针

ElemType \*top; //栈顶指针

int stacksize; //当前已分配存储空间，以元素为单位

}ClientStack;

//------------------------------------------------------

//电梯相关的结构体定义

typedef struct ElevatorNode {

int ClientNum; //乘客数量

ElevatorState State; //电梯的状态

int StateCount; //当前状态的计时

int floor; //当前所在的楼层

ElevatorMove move; //当前电梯的动作

int CallCar[5]; //目标楼层数据

ClientStack Stack[5]; //乘客栈

int ClientID[8]; //该数组用来存储电梯内的乘客编号

}\*Elevator;

//------------------------------------------------------

void PrintElevatorFrame(void);

void ShowTitle();

void gotoxy(short x, short y);

void ShowWaitingClient(WaitQueue Q, int flag, int x);

void DisappearWaitingClient(WaitQueue Q, int flag, int num);

void ShowElevatorBody(Elevator e, int k, int x);

void DisappearEvelatorBody(Elevator e, int k, int x);

void PrintfElevatorMove(Elevator e, int k);

void PrintfHelp();

Status InitQueue(WaitQueue &Q);

Status DestroyQueue(WaitQueue &Q);

Status EnterQueue(WaitQueue &Q, ElemType e);

Status DeleteQueue(WaitQueue &Q, ElemType &e);

Status IsQueueEmpty(WaitQueue Q);

Status DeleteQueueNode(WaitQueue &Q, ClientQueuePtr p);

void PrintQueue(WaitQueue Q);

Status InitStack(ClientStack &S);

Status DestroyStack(ClientStack &S);

Status StackEmpty(ClientStack S);

Status Push(ClientStack &S, ElemType e);

Status Pop(ClientStack &S, ElemType &e);

Status TimeUp(Elevator e);

void ElevatorChangeMove(Elevator e, int k);

void Controler(Elevator E[]);

int HigherRequires(Elevator e);

int LowerRequires(Elevator e);

void NewClient();

Status ClientGiveUp(WaitQueue Q, Elevator E[], int floor);

Status ClientInOrOut(Elevator e, int k);

#endif

## 主函数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*main.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<stdio.h>

#include"elevator.h"

#include <stdlib.h>

#include <conio.h>

void InputMaxTime();

void init(Elevator \* E, int i);

void CheckGiveUp(Elevator E[]);

int MaxClientNum = 7; //最大乘客人数

int MaxRunTime; //最大运行时间

int NextClientInterTime; //下一个乘客进入系统的时间

int TotalClient=1; //当前总共有多少个乘客(用来产生不重复的序号)

WaitQueue FloorWaitQueue[2][5]; //每层楼有俩个等待队列，0向上,1向下

int CallUp[5], CallDown[5], CallCar[5]; //楼层的向上向下按钮,以及目标楼层,取值为1表示按下,-1表示电梯已经响应

void main()

{

//分别对应每层的电梯向上按钮，向下按钮，以及目标楼层

int i, Time = 0, t[2];

Elevator E[2];

srand((unsigned)time(NULL));

init(E,2); //全局初始化操作

t[1] = t[0] = 0;

ShowTitle();

InputMaxTime();

system("cls");

PrintElevatorFrame(); //输出电梯框架

PrintfHelp();

ShowElevatorBody(E[0],0,1);

ShowElevatorBody(E[1], 1,1); //输出电梯主体

while (Time++ < MaxRunTime)

{

//产生新的乘客

if (NextClientInterTime == 0)

NewClient();

else

NextClientInterTime--;

//检测是否乘客放弃搭乘

CheckGiveUp(E);

//调用电梯调度函数

Controler(E);

for (i = 0; i < 2; i++)

{

//如果处于开门状态，每隔一定的时间周期允许进出乘客

if (E[i]->move == Opened)

{

//每隔25s进一个人

if (t[i] == InOutTime)

{

t[i] = 0;

//等待的乘客进入电梯，应该消隐后重新显示在等待的乘客

DisappearWaitingClient(FloorWaitQueue[1][E[i]->floor], 0, 0);

DisappearWaitingClient(FloorWaitQueue[0][E[i]->floor], 1, FloorWaitQueue[1][E[i]->floor].WaitClientNum);

//乘客进出,电梯的显示情况发生改变，显示出来

if (TRUE == ClientInOrOut(E[i], i))

{

//消隐之前的电梯，显示乘客进入后的电梯

DisappearEvelatorBody(E[i],i,E[i]->floor);

ShowElevatorBody(E[i],i,E[i]->floor);

}//消隐后再显示

ShowWaitingClient(FloorWaitQueue[1][E[i]->floor], 0, 0);

ShowWaitingClient(FloorWaitQueue[0][E[i]->floor], 1, FloorWaitQueue[1][E[i]->floor].WaitClientNum);

}

else

t[i]++;

}

//如果当前状态的时间周期到了，需要改变状态

if (TimeUp(E[i]) == TRUE)

{

ElevatorChangeMove(E[i], i);

PrintfElevatorMove(E[i],i);

}

else

E[i]->StateCount--; //计时

}

if (Time % 10 == 0)

{

gotoxy(20,2);

printf("时间为： %d", Time);

}

if(Time%100==0)

PrintElevatorFrame(); //输出电梯框架

Sleep(100);

}

gotoxy(60,30);

printf("演示结束，按下任意键退出\n");

getch();

}

void init(Elevator \* E,int num)

{

int i;

//初始化等待队列操作

for (i = 0; i < 5; i++)

{

InitQueue(FloorWaitQueue[0][i]);

InitQueue(FloorWaitQueue[1][i]);

}

//初始化俩台电梯

for (i = 0; i < num; i++)

{

E[i] = (Elevator)malloc(sizeof(ElevatorNode));

E[i]->ClientNum = 0;

//初始状态在1楼停留

E[i]->floor = 1;

E[i]->State = Idle;

E[i]->StateCount = 0;

E[i]->move = Waiting;

for (int j = 0; j < 5; j++)

{

//目标楼层初始化为0

E[i]->CallCar[j] = 0;

//初始化该电梯的乘客栈

InitStack(E[i]->Stack[j]);

}

//初始化乘客ID数组

for (int j = 0; j <= 8; j++)

E[i]->ClientID[j] = 0;

}

//初始化三个电梯按钮数组

for (i = 0; i < 5; i++)

{

CallUp[i] = 0;

CallDown[i] = 0;

}

}

void CheckGiveUp(Elevator E[])

{

int i;

//检测是否乘客放弃搭乘

for (i = 0; i < 5; i++)

{

ClientGiveUp(FloorWaitQueue[0][i], E, i);

ClientGiveUp(FloorWaitQueue[1][i], E, i);

}

}

void InputMaxTime()

{

printf("\n\t\t\t请输入电梯的最大运行时间，达到该时间，电梯系统将会自动停止\n");

printf("\t\t\t请输入:");

while (scanf("%d", &MaxRunTime) != 1 || MaxRunTime <= 500 || MaxRunTime > 10000)

{

printf("\t\t\t输入有误，重新输入（500=<x<10000）\n");

while (getchar() != '\n')

continue;

printf("\t\t\t请输入:");

}

}

## 电梯与乘客操作函数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*elevator.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include"elevator.h"

extern int MaxClientNum ; //最大乘客人数

extern int MaxRunTime; //最大运行时间

extern int NextClientInterTime; //下一个乘客进入系统的时间

extern WaitQueue FloorWaitQueue[2][5]; //每层楼有俩个等待队列，0向上,1向下

extern int CallUp[5], CallDown[5]; //楼层的向上向下按钮

//判断该动作的时间周期是否到了

Status TimeUp(Elevator e)

{

if (e->State == Idle && e->floor == 1)

return FALSE;

else

if (e->StateCount<=0)

return TRUE;

}

//没有人进入电梯

Status NobodyIn(Elevator e)

{

if (e->State == GoingUp && FloorWaitQueue[0][e->floor].WaitClientNum == 0)

return TRUE;

else if (e->State == GoingDown && FloorWaitQueue[1][e->floor].WaitClientNum == 0)

return TRUE;

else

return FALSE;

}

//没有人出电梯

Status NobodyOut(Elevator e)

{

//没有 乘客坐这个楼层的电梯

if (!e->CallCar[e->floor])

return TRUE;

//该层没有乘客或者走光

else if (e->State == GoingUp && StackEmpty(e->Stack[e->floor]) == TRUE )

return TRUE;

//该层没有乘客出乘客

else if (e->State == GoingDown && StackEmpty(e->Stack[e->floor]) == TRUE )

return TRUE;

else

return FALSE;

}

//没人进出电梯

Status NobodyInOrOut(Elevator e)

{

//没有 乘客坐这个楼层的电梯

if (NobodyIn(e) == TRUE && NobodyOut(e) == TRUE)

return TRUE;

else

return FALSE;

}

//是否要在下一层停留

Status StopNextFloor(Elevator e)

{

//三种情况需要在下一层停留

//下一层有乘客，下一层为目的层，下一层为终点

if (e->State == GoingUp)

{

if (e->floor == MaxFloor)

return TRUE;

else if (CallUp[e->floor] == 1)

return TRUE;

else if (e->CallCar[e->floor] == 1)

return TRUE;

else

return FALSE;

}

else

{

if (e->floor == MinFloor)

return TRUE;

else if (CallDown[e->floor ] == 1)

return TRUE;

else if (e->CallCar[e->floor ] == 1)

return TRUE;

else

return FALSE;

}

}

//改变电梯的行为

void ElevatorChangeMove(Elevator e,int k)

{

switch (e->move)

{

case Opening:

e->move = Opened;

e->StateCount = CloseTestTime;

break;

case Opened:

//如果没有人进出或者电梯人数达到上限，则关门

if (NobodyInOrOut(e) == TRUE || e->ClientNum == MaxClientNum)

{

e->move = Closing;

e->StateCount = DoorTime;

}

else

e->StateCount = CloseTestTime;

break;

case Closing:

if (NobodyInOrOut(e) == TRUE)

{

e->move = Closed;

}

//如果关门期间有人进入，则开门

else if(e->ClientNum<MaxClientNum)

{

e->move = Opening;

e->StateCount = DoorTime;

}

break;

case Closed:

//关门完成后判断是要向上移动还是向下移动还是停止

if (e->State == GoingUp)

{

//清空响应事件

if (-1 == CallUp[e->floor])

CallUp[e->floor] = 0;

int j;

//判断是否需要继续往上

for ( j = e->floor; j <= MaxFloor; j++)

{

//如果上层有目标楼层，则需要往上

if (e->CallCar[j] == 1)

{

e->move = Accelerate;

e->StateCount = AccelerateTime;

break;

}

}//说明没有往上的目标楼层

if (j > MaxFloor)

e->move = Waiting, e->State = Idle;

}

else if (e->State == GoingDown)

{

//清空响应事件

if (-1 == CallDown[e->floor])

CallDown[e->floor] = 0;

int j;

//还有更底的目标楼层，加速启动

for ( j = e->floor; j >=0; j--)

{

if (1 == e->CallCar[j])

{

e->move = Accelerate;

e->StateCount = AccelerateTime;

break;

}

}

if (j < 0)

e->move = Waiting, e->State = Idle;

}

else

e->move = Waiting;

break;

case Accelerate:

e->move = Moving;

//上升时间跟下降时间不一样,要区分开

if (e->State == GoingUp)

e->StateCount = UpTime;

else

e->StateCount = DownTime;

break;

case Moving: //完成移动

//先消隐电梯再显示电梯

DisappearEvelatorBody(e,k,e->floor);

if (e->State == GoingUp)

{

e->floor++;

if (TRUE == StopNextFloor(e))

{

e->move = SlowDown;

e->StateCount = UpSlowTime;

}

//否则继续上升

else

e->StateCount = UpTime;

}

else if(GoingDown==e->State)

{

e->floor--;

if (TRUE == StopNextFloor(e))

{

e->move = SlowDown;

e->StateCount = DownSlowTime;

}

//继续下降

else

e->StateCount = DownTime;

}

ShowElevatorBody(e,k,e->floor);

break;

case SlowDown:

//到达该层，清空电梯内的目的楼层标志位

e->CallCar[e->floor] = 0;

//判断是否需要转变方向

if (e->State == GoingUp)

{

int need = 1;

//还没到达目标层，无需转向

for (int j = e->floor; j < MaxFloor; j++)

{

if (1 == e->CallCar[j])

{

need = 0;

break;

}

}

//否则改变方向,判断高层是否有请求，如果没有，改变方向

if (1==need&& -1==HigherRequires(e) )

e->State = GoingDown;;

}

else if (e->State = GoingDown)

{

int need = 1;

//没达到最低目标层无需改向

for (int j = e->floor; j > 0; j--)

{

if (1 == e->CallCar[j])

{

need = 0;

break;

}

}

//否则，如若底层还没响应改变状态

if (1 == need && -1 == LowerRequires(e))

e->State = GoingUp;

}

e->move = Opening;

e->StateCount = DoorTime;

break;

case Waiting: //等待状态有Control函数进行下一行动的判断

break;

}

}

//有向上或者向下的请求，返回最近一层的电梯请求

int HasUpOrDownRequires(Elevator e)

{

int i, j;

for (j = e->floor-1, i = e->floor+1; j >= 0 || i <= MaxFloor; j--, i++)

{

if (j>=0&&(CallDown[j]==1 || CallUp[j]==1))

return j;

if (i <= MaxFloor && (CallDown[i] == 1 || CallUp[i] == 1))

return i;

}

}

//高层是否有请求，有返回该楼层，没有返回-1

int HigherRequires(Elevator e)

{

int floor;

for (floor = e->floor; floor <= MaxFloor; floor++)

{

if (CallUp[floor])

return floor;

}

return -1;

}

//底层是否有请求,有返回该楼层，没有返回-1

int LowerRequires(Elevator e)

{

int floor;

for (floor = e->floor; floor >= 0; floor--)

{

if (CallDown[floor])

return floor;

}

return -1;

}

//电梯向下移动

Status ElevatorMoveDown(Elevator &e)

{

int floor = LowerRequires(e);

if (floor != -1)

{

//如果请求的是该层电梯，直接打开电梯门

if (e->floor == floor)

e->move = Opening;

else

{

e->CallCar[floor] = 1;

e->move = Accelerate;

//响应该层的请求

CallDown[floor] = -1;

}

return TRUE;

}

return FALSE;

}

//电梯向上移动

Status ElevatorMoveUp(Elevator &e)

{

int floor = HigherRequires(e);

//高楼有请求

if (floor != -1)

{

//如果响应的电梯为本层，则直接开门

if (floor == e->floor)

e->move = Opening;

else

{

e->CallCar[floor] = 1;

e->move = Accelerate;

//响应标志

CallUp[floor] = -1;

}

return TRUE;

}

else

return FALSE;

}

//控制函数，用来判断电梯的下一个移动方向

//该步骤的思想为先让0号电梯去响应最近的一台电梯

//再让1号电梯去响应最近的一台电梯

//通过响应事件，把响应层的电梯按钮设置为-1，避免俩台电梯竞争同一资源

void Controler(Elevator E[])

{

for (int i = 0; i <= 1; i++)

{//如果i号电梯位于等待状态

if (E[i]->move == Waiting)

{

//计算最近的上下电梯的花费

int floor;

//如果在本层有乘客，直接开门

if (CallUp[E[i]->floor] == 1)

E[i]->move = Opening, CallUp[E[i]->floor] = -1, E[i]->State = GoingUp;

else if (CallDown[E[0]->floor] == 1)

E[i]->move = Opening, CallDown[E[i]->floor] = -1, E[i]->State = GoingDown;

//否则找到最近的请求楼层

else

{

floor = HasUpOrDownRequires(E[i]);

//没有乘客响应

if (floor<0 || floor>MaxFloor)

return;

else if (1==CallUp[floor])

{

//向上运动

if (floor >= E[i]->floor)

{

E[i]->move = Accelerate;

E[i]->CallCar[floor] = 1;

//设置楼层响应标志

CallUp[floor] = -1;

E[i]->State = GoingUp;

}

else

{

E[i]->move = Accelerate;

E[i]->CallCar[floor] = 1;

//设置楼层响应标志

CallUp[floor] = -1;

E[i]->State = GoingDown;

}

}

else if (CallDown[floor] == 1)

{

//电梯向上运动

if (floor >= E[i]->floor)

{

E[i]->move = Accelerate;

E[i]->CallCar[floor] = 1;

//设置楼层响应标志

CallDown[floor] = -1;

E[i]->State = GoingUp;

}

else

{

E[i]->move = Accelerate;

E[i]->CallCar[floor] = 1;

//设置楼层响应标志

CallDown[floor] = -1;

E[i]->State = GoingDown;

}

}

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*client.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include"elevator.h"

extern int NextClientInterTime; //下一个乘客进入系统的时间

extern int TotalClient; //总共有多少个乘客

extern WaitQueue FloorWaitQueue[2][5]; //每层楼有俩个等待队列，0向上,1向下

extern int CallUp[5], CallDown[5]; //楼层的向上向下按钮,以及目标楼层

extern int MaxClientNum;

void NewClient()

{

Client c;

c = (Client)malloc(sizeof(ClientNode));

srand((unsigned)time(NULL));

//产生所在楼层，目前楼层，忍耐时间

c->ClientID = TotalClient++;

c->GiveupTime = rand()%MaxGiveUpTime;

c->InFloor = rand()\*11 % (MaxFloor+1);

c->OutFloor = rand()\*17 % (MaxFloor+1);

//如果目前楼层于现在楼层一样，则重新赋值

while (c->OutFloor == c->InFloor)

c->OutFloor= rand() % MaxFloor;

NextClientInterTime = rand() % MaxInterTime;

//下一 乘客出现的时间在5-120s之间

while(NextClientInterTime<5)

NextClientInterTime = rand() % MaxInterTime;

//进入等待队列

//先判断其是向上还是先下，然后消隐原来的队列显示，加入乘客到队列

//再重新显示新的乘客队列

if (c->InFloor > c->OutFloor)

{

DisappearWaitingClient(FloorWaitQueue[1][c->InFloor], 0, 0);

DisappearWaitingClient(FloorWaitQueue[0][c->InFloor], 1, FloorWaitQueue[1][c->InFloor].WaitClientNum);

//进入等待队列

EnterQueue(FloorWaitQueue[1][c->InFloor], c);

//按下该层的电梯按钮

if(CallDown[c->InFloor] !=-1)

CallDown[c->InFloor] = 1;

ShowWaitingClient(FloorWaitQueue[1][c->InFloor], 0,0);

ShowWaitingClient(FloorWaitQueue[0][c->InFloor], 1,FloorWaitQueue[1][c->InFloor].WaitClientNum);

}

else

{

DisappearWaitingClient(FloorWaitQueue[1][c->InFloor], 0, 0);

DisappearWaitingClient(FloorWaitQueue[0][c->InFloor], 1, FloorWaitQueue[1][c->InFloor].WaitClientNum);

EnterQueue(FloorWaitQueue[0][c->InFloor], c);

//按下该层的电梯按钮

if (CallUp[c->InFloor] != -1)

CallUp[c->InFloor] = 1;

//显示向上跟向下的乘客

ShowWaitingClient(FloorWaitQueue[1][c->InFloor], 0,0);

ShowWaitingClient(FloorWaitQueue[0][c->InFloor], 1,FloorWaitQueue[1][c->InFloor].WaitClientNum);

}

}

Status ClientOut(Elevator &e, int k)

{

Client p;

int i, j;

//说明有乘客的目标楼层为这个,并且该层乘客栈不为空,让乘客出栈,每出去一个乘客需要等待25个t

if (StackEmpty(e->Stack[e->floor]) != TRUE)

{

Pop(e->Stack[e->floor], p);

for (i = 0; i < MaxClientNum; i++)

{

if (e->ClientID[i] == p->ClientID)

break;

}

e->ClientNum--;

//覆盖掉乘客ID数组的数据

for (j = i; j < 7; j++)

e->ClientID[j] = e->ClientID[j + 1];

gotoxy(10 + 30 \* k, 6);

printf(" ");

gotoxy(10 + 30 \* k, 6);

printf(" %d乘客出电梯 %d", p->ClientID, k);

free(p);

return TRUE;

}

else

return FALSE;

}

Status ClientIn(Elevator &e, int k)

{

Client p;

if (e->ClientNum >= 8)

return FALSE;

switch (e->State)

{

//如果该电梯向下，只接收向下的乘客

case GoingDown:

if (FloorWaitQueue[1][e->floor].WaitClientNum != 0)

{ //出队成功

if (OK == DeleteQueue(FloorWaitQueue[1][e->floor], p))

{

if (OK == Push(e->Stack[p->OutFloor], p))

{

e->CallCar[p->OutFloor] = 1;

e->ClientID[e->ClientNum++] = p->ClientID;

gotoxy(10 + 30 \* k, 6);

printf(" ");

gotoxy(10 + 30 \* k, 6);

printf(" %d乘客成功进入电梯%d", p->ClientID, k);

}

else

printf(" %d乘客进入电梯%d失败", p->ClientID, k);

}

}

else

return FALSE;

break;

case GoingUp:

//如果该电梯向下只接收向下的乘客

if (FloorWaitQueue[0][e->floor].WaitClientNum != 0)

{

if (OK == DeleteQueue(FloorWaitQueue[0][e->floor], p))

{

//压入栈中

if (OK == Push(e->Stack[p->OutFloor], p))

{

e->CallCar[p->OutFloor] = 1;

e->ClientID[e->ClientNum++] = p->ClientID;

gotoxy(10 + 30 \* k, 6);

printf(" ");

gotoxy(10 + 30 \* k, 6);

printf(" %d乘客成功进入电梯%d", p->ClientID, k);

}

else

printf(" %d乘客进入电梯%d失败", p->ClientID, k);

}

else

return FALSE;

}

else

return FALSE;

break;

//如果是处于等待状态需要谨慎考虑

case Idle:

break;

}

}

//乘客进出电梯处理函数，先出后进

Status ClientInOrOut(Elevator e, int k)

{

if (ClientOut(e, k) == FALSE)

{

if (ClientIn(e, k) == FALSE)

return FALSE;

//有人进入电梯，需要重新显示等待队列

DisappearWaitingClient(FloorWaitQueue[1][e->floor], 0, 0);

DisappearWaitingClient(FloorWaitQueue[0][e->floor], 1, FloorWaitQueue[1][e->floor].WaitClientNum);

ShowWaitingClient(FloorWaitQueue[1][e->floor], 0, 0);

ShowWaitingClient(FloorWaitQueue[0][e->floor], 1, FloorWaitQueue[1][e->floor].WaitClientNum);

}

return TRUE;

}

//检测队列中的乘客是否要放弃搭乘电梯

//floor表示当前楼层，flag表示向下或者向上

Status ClientGiveUp(WaitQueue Q,Elevator E[],int floor)

{

Client node;

if (Q.WaitClientNum <= 0)

return FALSE;

else if (NULL != Q.front->next)

node = Q.front->next->data;

else

return FALSE;

//如果忍耐时间到了并且该层没有电梯

if (node != NULL && node->GiveupTime <= 0 && E[0]->floor != floor && E[1]->floor != floor)

{

gotoxy(10 + 30, 6);

printf(" ");

gotoxy(10 + 30, 6);

printf(" %d号乘客放弃搭乘电梯\n", node->ClientID);

//等待的乘客离开电梯，应该消隐后重新显示在等待的乘客

DisappearWaitingClient(FloorWaitQueue[1][floor], 0, 0);

DisappearWaitingClient(FloorWaitQueue[0][floor], 1, FloorWaitQueue[1][floor].WaitClientNum);

DeleteQueue(Q, node);

ShowWaitingClient(FloorWaitQueue[1][floor], 0, 0);

ShowWaitingClient(FloorWaitQueue[0][floor], 1, FloorWaitQueue[1][floor].WaitClientNum);

}

else

node->GiveupTime--;

return TRUE;

}

## 基本存储类型操作文件

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*stack.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "elevator.h"

Status InitStack(ClientStack &S) {

//构造一个空栈

S.base = (ElemType \*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if (!S.base) return OVERFLOW;

S.top = S.base;

S.stacksize = STACK\_INIT\_SIZE;

return OK;

}//InitStack

Status DestroyStack(ClientStack &S) {

//销毁栈S

ElemType \*p;

if (S.base) {

for (p = S.base; p < S.top; p++)

// DestoryClient(\*p);

free(S.base);

}

return OK;

}

Status StackEmpty(ClientStack S) {

//若栈S为空，则返回TRUE，否则返回FALSE

if (S.top == S.base)

return TRUE;

else

return FALSE;

}

Status Push(ClientStack &S, ElemType e) {

//入栈

if (!S.base)

return ERROR;

if (S.top - S.base >= S.stacksize) {

S.base = (ElemType \*)realloc(S.base, (S.stacksize + STACKINCREMENT) \* sizeof(ElemType));

if (!S.base)

return OVERFLOW;

S.top = S.base + S.stacksize;

S.stacksize += STACKINCREMENT;

}

\*S.top++ = e;

return OK;

}//Push

Status Pop(ClientStack &S, ElemType &e) { //出栈并将出栈信息返回

//出栈

if (S.top == S.base)

return ERROR;

e = \*(--S.top);

return OK;

}//Pop

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*queue.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "elevator.h"

//等待队列的基本操作

Status InitQueue(WaitQueue &Q) {

//构造一个空队列Q

Q.front=Q.rear= new ClientQueueNode;

if (!Q.front)

return OVERFLOW;//分配存储失败

Q.front->next = NULL;

Q.front->data = NULL;

Q.WaitClientNum = 0;

return OK;

}

Status DestroyQueue(WaitQueue &Q) {

//销毁队列Q

while (Q.front) {

Q.rear = Q.front->next;

if (Q.front->data) free(Q.front->data);

delete Q.front;

Q.front = Q.rear;

}

return OK;

}

Status EnterQueue(WaitQueue &Q, ElemType e) {

//插入元素e为Q的新的队尾元素

ClientQueuePtr p;

p = new ClientQueueNode;

if (!p)

return OVERFLOW;

p->data = e; p->next = NULL;

Q.rear->next = p;

Q.rear = p;

Q.WaitClientNum++;

return OK;

}

Status DeleteQueue(WaitQueue &Q, ElemType &e) { //带头结点的队列

//若队列不空,则删除Q的队头元素,用e返回其值,并返回OK;

//否则返回ERROR

ClientQueuePtr p;

if (Q.front == Q.rear) //如果队列为空，则返回错误

return ERROR;

p = Q.front->next; //绕过头结点

e = p->data;

Q.front->next = p->next;

if (Q.rear == p)

Q.rear = Q.front;

delete p;

Q.WaitClientNum--;

return OK;

}

Status IsQueueEmpty(WaitQueue Q) {

//判断队列是否为空

if (Q.front == Q.rear)

return TRUE;

else

return FALSE;

}

Status DeleteQueueNode(WaitQueue &Q, ClientQueuePtr p) {

//删除队列中p指向的结点的下一个结点

ClientQueuePtr q;

if (p == NULL || p->next == NULL)

return ERROR;

q = p->next;

p->next = q->next;

if (p->next == NULL)

Q.rear = p;

//free(q->data);

//delete q;

Q.WaitClientNum--;

return OK;

}

void PrintQueue(WaitQueue Q) {

//输出队列

ClientQueuePtr q;

int count = 0;

if (Q.front->next == NULL) //如果队列为空，跳转到end

goto end;

q = Q.front->next;

while (q != NULL) {

cout << setw(3) << q->data->ClientID << ' ';

q = q->next;

count++;

}

end:

while (count++ <= 4) //输出空格补全占位，每次占3个字符位

cout << " ";

}