动车组检修的最佳方案模型

摘要

本文就动车组检修问题，运用优先队列、散列表等方法，建立算法与数学模型，用编程软件，给出了一定时刻动车组维修的总时长。

对于问题一，本文首先计算每个工序每个小时的处理能力，其次运用优先队列的方法，建立一个基于优先队列的模拟时间驱动的动车组模型，然后根据此模型，用 Visual Studio 编写程序。最后得出检修完所有的动车组需要 **20 时 30 分**。

对于问题二，首先动车拥有四个类别 CRH2、CRH3、CRH5、CRH6，每个类别的动车在 a、b、c 三个车间的维修时间都不一致，其次利用**散列表**将不同类别的动车在 a、b、c 三个车间检修需要的时间映射到表中相应位置来访问记录，加快查找的速度。然后根据问题一的模型，用 Visual Studio 编写程序。当动车进入车间后判断该动车的类别，运用散列表快速查找的功能，根据动车的不同类别获取对应车间的检修时间。最后得出检修完所有的动车组需要 **9 时 1 分**。

对于问题三，首先与问题一不同的是，动车的属性新增了检修等级，与新的检修车间，不同的检修等级对应着不同的检修车间组合，一辆动车对应的检修等级和车间组合的关系，其次根据列车的行驶时间、历程和检修周期，动车组的检修被划分成不同检修等级 I~V，不同的检修等级对应不同的工序组合，利用二维数组来存储不同检修等级对应的车间的检修时间，根据检修等级映射到二位数组来获取对应的工序组合。然后根据问题一的模型，用 Visual Studio 编写程序。根据问题一的维修顺序，最后得出检修完附表二这些列车需要的总时间为 **18 时 45 分**。

关键词：优先队列　Visual Studio　散列表　映射

一、问题重述

1.1 问题背景

目前，我国的铁路已经实现了第六次大提速，动车越来越普遍出现在运营线上，并且成为我国客运交通的主要出行方式。动车的运输量与运输速度不断增长，动车的运行状况与人们的生活息息相关，动车组是铁路旅客运输的高速运载工具，动车组的运用检修工作是铁路运输的重要组成部分，其运用效率和检修质量直接关系到旅客生命财产安全和企业的经济效益。由于我国动车组检修安排不合理，造成检修周 期较长，相较国外的动车组检修周期，我国对于各个车型的动车组在检修的过程中都需要耗费更多的检修周期，这就对动车的正常上线率有影响。

1.2 问题提出

1、问题一要求解决在动车运用所 a、b、c 三道工序对应的不同车间中，12 个小时内每 15 分钟检修一辆车，按照 a-b-c 的顺序进行检修，检修完所有的动车组需要多长时间。给出最佳方案。
2、问题二给出不同的动车类别，根据附件一中动车到达动车所的时间表，计算维修完这些动车需要多长时间。
3、增加检修等级难度，增加检修工序 d、e,根据不同的检修等级对应不同的工序组合，根据附表二的到所动车信息，计算检修完这些动车需要的总时间。

二、问题分析

2.1 问题一

针对问题一，本文首先计算每个工序每个小时的处理能力 a=3,b=4,c=3.333，其次运用优先队列的方法，建立一个基于优先队列的模拟时间驱动的动车组模型，然后根据此模型，用 Visual Studio 编写程序，进行判断，若 a 工序空闲车间队列不为空，则进入 a 检测，完成后，若 b 工序有空余的车间，则进入 b 检测，完成后，若 c 工序有空余的车间，则进入 c 检测，车间 c 完成后进入一个虚拟车间 H，维修结束。

对于每一个工序，我们都要先判断该工序是否完成，是否能够进入下一个工序（即判断当前工序能不能出和能不能入下一个工序的问题）判断因 a 取决于 b，b 取决于 c,则对于是否可以进入下一车间，采用空闲车间队列是否不为空，如果不为空，动车不进入，不为满，动车进入，记录下动车进入与离开的时间，当超过最后一辆列车进站时间后都要判断 a、b、

c 三个车间是否为空（虚拟车间 h 用于接受维修成功的动车）如果为空，则维修完毕，输出每辆动车进入车间和离开车间的时间，得到维修动车的总耗时。

2.2 问题二

针对问题二，在问题一的基础上进行拓展，动车拥有四个类别 CRH2、CRH3、CRH5、CRH6，每个类别的动车在 a、b、c 三个车间的维修时间都不一致，利用**散列表**将不同类别的动车在 a、b、c 三个车间检修需要的时间影射到表中的一个位置来访问记录，加快查找的速度。

当动车进入车间后判断该动车的类别，根据动车类别得到该动车在 a、b、c 车间中的检修时间，然后记录下动车进入车间和离开车间的时间，对于是否可以进入下一车间，采用判断下一车间是否为满，如果为满，动车不进入，不为满，动车进入，记录下动车进入与离开的时间，每次进行该操作时都要判断 a、b、c 三个车间是否为空（虚拟车间 h 用于接受维修成功的动车）如果为空，则维修完毕，输出每辆动车进入车间和离开车间的时间，得到维修动车的总耗时。

2.3 问题三

针对问题三，首先与问题一不同的是，动车的属性新增了检修等级，与新的检修车间，不同的检修等级对应着不同的检修车间组合，一辆动车对应的检修等级和车间组合的关系，其次根据列车的行驶时间、历程和检修周期，动车组的检修被划分成不同检修等级 I~V，不同的检修等级对应不同的工序组合，利用二维数组来存储，根据检修等级映射到二维数组来获取对应的工序组合。然后根据问题一的模型，用 Visual Studio 编写程序。将分钟值时间转化为小时值时间，使用键-值（key-value）映射使每辆动车与所需的维修级别相对应，建立以 train（trains 数组的下标 m 即 m=0,1,2,3,4,5,6）为唯一键值，检修等级数组 maintenance_level 的行下标为对应值的映射，当第一维修工序不为空工序，则加入该维修工序等待队列中。根据问题一的维修顺序，得出检修完附表二这些列车需要的总时间。

三、模型假设

1、假设第一辆动车组抵达动车运用所时，所有检修车间都是空闲的，且车间之间的转换时间忽略不计。
2、假设 12 个小时开始零时刻就有一辆动车组到达动车所检修。
3、假设问题二中的当前时间为第一辆动车的到达时间即 00:16。
4、假设相同工序不同车间的耗费时间相同

四、符号说明

符号	符号说明
t_a	动车在 a 车间的检修时间
t_b	动车在 b 车间的检修时间
t_c	动车在 c 车间的检修时间
W_a	进入 a 车间的等待时间
W_b	进入 b 车间的等待时间
W_c	进入 c 车间的等待时间
E_a	离开 a 车间的时间
E_b	离开 b 车间的时间
E_c	离开 c 车间的时间
A_a	进入 a 车间对的时间
A_b	进入 b 车间对的时间
A_c	进入 c 车间对的时间
$R_{ai}^{f}(T)$	动车 f 在 T 时刻是否占用 a 车间
$R_{bi}^{f}(T)$	动车 f 在 T 时刻是否占用 b 车间
$R_{ci}^{f}(T)$	动车 f 在 T 时刻是否占用 c 车间

五、模型建立与求解

5.1 问题一模型的建立及求解

5.1.1 问题一分析

针对问题一，本文首先计算每个工序每个小时的处理能力 a=3,b=4,c=3.333，其次运用优先队列的方法，建立一个基于优先队列的模拟时间驱动的动车组模型，然后根据此模型，用 Visual Studio 编写程序，进行判断，若 a 工序空闲车间队列不为空，则进入 a 检测，完成后，若 b 工序有空余的车间，则进入 b 检测，完成后，若 c 工序有空余的车间，则进入 c 检测，车间 c 完成后进入一个虚拟车间 H，维修结束。

对于每一个工序，我们都要先判断该工序是否完成，是否能够进入下一个工序（即判断当前工序能不能出和能不能入下一个工序的问题）判断因 a 取决于 b，b 取决于 c,则对于是否可以进入下一车间，采用空闲车间队列是否不为空，如果不为空，动车不进入，不为满，动车进入，记录下动车进入与离开的时间，当超过最后一辆列车进站时间后都要判断 a、b、c 三个车间是否为空（虚拟车间 h 用于接受维修成功的动车）如果为空，则维修完毕，输出每辆动车进入车间和离开车间的时间，得到维修动车的总耗时。

5.1.2 基于优先队列的动车组检修算法

5.1.2.1 定义[1]

优先队列是一种基于堆的动态排序集合，常使用最小二叉堆为基础进行构造。利用其在数据结构上的优势，能够有 效组织节点并提高计算速度。普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先进先出的行为特征。通常采用堆数据结构来实现。

优先队列 H 支持如下操作:

$findMin(H)$ ：返回 H 中最小的元素。由于在优先队列中最小元素在队列首位，返回队列中第一个元素即可。此操作时间 复杂度为 $O(1)$ 。

$insert(H,a)$ ：向 H 中插入元素 a 。由于在插入后需要对优先队列进行维护，此操作时间复杂度为 $O(\log n)$ 。

$dec-key(H,a,x)$ ：将对象 a 的值降至 x。若 a 的值小于 x，则返回一个错误。此操作时间复杂度为 $O(\log n)$。

$extracMin(H)$ ：去掉并返回 H 中最小值的元素，即队列首元素，此操作时间复杂度为 $O(\log n)$。

优先队列主要应用于基于事件驱动的模型中，一般用于对操作系统中进程的管理。操作系统将进程按优先级或执行时间进行排序后，可每次从队列首位取得需要操作的进程。得益于其较高的排序效率，在队列中某个进程的优先级发生变化时，优先队列能快速地对队列进行重新排序。

优先队列里的小根堆形象化(弧线表示将堆顶元素移除除堆时各端点的变化)：

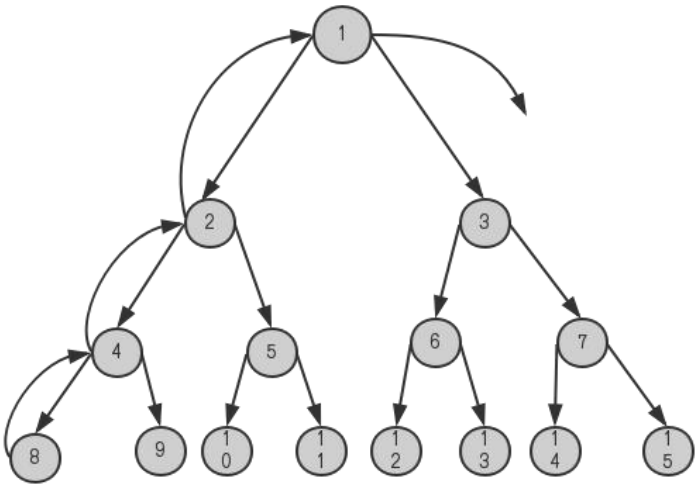

图 1 优先队列小根堆

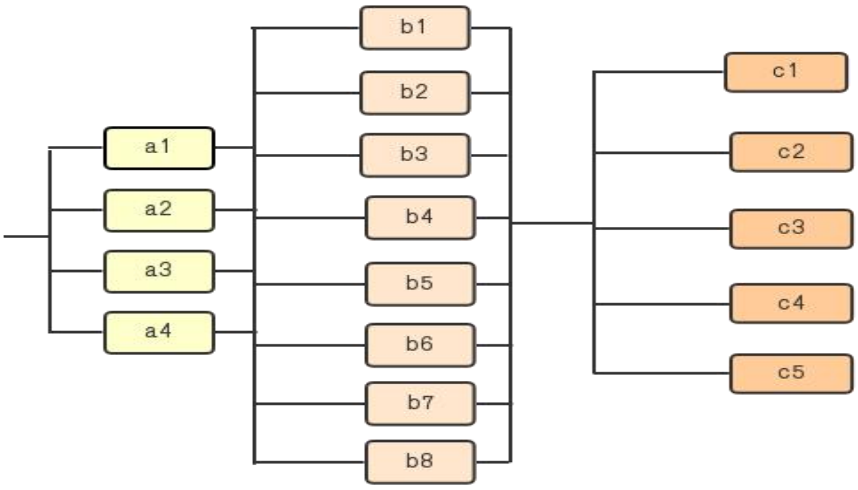

图 2 a、b、c 三个工序车间的示意图

5.1.2.2 模型的建立[3]

以动车组高级修检修所需时间最少为目标， 建立动车组优先队列模型如下 ：

$$\min H = e_t - b_t \tag{1}$$

$$s.t.$$

列车到达的间隔时间：

$$W_a \geq 15 \tag{2}$$

a 、 b 、 c 车间的检修时间：

$$E_a - A_a \geq t_a \tag{3}$$

$$E_b - A_b \geq t_b \tag{4}$$

$$E_c - A_c \geq t_c \tag{5}$$

动车进入 c 、 b 车间的检修开始时间：

$$A_c = E_b + W_c \tag{6}$$

$$A_b = E_a + W_b \tag{7}$$

a 、 b 、 c 三个工序的每个车间至多有一辆列车：

$$\sum_f R_{ai}^f(T) \leq 1 \tag{8}$$

$$\sum_f R_{bi}^f(T) \leq 1 \tag{9}$$

$$\sum_f R_{ci}^f(T) \leq 1 \tag{10}$$

每一辆列车在 T 时刻是否占用对应车间（1 为占用，0 为不占用）：

$$R_{ai}(T) = \begin{cases} 0 & , \quad T < A_a \\ 1 & , \quad A_a \leq T < E_a \\ 0 & , \quad T \geq E_a \end{cases} \tag{11}$$

$$R_{bi}(T)=\begin{cases}0 & , \quad T<A_b \\ 1 & , \quad A_b \le T<E_b \\ 0 & , \quad T \ge E_b\end{cases} \tag{12}$$

$$R_{ci}(T)=\begin{cases}0 & , \quad T<A_c \\ 1 & , \quad A_c \le T<E_c \\ 0 & , \quad T \ge E_c\end{cases} \tag{13}$$

其中，H 表示维修完所有这些动车组需要的总时间，e_t 表示最后一列动车进入车间的时间，b_t 表示第一动车进入车间的时间。

5.1.2.3 算法形式化描述

算法 1 基于优先队列以模拟时间驱动的动车组检修算法
输出：每辆动车的到达时间以及进入每个车间的时间和离开车间的时间，维修所有动车的耗时。
1）将车间队列、车辆数组、等待车辆数组初始化；将工序车间间队列编号，动车组编号，等待进入下一工序的动车编号；转化时间格式，将所有时间都转化成/h。
2）根据步长值为 15 min，建立 while 无限循环，并执行。
3）使用优先队列，将到达的动车加入动车队中，获取系统的时间，将系统时间转化为本地时间。
4）记录车辆停车时间，两辆车之间到达间隔时间，车辆到达时间的起始时间。
5）当前时间小于动车到达时间，则增加新动车进入维修队列 i，当前进入动车编号，进入等待进入工序 0 队列。
6) 当前时间超过动车到达时间判断所有车间是否为空。默认 flag 为 true，全部维修完毕，flag 为 false，未全部维修完成。如果 flag 为 false，且车间维修车辆的结束时间小于当前时间，将车辆出队并放入下一维修等待队列。如果 flag 为 true，当前时间大于车辆进入的时间（最后一辆动车到达第一工序的时间），判断三个工作队列是否为空，如为空，则车辆维修完毕，及 while 循环满足结束条件，退出循环，输出结果，程序结束。

程序运行结果：

表 1 a、b、c 车间前九辆动车组的开始时间和结束时间

编号	型号	到达时间	A车间编号	A开始	A结束	B车间编号	B开始	B结束	C车间编号	C开始	C结束
0	CRH0	15:15	1	15:15	16:15	1	16:15	18:15	1	18:15	19:45
1	CRH0	15:30	2	15:30	16:30	2	16:30	18:30	2	18:30	20:00
2	CRH0	15:45	3	15:45	16:45	3	16:45	18:45	3	18:45	20:15
3	CRH0	16:00	1	16:15	17:15	4	17:15	19:15	4	19:15	20:45
4	CRH0	16:15	2	16:30	17:30	5	17:30	19:30	5	19:30	21:00
5	CRH0	16:30	3	16:45	17:45	6	17:45	19:45	1	19:45	21:15
6	CRH0	16:45	1	17:15	18:15	7	18:15	20:15	2	20:15	21:45
7	CRH0	17:00	2	17:30	18:30	8	18:30	20:30	3	20:30	22:00
8	CRH0	17:15	3	17:45	18:45	1	18:45	20:45	4	20:45	22:15

由程序的结果可得在动车运用所 a、b、c 三道工序对应的不同车间中，12 个小时内每 15 分钟检修一辆车，共有 48 辆动车组，按照 a-b-c 的顺序进行检修，**检修完所有的动车组需要 20 时 30 分。**

5.1.2 问题一模型的检验

根据是实际情况，对得出结果数据进行检验，误差相对较小，模型通过检验

5.2 问题二模型的建立与求解

5.2.1 模型的建立

5.2.1.1 问题二的分析

针对问题二，在问题一的基础上进行拓展，动车拥有四个类别 CRH2、CRH3、CRH5、CRH6，每个类别的动车在 a、b、c 三个车间的维修时间都不一致，利用**散列表**将不同类别的动车在 a、b、c 三个车间检修需要的时间影射到表中的一个位置来访问记录，加快查找的速度。

当动车进入车间后判断该动车的类别，根据动车类别得到该动车在 a、b、c 车间中的检修时间，然后记录下动车进入车间和离开车间的时间，对于是否可以进入下一车间，采用判断下一车间是否为满，如果为满，动车不进入，不为满，动车进入，记录下动车进入与离开的时间，每次进行该操作时都要判断 a、b、c 三个车间是否为空（虚拟车间 h 用于接受维修成功的动车）如果为空，则维修完毕，打印每辆动车的检修时刻表，得到维修动车的总耗时。

5.2.1.2 散列表的定义

散列表(Hash table，也叫哈希表)，是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。公式为 $f(key) = key \bmod p(p \le m)$（key 表示键值即 CRH2 为 2 其余动车类别类似，p 为动车类别 CRH6 的键值+1 即 7）。

给定表 M，存在函数 f(key)，对任意给定的关键字值 key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表 M 为哈希(Hash)表，函数 f(key)为哈希(Hash) 函数。

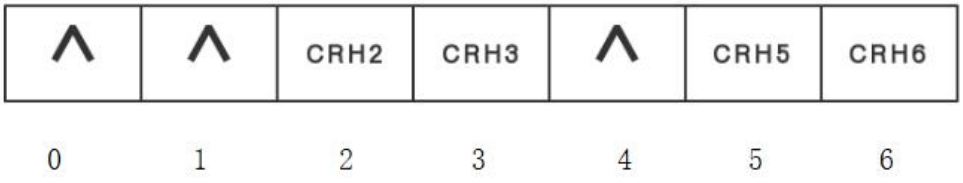

图 3 散列表

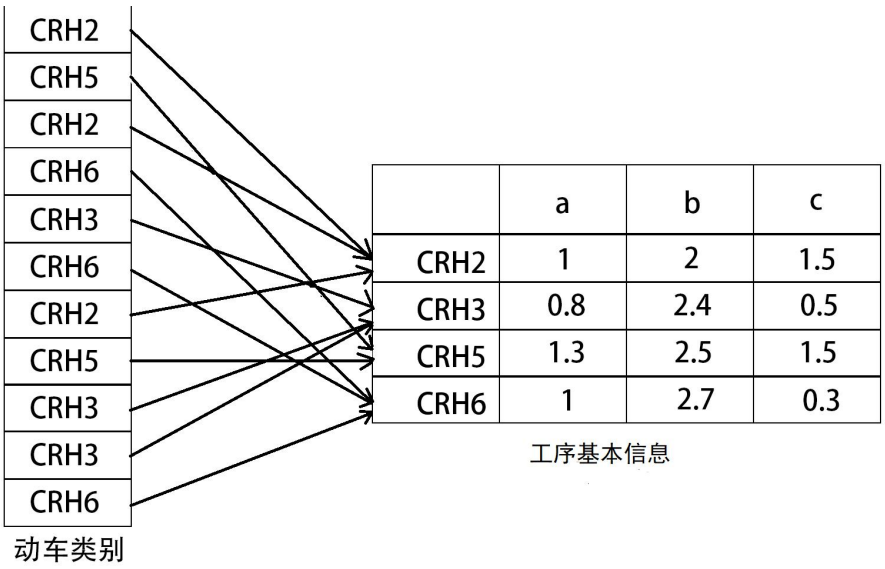

图 4 映射关系图

表 2 映射关系结果表

工序类别 动车类别	a	b	c
CHR2	1	2	1.5
CHR5	1.3	2.5	0.5
CHR2	1	2	1.5
CHR6	1	2.7	0.3
CHR3	0.8	2.4	0.5
CHR6	1	2.7	0.3
CHR2	1	2	1.5
CHR5	1.3	2.5	1.5
CHR3	0.8	2.4	0.5
CHR3	0.8	2.4	0.5
CHR6	1	2.7	0.3

5.2.1.3 算法形式化描述

输出：每辆动车的到达时间以及进入每个车间的时间和离开车间的时间，维修所有动车的耗时。

(1)将车间队列、车辆数组、等待车辆数组初始化；将工序车间间队列编号，动车组编号，等待进入下一工序的动车编号；动车类别与其对应的 a、b、c 三个车间的检修时间运用散列表形成与其对应的映射关系；转化时间格式，将所有时间都转化成 h:min。

(2)根据步长值为 1 min,建立 while 无限循环，并执行。

(3)使用优先队列，将到达的动车加入动车队中；

(4)记录车辆停车时间，两辆车之间到达间隔时间，车辆到达时间的起始时间。

(5)当前时间小于动车到达时间，则增加新动车进入维修队列 i,当前进入动车编号，进入等待进入工序 0 队列。

(6)当前时间超过动车到达时间判断所有车间是否为空。设置 flag 为 true,全部维修完毕，flag 为 false，未全部维修完成。如果 flag 为 false,且车间维修车辆的结束时间小于当前时间，将车辆出队并放入下一维修等待队列。如果 flag 为 true,车辆维修完毕，即 while 循环满足结束条件，退出循环，输出结果，程序结束。

程序输出结果：

表 3 问题二动车维修时刻表

编号	类别	到达时间	工序 a			工序 b			工序 c		
			车间编号	开始时间	结束时间	车间编号	开始时间	结束时间	车间编号	开始时间	结束时间
0	CRH2	0:16	1	0:16	1:16	1	1:16	3:16	1	3:16	4:46
1	CRH5	0:47	2	0:47	2:05	2	2:05	4:35	3	4:35	6:05
2	CRH2	1:22	3	1:22	2:22	3	2:22	4:22	2	4:22	5:52
3	CRH6	2:00	1	2:00	3:00	4	3:00	5:42	5	5:42	6:00
4	CRH3	2:21	2	2:21	3:09	5	3:09	5:33	4	5:33	6:03
5	CRH6	3:02	3	3:02	4:02	6	4:02	6:44	2	6:44	7:02
6	CRH2	3:31	1	3:31	4:31	7	4:31	6:31	1	6:31	8:01
7	CRH5	3:59	2	3:59	5:17	1	5:17	7:47	3	7:47	9:17
8	CRH3	4:01	3	4:02	4:50	8	4:50	7:14	5	7:14	7:44
9	CRH3	4:27	1	4:31	5:19	3	5:19	7:43	4	7:43	8:13
10	CRH6	5:09	3	5:09	6:09	2	6:09	8:51	2	8:51	9:09

根据附件中附表一到达动车运用所的动车信息，计算维修完这些动车的总时间是 9 时 1 分。（程序见附录二）

5.3 问题三模型的建立与求解

5.3.1 模型的建立

5.3.1.1 问题三的分析

针对问题三，首先与问题一不同的是，动车的属性新增了检修等级，与新的检修车间，不同的检修等级对应着不同的检修车间组合，一辆动车对应的检修等级和车间组合的关系，其次根据列车的行驶时间、历程和检修周期，动车组的检修被划分成不同检修等级 I~V，不同的检修等级对应不同的工序组合，利用二维数组来存储，根据检修等级映射到二维数组来获取对应的工序组合。

然后根据问题一的模型，用 Visual Studio 编写程序。将分钟值时间转化为小时值时间，使用键-值（key-value）映射使每辆动车与所需的维修级别相对应，建立以 train（trains 数组的下标 m 即 m=0,1,2,3,4,5,6）为唯一键值，检修等级数组 maintenance_level 的行下标为对应值的映射，当第一维修工序不为空工序，则加入该维修工序等待队列中。根据问题一的维修顺序，得出检修完附表二这些列车需要的总时间。

所得映射数组：

$$\begin{bmatrix} 1 & 2 & -1 & -1 & -1 & -1 \\ 1 & 2 & 3 & -1 & -1 & -1 \\ 1 & 2 & 3 & -1 & -1 & -1 \\ 1 & 3 & -1 & 4 & 5 & -1 \\ 1 & 2 & 3 & 4 & 5 & -1 \end{bmatrix} \tag{13}$$

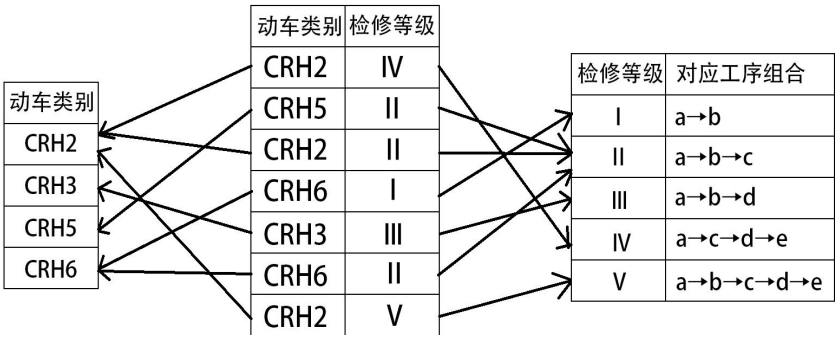

图 5 映射关系图

表 4 映射关系结果表

动车类别	工序				
	a	b	c	d	e
CRH2	1	-	1.5	4	7
CRH5	1.3	2.5	1.5	-	-
CRH2	1	2	1.5	-	-

CRH6	1	2.7	-	-	-
CRH3	0.8	2.4	-	-	-
CRH6	1	2.7	0.3	-	-
CRH2	1	2	1.5	4	7

然后根据问题一的模型，用 Visual Studio 编写程序。将分钟值时间转化为小时值时间。使用键-值（key-value）映射使每辆动车与所需的维修级别相对应。建立以 train（trains 数组的下标 m 即 m=0,1,2,3,4,5,6）为唯一键值。检修等级数组 maintenance_level 的行下标为对应值的映射，当第一维修工序不为空工序，则加入该维修工序等待队列中。

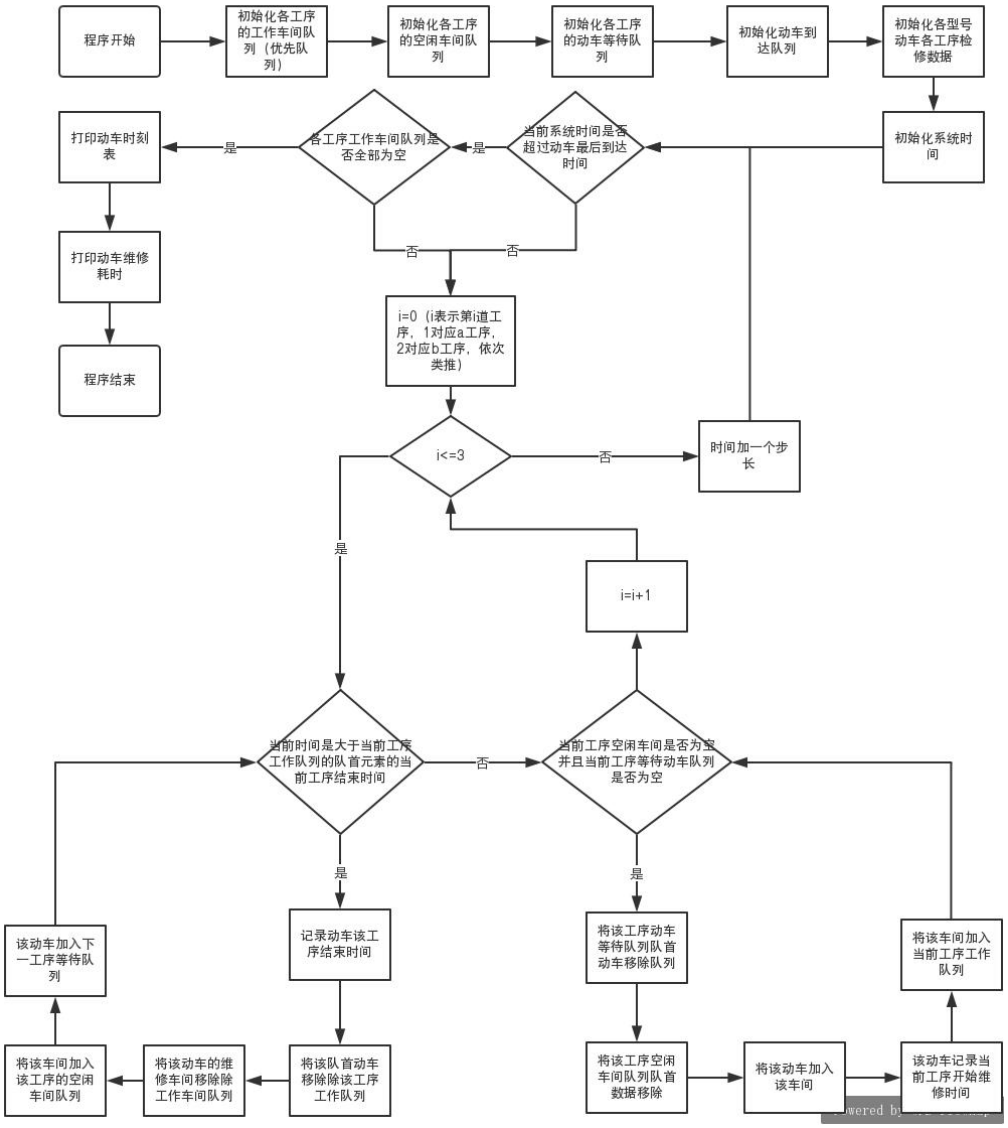

图 7 流程图 1（程序概述）

5.3.1.2 算法描述

输出：每辆动车的到达时间以及进入每个车间的时间和离开车间的时间,维修所有动车的耗时。

(1) 将车间队列、车辆数组、等待车辆数组初始化；将工序车间间队列编号，动车组编号，等待进入下一工序的动车编号；动车的编号（动车数组的下标:0、1、2、3、4、5、6）与检修类型数组的下标相形成映射，对于检修类型下的的工序组合则采用二维数组的形式存储，由检验类型数组的下标进行映射，获取对应的工序组合。

(2) 根据步长值为 1 min,建立 while 无限循环，并执行。

(3)使用优先队列，将到达的动车加入工作队列中；

(4)记录车辆到达时间，两辆车之间到达间隔时间，车辆到达时间的起始时间。

(5)当前时间小于动车到达时间，如果该动车的检修工序组合中存在该车间，则增加新动车进入维修队列 i,当前进入动车编号，进入等待进入指向队列，反之则不进入。

(6)当前时间超过动车到达时间判断所有车间是否为空。设置 flag 为 true,全部维修完毕，flag 为 false，未全部维修完成。如果 flag 为 false,且车间维修车辆的结束时间小于当前时间，将车辆出队并放入下一维修等待队列。如果 flag 为 true,车辆维修完毕，即 while 循环满足结束条件，退出循环，输出结果，程序结束。

表 5 问题三动车检修时刻表

编号	类别	到达时间	工序 a			工序 b			工序 c			工序 d			工序 e		
			车间编号	开始时间	结束时间	车间编号	开始时间	结束时间	车间编号	开始时间	结束时间	车间编号	开始时间	结束时间	车间编号	开始时间	结束时间
0	CRH2	0:16	1	0:16	1:16	-	-	-	1	1:16	2:46	1	2:46	6:46	1	6:46	13:46
1	CRH5	0:47	2	0:47	2:05	1	2:05	4:35	3	4:35	6:05	-	-	-	-	-	-
2	CRH2	1:22	3	1:22	2:22	2	2:22	4:22	2	4:22	5:52	-	-	-	-	-	-
3	CRH6	2:00	1	2:00	3:00	3	3:00	5:42	-	-	-	-	-	-	-	-	-
4	CRH3	2:21	2	2:21	3:09	4	3:09	5:33	-	-	-	2	5:33	10:21	-	-	-
5	CRH6	3:02	3	3:02	4:02	5	4:02	6:44	5	6:44	7:02	-	-	-	-	-	-
6	CRH2	3:31	1	3:31	4:31	6	4:31	6:31	4	6:31	8:01	3	8:01	12:01	2	12:01	19:01

增加检修等级难度，增加检修工序 d、e,根据不同的检修等级对应不同的工序组合，根据附表二的到所动车信息，计算检修完这些动车需要的总时间为 18 时 45 分

六、模型的评价与推广

6.1 模型的优点

本文对动车组检修时间问题进行分析，在此基础上，在保证所有检修任务均有安排的前提下，以动车组高级修检修所需时间最短为优化目标，建立动车组检修优先队列模型，设计该模型的求解算法。该模型和算法已在动车组检修管理信息系统中实现[3]，工程应用表明该算法求解效率高，可在短时间内获得动车组检修计划，大幅度提高检修计划的编制质量和效率。

6.2 模型的缺点

第一，在动车组检修优先队列模型中，本文没有考虑动车组检修时间的浪费，动车组每天上线运用担当交路终到动车所时间 分布不均，需检修动车组终到时间分布不均则会导致出现检修线空余的现象，例如一个四线检修库，如果夜间终到动车所检修车组时间为 19：30 的有 3 列，第 4 列终到入所时间为 20：30，则会造成第 4 条检修线空闲 1 个小时的冗余时间，检修能力产生浪费。 但实际上一列调车入库到位后需等待第二列到达入库才能开始作业，中间过程存在检修时间浪费。

第二，动车运用所站场布局对检修能力的制约。经过多个站段开展实地调研，发现尽头式检修库的检修作业能力比通过式检修库作业能力要低，其主要原因在于在检修库内车组检修作业完毕后通过式检修库可从一端调车出库，同时检修库另一端可以同步调车入库，减少调车出库后等待第二列车组调车入库的时间，提高调车作业效率。

6.3 模型的推广

降低动车组检备率，随着高速铁路设备完善和运营组织进一步成熟，以及线网密度快速扩张，可考虑适当降低动车组检备率，提高动车组运用效率；优化动车组出入库方式，尽可能统一为长编组动车组，提高动车组出入库能力，优化作业流程；优化动车所站场布局，建议动车运用所多采用通过式检修库布局，提高作业效率。该模型就可运用于我国高速铁路运输交通维修工程中。

七、参考文献

[1] 杨传印,黄玮,薛少聪,王劲松.基于优先队列的时变网络最短路径算法[J].计算机应用研究,2019,36(05):1403-1408.
[2] 武建平,何君礼,林柏梁,王辉,张旭辉,王忠凯.动车组高级修计划优化模型及算法研究[J].铁道学报,2019(07):1-9.
[3] 陈彦.动车组高级修检修计划编制模型与算法[J].铁路计算机应用,2013,22(01):22-24.
[4] 曹栋. 动车组检修模式研究[A]. 科技与企业杂志社、北京科技大学计算机与通信工程学院、北京科技大学土木与环境工程学院.科技与企业——企业科技创新与管理学术研讨会论文集（上）[C].科技与企业杂志社、北京科技大学计算机与通信工程学院、北京科技大学土木与环境工程学院:《科技与企业》编辑部,2016:1.
[5] CSDN 网.动车组维修 https://ask.csdn.net/questions/760696.2019.08.16
[6] CSDN 网.优先队列+模拟 https://blog.csdn.net/ZscDst/article/details/80159247.2019.08.16
[7] 韩中庆，数学建模方法及其应用，北京：高等教育出版社，2009.06

八、附录清单

1、附录一问题一的代码
2、附录二问题二的代码
3、附录三问题三的代码

附录一

```
// 动车 1.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
//

#include "pch.h"
#include <iostream>
#pragma warning (disable:4996)
#include<time.h>
#include<vector>
#include<queue>
#include<string>
using namespace std;
#define ARRIVE_TIME 15 //列车到站时间间隔（单位：分钟）
#define FACTORY_A_TIME 1*60 //工序 A 消耗时间 （单位:分钟）
#define FACTORY_B_TIME 2*60 //工序 B 消耗时间 （单位:分钟）
#define FACTORY_C_TIME 90 //工序 C 消耗时间（单位：分钟）
#define ALL_TIME 12*60 //工作总时间（单位：分钟）
#define START_TIME 0 //列车到达时间 （单位：分钟）

const int numbers = 3; //工序数

/**
 *自定义时间结构体
 *包含小时和分钟两种属性
 */
struct myTm {
	int hour; //小时
	int minute;//分钟
	/**
	 *初始化函数
	 *参数说明
	 *_hour：属性 hour
	 *_minute:属性 minute
	 */
	myTm(int _hour, int _minute) :hour(_hour), minute(_minute) {
		hour = hour + minute / 60;
		minute = minute % 60;
```

```
}
/**
 *构造函数
 */
myTm() {
	hour = 0;
	minute = 0;
}
/**
 *  比较函数
 *  和参数时间想比较返回较小者
 */
bool cmp(const myTm& time) {
	if (hour == time.hour) {
		return minute <= time.minute;
	}
	return hour < time.hour;
}



/*
 *重载运算符：+
 */
myTm operator+ (const myTm &time1) {
	myTm t(0, 0);
	t.minute = time1.minute + minute;
	t.hour = time1.hour + hour + t.minute / 60;
	t.minute = t.minute % 60;
	return t;
}

/*
 *重载运算符：-
 */
myTm operator- (const myTm &time2) {
	myTm t(0, 0);
	t.minute = minute - time2.minute;
	t.hour = hour - time2.hour;
	if (t.minute < 0) {
		t.minute = t.minute + 60;
		t.hour = t.hour - 1;
	}
	return t;
```

```
    }

    void print() {
        printf("    %02d:%02d", hour % 24, minute);
    }
};

//列车结构体
struct _Train {
    int model; //列车型号
    myTm arrive_time; //列车到达时间
    int factory_id[numbers]; //工序对应车间 id
    myTm working_start_time[numbers]; //工序开始时间
    myTm working_end_time[numbers]; //工序结束时间
    /**
     *构造函数，初始化列车型号和到站时间和工作时长
     */
    _Train(int _model, myTm _arrive_time)
        :model(_model), arrive_time(_arrive_time) {}
    /**
     *构造函数
     *初始化到站时间
     *型号默认为 0
     */
    _Train(myTm _arrive_time) :arrive_time(_arrive_time) {
        model = 0;
    };


    //列车维修时间表打印函数
    void print() {
        cout << "    CRH" << model;
        arrive_time.print();
        for (int i = 0; i < numbers; i++) {
            //cout<<setw(2) << factory_id[i];
            printf("          %02d      ", factory_id[i]);
            working_start_time[i].print(); working_end_time[i].print();
        }
        cout << endl;
    }
    //获取当前（cur）工序的工作时间：working_end_time - working_start_time
    myTm get_working_time(int cur) {
        return working_end_time[cur] - working_start_time[cur];
    }
```

```
};

//工序结构体
struct Factory {
	int train_id; //当前车间维修列车 id；
	myTm end_work_time;//结束工作的时间
	int id;//工序 id；

	/**
	 *构造函数
	 */
	Factory(int _id) :id(_id) {	}
	Factory() {	}

	//开始工作
	void start_work(myTm now_time, myTm working_time) {
		end_work_time = end_work_time + working_time;
		end_work_time = end_work_time + now_time;
	}

};
bool operator< (Factory f1, Factory f2) {
	return !f1.end_work_time.cmp(f2.end_work_time);
}

int main() {
	int factory_number[numbers] = { 3,8,5 };//车间的数量
	queue<int> free_factory_ids[numbers]; //空闲车间 id；
	//使用优先队列
	priority_queue<Factory> factories[3]; //三道工序的车间
	//初始化三道工序的车间数量，当前所有车间全部处于空闲状态
	for (int i = 0; i < numbers; i++) {
		//构造当前工序车间的 id
		for (int j = 1; j <= factory_number[i]; j++) {
			free_factory_ids[i].push(j);
		}
	}

	/**
	 *初始化 trains 队列
	 *将到达的列车加入列车队列中
	 */
	vector<_Train> trains;//列车安排时刻表队列
	queue<int> waitting_trains_id[numbers + 1]; //等待进入下一工序的列车的 id
```

```
    time_t now = time(0);//获取系统时间，从 1900/0/1 0:0:0 开始的秒数
    tm* local_time = localtime(&now);//将系统时间转换为本地时间
    myTm now_time(local_time->tm_hour, local_time->tm_min);//将本地时间获取小时和分钟
    myTm end_time(0, ALL_TIME);//车辆停止到达时间
    end_time = end_time + now_time;
    myTm step_time(0, ARRIVE_TIME);//两辆车之间到达间隔时间
    myTm start_time(0, START_TIME);//车辆到达时间的起始时间
    start_time = now_time; //开始时间等于当前时间
    //初始话车间工作时间
    myTm factory_working_time[numbers];
    factory_working_time[0].minute = FACTORY_A_TIME % 60;
    factory_working_time[0].hour = FACTORY_A_TIME / 60;
    factory_working_time[1].minute = FACTORY_B_TIME % 60;
    factory_working_time[1].hour = FACTORY_B_TIME / 60;
    factory_working_time[2].minute = FACTORY_C_TIME % 60;
    factory_working_time[2].hour = FACTORY_C_TIME / 60;


    myTm total_time(0, 0);//总计耗时
    myTm _now = now_time; //当前时间


    /**
     *当前队列不空的情况下继续操作
     */
    while (true) {

        //当前时间小于列车停止到达时间这继续添加进入列车队列中。并将该车次 id 加入等待队列 1 中
        if (_now.cmp(end_time)) {
            _Train train(_now);//当前时间小于车辆停止到达时间，增加新的列车进入维修 i 队列
            trains.push_back(train);
            waitting_trains_id[0].push(trains.size() - 1); //当前进入列车 id 进入等待进入工序 0 队列
        }
        else {//当前时间超过列车到达时间判断所有车间是否全部为空
            bool allFactoryEmpty = true;//设置标志 flag 为 true：全部维修完毕，false：未全部维修完成
            for (int i = 0; i < numbers; i++) {
                if (!factories[i].empty()) {
                    allFactoryEmpty = false;
                    break;
```

```
                }
            }
            //如果全部车间为空，车辆维修完毕退出程序
            if (allFactoryEmpty) {
                total_time = total_time - step_time;
                break;
            }
        }

        //for 循环 numbers 次工序
        for (int i = 0; i < numbers; i++) {
            //当前工序车间的队头元素
            Factory f;
            if (!factories[i].empty()) {
                f = factories[i].top();
            }
            //当前车间非空且车间维修车辆的结束时间小于当前时间，则将当前车辆出队并放入下一维修等待队列
            while (!factories[i].empty() && f.end_work_time.cmp(_now)) {

                Factory factory = factories[i].top(); //获取当前队首元素
                factories[i].pop(); //出队
                waitting_trains_id[i + 1].push(factory.train_id); //加入下一工序等待环节
                free_factory_ids[i].push(factory.id);//添加进入当前车间队列
                if (!factories[i].empty()) {
                    f = factories[i].top();
                }
            }
            //当前工序空闲车间不为空并且当前工序等待列车不为空
            while (!free_factory_ids[i].empty() && !waitting_trains_id[i].empty()) {
                Factory factory;
                factory.id = free_factory_ids[i].front(); //当前车间 id 来自空闲车间
                free_factory_ids[i].pop();//空闲车间出队一个
                factory.train_id = waitting_trains_id[i].front(); //得到当前等待列车 id
                waitting_trains_id[i].pop();//等待列车出队一辆
                trains[factory.train_id].working_start_time[i] = _now; //添加当前工序结束时间
                trains[factory.train_id].working_end_time[i] = _now + factory_working_time[i]; //添加当前工序结束时间
                factory.start_work(_now, factory_working_time[i]); //车间开始工作

                factories[i].push(factory); //车间进入工作队列
                trains[factory.train_id].factory_id[i] = factory.id; //添加当前工序车间
```

```
                }
            }
            total_time = total_time + step_time; //总消耗时间增加一个列车到达间隔时间
            _now = _now + step_time; //当前时间增加一个列车到达间隔时间
        }
        cout << "编号  类别  到达时间  A 车间编号  A 开始  A 结束     B 车间编号     B 开始  B 结束       C 车间编号  C 开始  C 结束" << endl;
        for (int i = 0; i < trains.size(); i++) {
            printf(" %02d", i);
            trains[i].print();
        }
        //total_time = _now - start_time;
        //打印耗时信息
        cout << "维修所有列车耗时共计：" << total_time.hour << "时" << total_time.minute << "分" << endl;
        system("pause"); //控制台暂停
        return 0;
}
```

编号	类别	到达时间	A车间编号	A开始	A结束	B车间编号	B开始	B结束	C车间编号	C开始	C结束
00	CRH0	14:18	01	14:18	15:18	01	15:18	17:18	01	17:18	18:48
01	CRH0	14:33	02	14:33	15:33	02	15:33	17:33	02	17:33	19:03
02	CRH0	14:48	03	14:48	15:48	03	15:48	17:48	03	17:48	19:18
03	CRH0	15:03	01	15:18	16:18	04	16:18	18:18	04	18:18	19:48
04	CRH0	15:18	02	15:33	16:33	05	16:33	18:33	05	18:33	20:03
05	CRH0	15:33	03	15:48	16:48	06	16:48	18:48	01	18:48	20:18
06	CRH0	15:48	01	16:18	17:18	07	17:18	19:18	02	19:18	20:48
07	CRH0	16:03	02	16:33	17:33	08	17:33	19:33	03	19:33	21:03
08	CRH0	16:18	03	16:48	17:48	01	17:48	19:48	04	19:48	21:18
09	CRH0	16:33	01	17:18	18:18	02	18:18	20:18	05	20:18	21:48
10	CRH0	16:48	02	17:33	18:33	03	18:33	20:33	01	20:33	22:03
11	CRH0	17:03	03	17:48	18:48	04	18:48	20:48	02	20:48	22:18
12	CRH0	17:18	01	18:18	19:18	05	19:18	21:18	03	21:18	22:48
13	CRH0	17:33	02	18:33	19:33	06	19:33	21:33	04	21:33	23:03
14	CRH0	17:48	03	18:48	19:48	07	19:48	21:48	05	21:48	23:18
15	CRH0	18:03	01	19:18	20:18	08	20:18	22:18	01	22:18	23:48
16	CRH0	18:18	02	19:33	20:33	01	20:33	22:33	02	22:33	00:03
17	CRH0	18:33	03	19:48	20:48	02	20:48	22:48	03	22:48	00:18
18	CRH0	18:48	01	20:18	21:18	03	21:18	23:18	04	23:18	00:48
19	CRH0	19:03	02	20:33	21:33	04	21:33	23:33	05	23:33	01:03
20	CRH0	19:18	03	20:48	21:48	05	21:48	23:48	01	23:48	01:18
21	CRH0	19:33	01	21:18	22:18	06	22:18	00:18	02	00:18	01:48
22	CRH0	19:48	02	21:33	22:33	07	22:33	00:33	03	00:33	02:03
23	CRH0	20:03	03	21:48	22:48	08	22:48	00:48	04	00:48	02:18
24	CRH0	20:18	01	22:18	23:18	01	23:18	01:18	05	01:18	02:48
25	CRH0	20:33	02	22:33	23:33	02	23:33	01:33	01	01:33	03:03
26	CRH0	20:48	03	22:48	23:48	03	23:48	01:48	02	01:48	03:18
27	CRH0	21:03	01	23:18	00:18	04	00:18	02:18	03	02:18	03:48
28	CRH0	21:18	02	23:33	00:33	05	00:33	02:33	04	02:33	04:03
29	CRH0	21:33	03	23:48	00:48	06	00:48	02:48	05	02:48	04:18
30	CRH0	21:48	01	00:18	01:18	07	01:18	03:18	01	03:18	04:48
31	CRH0	22:03	02	00:33	01:33	08	01:33	03:33	02	03:33	05:03
32	CRH0	22:18	03	00:48	01:48	01	01:48	03:48	03	03:48	05:18
33	CRH0	22:33	01	01:18	02:18	02	02:18	04:18	04	04:18	05:48
34	CRH0	22:48	02	01:33	02:33	03	02:33	04:33	05	04:33	06:03
35	CRH0	23:03	03	01:48	02:48	04	02:48	04:48	01	04:48	06:18
36	CRH0	23:18	01	02:18	03:18	05	03:18	05:18	02	05:18	06:48
37	CRH0	23:33	02	02:33	03:33	06	03:33	05:33	03	05:33	07:03
38	CRH0	23:48	03	02:48	03:48	07	03:48	05:48	04	05:48	07:18
39	CRH0	00:03	01	03:18	04:18	08	04:18	06:18	05	06:18	07:48
40	CRH0	00:18	02	03:33	04:33	01	04:33	06:33	01	06:33	08:03
41	CRH0	00:33	03	03:48	04:48	02	04:48	06:48	02	06:48	08:18
42	CRH0	00:48	01	04:18	05:18	03	05:18	07:18	03	07:18	08:48
43	CRH0	01:03	02	04:33	05:33	04	05:33	07:33	04	07:33	09:03
44	CRH0	01:18	03	04:48	05:48	05	05:48	07:48	05	07:48	09:18
45	CRH0	01:33	01	05:18	06:18	06	06:18	08:18	01	08:18	09:48
46	CRH0	01:48	02	05:33	06:33	07	06:33	08:33	02	08:33	10:03
47	CRH0	02:03	03	05:48	06:48	08	06:48	08:48	03	08:48	10:18
48	CRH0	02:18	01	06:18	07:18	01	07:18	09:18	04	09:18	10:48

维修所有列车耗时共计：20时30分

附录二

```
// 动车 第二题代码.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
//

#include "pch.h"
#include <iostream>
#include<queue>
#include<vector>
#pragma warning(disable :4996)

using namespace std;
#define STEP_TIME 1 //列车到站时间间隔（单位：分钟）
#define CRH2 2 //CRH2 火车型号
#define CRH3 3 //CRH3 火车型号
#define CRH5 5 //CRH5 火车型号
#define CRH6 6 //CRH6 火车型号
const int arriving_train_numbers = 11; //将要进站火车数量
const int train_models = 4; //列车类型数量
const int numbers = 3; //工序数
const int max_train_model_no = 6; //最大列车类型编号
/**
 *自定义时间结构体
 *包含小时和分钟两种属性
 */
struct myTm {
	int hour; //小时
	int minute;//分钟
	/**
	 *初始化函数
	 *参数说明
	 *_hour：属性 hour
	 *_minute:属性 minute
	 */
	myTm(int _hour, int _minute) :hour(_hour), minute(_minute) {
		hour = hour + minute / 60;
		minute = minute % 60;
	}
	/**
	 *构造函数
	 */
	myTm() {
		hour = 0;
		minute = 0;
	}
```

```
    /**
     * 比较函数
     * 和参数时间想比较返回较小者
     */
    bool cmp(const myTm& time) {
        if (hour == time.hour) {
            return minute <= time.minute;
        }
        return hour < time.hour;
    }
    //根据分钟制时间单位转换为小时制
    void setTime(int _minute) {
        hour = _minute / 60;
        minute = _minute % 60;
    }
    /*
     *重载运算符：+
     */
    myTm operator+ (const myTm &time1) {
        myTm t(0, 0);
        t.minute = time1.minute + minute;
        t.hour = time1.hour + hour + t.minute / 60;
        t.minute = t.minute % 60;
        return t;
    }

    /*
     *重载运算符：-
     */
    myTm operator- (const myTm &time2) {
        myTm t(0, 0);
        t.minute = minute - time2.minute;
        t.hour = hour - time2.hour;
        if (t.minute < 0) {
            t.minute = t.minute + 60;
            t.hour = t.hour - 1;
        }
        return t;
    }

    void print() {
        printf("    %02d:%02d", hour % 24, minute);
    }
};
```

```
//列车结构体
struct _Train {
	int model; //列车型号
	myTm arrive_time; //列车到达时间
	int factory_id[numbers]; //工序对应车间 id
	myTm working_start_time[numbers]; //工序开始时间
	myTm working_end_time[numbers]; //工序结束时间
	/**
	 *构造函数，初始化列车型号和到站时间和工作时长
	 */
	_Train(int _model, myTm _arrive_time)
		:model(_model), arrive_time(_arrive_time) {}
	/**
	 *构造函数
	 *初始化到站时间
	 *型号默认为 0
	 */
	_Train(myTm _arrive_time) :arrive_time(_arrive_time) {
		model = 0;
	};


	//列车维修时间表打印函数
	void print() {
		cout << "    CRH" << model;
		arrive_time.print();
		for (int i = 0; i < numbers; i++) {
			//cout<<setw(2) << factory_id[i];
			printf("          %02d      ", factory_id[i]);
			working_start_time[i].print(); working_end_time[i].print();
		}
		cout << endl;
	}
	//获取当前（cur）工序的工作时间：working_end_time - working_start_time
	myTm get_working_time(int cur) {
		return working_end_time[cur] - working_start_time[cur];
	}
};


//工序结构体
struct Factory {
	int train_id; //当前车间维修列车 id；
```

```
    myTm end_work_time;//结束工作的时间
    int id;//工序 id；

    /**
     *构造函数
     */
    Factory(int _id) :id(_id) {     }
    Factory() {     }

    //开始工作
    void start_work(myTm now_time, myTm working_time) {
        end_work_time = end_work_time + working_time;
        end_work_time = end_work_time + now_time;
    }

};
bool operator< (Factory f1, Factory f2) {
    return !f1.end_work_time.cmp(f2.end_work_time);
}

int main() {
    /**
     * 初始化车间
     */
    int factory_number[numbers] = { 3,8,5 };//车间的数量
    queue<int> free_factory_ids[numbers]; //空闲车间 id；
    //初始化三道工序的车间数量，当前所有车间全部处于空闲状态
    for (int i = 0; i < numbers; i++) {
        //构造当前工序车间的 id
        for (int j = 1; j <= factory_number[i]; j++) {
            free_factory_ids[i].push(j);
        }
    }
    /**
     *初始化各工序等待列车队列
     */
    queue<int> waitting_trains_id[numbers + 1];//等待列车的 id

    /**
      * 初始化各工序各车间维修队列（优先队列）
      */
    priority_queue<Factory> factories[3]; //三道工序的车间优先队列

    /**
```

```
     * 初始化各类型列车的各道工序的维修时间
     */
     //列车型号数组
    int train_model[train_models] = { CRH2,CRH3,CRH5,CRH6 };

    //各型号列车工作时间（分钟制）
    int working_time[train_models][numbers] = {
        {60,120,90}, //CRH2
        {48,144,30}, //CRH3
        {78,150,90}, //CRH5
        {60,162,18}    //CRH6
    };
    /**
     * 使用简单对应键值关系建立散列表
     * 散列方程为 列车型号工序时间= 列车型号值*1
     */
     //各型号列车工作时间散列表（分钟制）
    int working_time_hash[max_train_model_no + 1][numbers];
    //将各型号列车工作时间存放到对应散列表
    for (int i = 0; i < train_models; i++) {
        for (int j = 0; j < numbers; j++) {
            working_time_hash[train_model[i]][j] = working_time[i][j];
        }
    }

    /**
      * 初始化即将维修列车的类型、到达时间
      */
      //初始化即将到达列车的型号
    int arring_train_model[arriving_train_numbers] =
{ CRH2,CRH5,CRH2,CRH6,CRH3,CRH6,CRH2,CRH5,CRH3,CRH3,CRH6 };
    //初始话即将到达的列车的时间（分钟制）
    int arriving_train_times_minutes[arriving_train_numbers] =
{ 16,47,82,120,141,182,211,239,241,267,309 };
    //初始化即将到达的列车的时间（小时制）
    myTm arriving_train_times_hours[arriving_train_numbers];
    //将分钟值时间转化为小时值时间
    for (int i = 0; i < arriving_train_numbers; i++) {
        arriving_train_times_hours[i].setTime(arriving_train_times_minutes[i]);
    }
    vector<_Train> trains; //即将到达的列车数组
    //使用即将到达列车的型号、到达时间、各工序时间初始化即将到达列车数组，
    for (int i = 0; i < arriving_train_numbers; i++) {
        _Train train(arring_train_model[i], arriving_train_times_hours[i]); //创建列车
```

```
        trains.push_back(train); //将列车存入队列中
}

/**
 * 初始化时间参数
 */
myTm first_arrived_time; //第一辆列车到达时间
myTm last_arrived_time;//最后一辆列车到达时间
if (trains.size() > 0) {
        first_arrived_time = trains[0].arrive_time;
        last_arrived_time = trains[trains.size() - 1].arrive_time;
}
//当前时间假设为第一辆列车的到达时间
myTm now_time = first_arrived_time;//当前时间
//程序开始时间为第一辆列车到达时间
myTm start_time = last_arrived_time; //程序开始时间
myTm step_time(0, STEP_TIME);//时间流逝步长为 1 分钟
//总计时间消耗
myTm total_time(0,0);
int cur = 0; //当前等待加入维修队列的列车
while (true) {
        /**
          *判断当前时间大于最后一辆列车的到达时间之
          *如果所有列车维修工厂工作队列都为空
          *那么维修工作全部完成
          *退出程序
          */
        if (!now_time.cmp(last_arrived_time))//当前时间大于最后一辆列车进入的时间时
        {
                /**
                  * 设置标志 allFactoryEmpty bool 类型
                  * true:全部维修完毕
                  * false：未全部维修完成
                  * 默认为为 true
                  */
                bool allFactoryEmpty = true;
                //遍历全部工厂队列，出现不为空的工作工厂队列则表示维修工作未完成
                for (int i = 0; i < numbers; i++) {
                        if (!factories[i].empty()) {
                                allFactoryEmpty = false;
                                break;
                        }
                }
                //如果全部车间为空，车辆维修完毕退出程序
```

```
            if (allFactoryEmpty) {
                total_time = total_time-step_time;
                now_time = now_time - step_time;
                break;
            }
        }
        /** 当前时间到达列车进站时间
          * 该列车进入 a 工序等待队列
          * 等待进站列车序号自增 1
          */
        if (cur < trains.size() && now_time.hour == trains[cur].arrive_time.hour &&
now_time.minute == trains[cur].arrive_time.minute) {
            waitting_trains_id[0].push(cur);
            cur++;
        }


        //for 循环 numbers 次工序
        for (int i = 0; i < numbers; i++) {
            //当前工序车间的队头元素
            Factory f;
            if (!factories[i].empty()) {
                f = factories[i].top();
            }
            //当前车间非空且车间维修车辆的结束时间小于当前时间，则将当前车辆出队
并放入下一维修等待队列
            while (!factories[i].empty() && f.end_work_time.cmp(now_time)) {
                Factory factory = factories[i].top(); //获取当前队首元素
                factories[i].pop(); //出队
                waitting_trains_id[i + 1].push(factory.train_id); //加入下一工序等待环节
                free_factory_ids[i].push(factory.id);//添加进入当前车间队列
                if (!factories[i].empty()) {
                    f = factories[i].top();
                }
            }
            //当前工序空闲车间不为空并且当前工序等待列车不为空
            while (!free_factory_ids[i].empty() && !waitting_trains_id[i].empty()) {
                Factory factory;
                factory.id = free_factory_ids[i].front(); //当前车间 id 来自空闲车间
                free_factory_ids[i].pop();//空闲车间出队一个
                factory.train_id = waitting_trains_id[i].front(); //得到当前等待列车 id
                waitting_trains_id[i].pop();//等待列车出队一辆
                trains[factory.train_id].working_start_time[i] = now_time; //添加当前工序结
束时间
```

```
                    myTm working_time;//当前工序的工作时间
                    //当前工序的工作时间来自当前列车类型工作时间散列表中对应的值
                    working_time.setTime(working_time_hash[trains[factory.train_id].model][i]);
                    trains[factory.train_id].working_end_time[i] = now_time + working_time; //添加当前工序结束时间
                    factory.start_work(now_time, working_time); //车间开始工作

                    factories[i].push(factory); //车间进入工作队列
                    trains[factory.train_id].factory_id[i] = factory.id; //添加当前工序车间

                }
            }
            total_time = total_time + step_time; //总消耗时间增加一个列车到达间隔时间
            now_time = now_time + step_time; //当前时间增加一个列车到达间隔时间
        }
        cout << "编号  类别  到达时间  A 车间编号  A 开始    A 结束     B 车间编号  B 开始    B 结束     C 车间编号  C 开始    C 结束" << endl;
        for (int i = 0; i < trains.size(); i++) {
            printf(" %02d", i);
            trains[i].print();
        }
        //total_time = now_time - start_time;
        //打印耗时信息
        cout << "维修所有列车耗时共计：" << total_time.hour << "时" << total_time.minute << "分" << endl;
        system("pause"); //控制台暂停
        return 0;
}
// 运行程序: Ctrl + F5  或调试  >“开始执行(不调试)”菜单
// 调试程序: F5  或调试  >“开始调试”菜单

// 入门提示:
//    1. 使用解决方案资源管理器窗口添加/管理文件
//    2. 使用团队资源管理器窗口连接到源代码管理
//    3. 使用输出窗口查看生成输出和其他消息
//    4. 使用错误列表窗口查看错误
//    5. 转到“项目”>“添加新项”以创建新的代码文件，或转到“项目”>“添加现有项”以将现有代码文件添加到项目
//    6. 将来，若要再次打开此项目，请转到“文件”>“打开”>“项目”并选择  .sln  文件
```

附录三

```
#include<stdio.h>
#include<iostream>
```

```
#include<vector>
#include<queue>
#pragma warning(disable :4996)
using namespace std;
#define STEP_TIME 1 //动车到站时间间隔（单位：分钟）
#define CRH2 2 //CRH2  动车型号
#define CRH3 3 //CRH3  动车型号
#define CRH5 5 //CRH5  动车型号
#define CRH6 6 //CRH6  动车型号
const int arriving_train_numbers = 7; //将要进站动车数量
const int train_models = 4; //动车类型数量
const int numbers = 5; //工序数
const int max_train_model_no = 6; //最大动车类型编号
const int maintenance_level_numbers = 5; //维修级别数量
/**
 *自定义时间结构体
 *包含小时和分钟两种属性
 */
struct myTm {
	int hour; //小时
	int minute;//分钟
	/**
	 *初始化函数
	 *参数说明
	 *_hour：属性 hour
	 *_minute:属性 minute
	 */
	myTm(int _hour, int _minute) :hour(_hour), minute(_minute) {
		hour = hour + minute / 60;
		minute = minute % 60;
	}
	/**
	 *构造函数
	 */
	myTm() {
		hour = 0;
		minute = 0;
	}
	/**
	 * 比较函数
	 * 和参数时间想比较返回较小者
	 */
	bool cmp(const myTm& time) {
		if (hour == time.hour) {
```

```
            return minute <= time.minute;
        }
        return hour < time.hour;
    }
    //根据分钟制时间单位转换为小时制
    void setTime(int _minute) {
        hour = _minute / 60;
        minute = _minute % 60;
    }
    /*
     *重载运算符：+
     */
    myTm operator+ (const myTm &time1) {
        myTm t(0, 0);
        t.minute = time1.minute + minute;
        t.hour = time1.hour + hour + t.minute / 60;
        t.minute = t.minute % 60;
        return t;
    }

    /*
     *重载运算符：-
     */
    myTm operator- (const myTm &time2) {
        myTm t(0, 0);
        t.minute = minute - time2.minute;
        t.hour = hour - time2.hour;
        if (t.minute < 0) {
            t.minute = t.minute + 60;
            t.hour = t.hour - 1;
        }
        return t;
    }

    void print() {
        printf("    %02d:%02d", hour % 24, minute);
    }
};

//动车结构体
struct _Train {
    int model; //动车型号
    myTm arrive_time; //动车到达时间
    int factory_id[numbers]; //工序对应车间 id
```

```
    myTm working_start_time[numbers]; //工序开始时间
    myTm working_end_time[numbers]; //工序结束时间
    /**
     *构造函数，初始化动车型号和到站时间和工作时长
     */
    _Train(int _model, myTm _arrive_time)
        :model(_model), arrive_time(_arrive_time) {
        memset(factory_id, -1, sizeof factory_id);
    }
    /**
     *构造函数
     *初始化到站时间
     *型号默认为 0
     */
    _Train(myTm _arrive_time) :arrive_time(_arrive_time) {
        model = 0;
        memset(factory_id, -1, sizeof factory_id);
    };


    //动车维修时间表打印函数
    void print() {
        cout << "    CRH" << model;
        arrive_time.print();
        for (int i = 0; i < numbers; i++) {
            //cout<<setw(2) << factory_id[i];
            printf("      %02d    ", factory_id[i]);
            working_start_time[i].print(); working_end_time[i].print();
        }
        cout << endl;
    }
    //获取当前（cur）工序的工作时间：working_end_time - working_start_time
    myTm get_working_time(int cur) {
        return working_end_time[cur] - working_start_time[cur];
    }
};


//工序结构体
struct Factory {
    int train_id; //当前车间维修动车 id；
    myTm end_work_time;//结束工作的时间
    int id;//工序 id；
```

```
    /**
     *构造函数
     */
    Factory(int _id) :id(_id) {      }
    Factory() {     }

    //开始工作
    void start_work(myTm now_time, myTm working_time) {
        end_work_time = end_work_time + working_time;
        end_work_time = end_work_time + now_time;
    }

};

bool operator< (Factory f1, Factory f2) {
    return !f1.end_work_time.cmp(f2.end_work_time);
}

int main() {
    /**
     * 初始化车间
     */
    int factory_number[numbers] = { 3,8,5,3,2 };//车间的数量
    queue<int> free_factory_ids[numbers]; //空闲车间 id；
    //初始化三道工序的车间数量，当前所有车间全部处于空闲状态
    for (int i = 0; i < numbers; i++) {
        //构造当前工序车间的 id
        for (int j = 1; j <= factory_number[i]; j++) {
            free_factory_ids[i].push(j);
        }
    }
    /**
     *初始化各工序等待动车队列
     */
    queue<int> waitting_trains_id[numbers + 1];//等待动车的 id

    /**
      * 初始化各工序各车间维修队列（优先队列）
      */
    priority_queue<Factory> factories[numbers]; //工序的车间优先队列

    /**
     * 初始化各类型动车的各道工序的维修时间
     */
```

```
 //动车型号数组
int train_model[train_models] = { CRH2,CRH3,CRH5,CRH6 };

//各型号动车工作时间（分钟制）
int working_time[train_models][numbers] = {
    {60,120,90,240,420}, //CRH2
    {48,144,30,288,390}, //CRH3
    {78,150,90,180,390}, //CRH5
    {60,162,18,300,420}   //CRH6
};
/**
 * 使用简单对应键值关系建立散列表
 * 散列方程为 动车型号工序时间= 动车型号值*1
 */
 //各型号动车工作时间散列表（分钟制）
int working_time_hash[max_train_model_no + 1][numbers];
//将各型号动车工作时间存放到对应散列表
for (int i = 0; i < train_models; i++) {
    for (int j = 0; j < numbers; j++) {
        working_time_hash[train_model[i]][j] = working_time[i][j];
    }
}
/**建立不同维修级别工序之间顺序关系
  * 将维修级别信息存储在一个二维数组中
  * 行：表示不同的级别第零行对应维修级别 I，第一行对应维修级别二，依次类推至第 4 行对应维修级别 V
  * 列：表示维修工序，第一列表示工序 a，第二列表示工序 b，依次类推至第六列对应工序 e，
  * 每行每列的值初始化为-1。
  * 当第 i 行，第 j 列的值指向下一工序的列下标，第零列指向对应维修级别的第一维修工序
  * 某一行的值为默认值时，表示该工序不在当前维修级别中或在当前维修级别中的最后一道维修工序
  */
int maintenance_level[maintenance_level_numbers][numbers + 1]{
    {1,2,-1,-1,-1,-1},
    {1,2,3,-1,-1,-1},
    {1,2,4,-1,-1,-1},
    {1,3,-1,4,5,-1},
    {1,2,3,4,5,-1}
};
/**
  * 初始化即将维修动车的类型、到达时间
  */
```

```
    //初始化即将到达动车的型号
    int arring_train_model[arriving_train_numbers] = { CRH2,CRH5,CRH2,CRH6,CRH3,CRH6,CRH2 };
    //初始话即将到达的动车的时间（分钟制）
    int arriving_train_times_minutes[arriving_train_numbers] = { 16,47,82,120,141,182,211 };
    //初始化即将到达的动车的时间（小时制）
    myTm arriving_train_times_hours[arriving_train_numbers];
    //将分钟值时间转化为小时值时间
    for (int i = 0; i < arriving_train_numbers; i++) {
        arriving_train_times_hours[i].setTime(arriving_train_times_minutes[i]);
    }
    vector<_Train> trains; //即将到达的动车数组
    //使用即将到达动车的型号、到达时间、各工序时间初始化即将到达动车数组，
    for (int i = 0; i < arriving_train_numbers; i++) {
        _Train train(arring_train_model[i], arriving_train_times_hours[i]); //创建动车
        trains.push_back(train); //将动车存入队列中
    }

    /**
     *使用键-值（key-value）映射使每辆动车与所需的维修级别想对应
     *建立以 train_id 为唯一键值，maintenance_level 的行下标为对应值的 map 映射
     */
    int train_maintenance_level_messages[arriving_train_numbers] = { 3,1,1,0,2,1,4 };
    /**
     * 初始化时间参数
     */
    myTm first_arrived_time; //第一辆动车到达时间
    myTm last_arrived_time;//最后一辆动车到达时间
    if (trains.size() > 0) {
        first_arrived_time = trains[0].arrive_time;
        last_arrived_time = trains[trains.size() - 1].arrive_time;
    }
    //当前时间假设为第一辆动车的到达时间
    myTm now_time = first_arrived_time;//当前时间
    //程序开始时间为第一辆动车到达时间
    myTm start_time = last_arrived_time; //程序开始时间
    myTm step_time(0, STEP_TIME);//时间流逝步长为 1 分钟
    //总计时间消耗
    myTm total_time;
    int cur = 0; //当前等待加入维修队列的动车
    while (true) {
        /**
         *判断当前时间大于最后一辆动车的到达时间之
         *如果所有动车维修工厂工作队列都为空
```

```
 *那么维修工作全部完成
 *退出程序
 */
if (!now_time.cmp(last_arrived_time))//当前时间大于最后一辆动车进入的时间时
{
    /**
     * 设置标志 allFactoryEmpty bool 类型
     * true:全部维修完毕
     * false：未全部维修完成
     * 默认为为 true
     */
    bool allFactoryEmpty = true;
    //遍历全部工厂队列，出现不为空的工作工厂队列则表示维修工作未完成
    for (int i = 0; i < numbers; i++) {
        if (!factories[i].empty()) {
            allFactoryEmpty = false;
            break;
        }
    }
    //如果全部车间为空，车辆维修完毕退出程序
    if (allFactoryEmpty) {
        //时间回溯一分钟，去除程序带来的时间偏差
        total_time = total_time - step_time;
        now_time = now_time - step_time;
        break;
    }
}
/** 当前时间到达动车进站时间
 * 该动车进入 a 工序等待队列
 * 等待进站动车序号自增 1
 */
if (cur < trains.size()
    && now_time.hour == trains[cur].arrive_time.hour
    && now_time.minute == trains[cur].arrive_time.minute) {
    //当前动车对应的维修级别
    int level = train_maintenance_level_messages[cur];
    //当前维修级别的维修第一工序
    int process = maintenance_level[level][0];
    //第一维修工序不为空工序，则加入该维修工序等待队列中
    if (process != -1) {
        waitting_trains_id[process - 1].push(cur);
    }
    //动车游标自增
    cur++;
```

```
        }

        //for 循环 numbers 次工序
        for (int i = 0; i < numbers; i++) {
            //当前工序车间的队头元素
            Factory f;
            if (!factories[i].empty()) {
                f = factories[i].top();
            }
            //当前车间非空且车间维修车辆的结束时间小于当前时间，则将当前车辆出队并放入下一维修等待队列
            while (!factories[i].empty() && f.end_work_time.cmp(now_time)) {
                Factory factory = factories[i].top(); //获取当前队首元素
                factories[i].pop(); //出队
                //当前动车对应的维修级别
                int level = train_maintenance_level_messages[factory.train_id];
                //当前维修级别的维修下一工序
                int process = maintenance_level[level][i + 1];
                //下一维修工序不为结束工序（值：-1），则加入下一维修工序等待队列中
                if (process != -1) {
                    waitting_trains_id[process - 1].push(factory.train_id);
                }
                free_factory_ids[i].push(factory.id);//添加进入当前车间队列
                if (!factories[i].empty()) {
                    f = factories[i].top();
                }
            }
            //当前工序空闲车间不为空并且当前工序等待动车不为空
            while (!free_factory_ids[i].empty() && !waitting_trains_id[i].empty()) {
                Factory factory;
                factory.id = free_factory_ids[i].front(); //当前车间 id 来自空闲车间
                free_factory_ids[i].pop();//空闲车间出队一个
                factory.train_id = waitting_trains_id[i].front(); //得到当前等待动车 id
                waitting_trains_id[i].pop();//等待动车出队一辆
                trains[factory.train_id].working_start_time[i] = now_time; //添加当前工序结束时间
                myTm working_time;//当前工序的工作时间
                //当前工序的工作时间来自当前动车类型工作时间散列表中对应的值
                working_time.setTime(working_time_hash[trains[factory.train_id].model][i]);
                trains[factory.train_id].working_end_time[i] = now_time + working_time; //添加当前工序结束时间
                factory.start_work(now_time, working_time); //车间开始工作
```

```
                factories[i].push(factory); //车间进入工作队列
                trains[factory.train_id].factory_id[i] = factory.id; //添加当前工序车间

            }
        }
        total_time = total_time + step_time; //总消耗时间增加一个动车到达间隔时间
        now_time = now_time + step_time; //当前时间增加一个动车到达间隔时间
    }
    cout << "编号 类别 到达时间 A 车间 A 开始 A 结束 B 车间 B 开始 B 结束 C 车间 C 开始 C 结束 D 车间 D 开始 D 结束 E 车间 E 开始 E 结束" << endl;
    for (int i = 0; i < trains.size(); i++) {
        printf(" %02d", i);
        trains[i].print();
    }
    //total_time = now_time - start_time;
    //打印耗时信息
    cout << "维修所有动车耗时共计：" << total_time.hour << "时" << total_time.minute << "分" << endl;
    system("pause"); //控制台暂停
    return 0;
}
```