题目：停车场管理

班级：19184115

姓名：

学号：19063140

日期：2020.11.11

### 需求分析：

设停车场是一个可停放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,在它之后进入的车辆必须先退出车场为它让路,待该辆车开出大门外,其他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。

以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。每一组输入数据包括三个数据项：汽车“到达”或“离去”信息、汽车牌照号码以及到达或离去的时刻。对每一组输入数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场内或便道上的停车位置；若是车辆离去，则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费）。栈依顺序结构实现，队列以链表结构实现以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。每一组输入数据包括三个数据项：汽车“到达”或“离去”信息、汽车牌照号码以及到达或离去的时刻。对每一组输入数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场内或便道上的停车位置；若是车辆离去，则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费）。栈依顺序结构实现，队列以链表结构实现。

。

### 概要设计：

struct Stack 依顺序结构实现的栈

struct Queue 以链表结构实现的队列

struct Car 汽车信息,包含汽车牌照号码以及到达或离去的时刻

void CarArrival(Car car) 汽车到达

void CarDeparture(Car car) 汽车离去

详情见附件代码注释

### 调试分析：

1. 遇到的问题
   1. 问题：自己写结构体好麻烦

解决：果断换C++

1. 时空分析

时间复杂度为 O(n^2)

入栈出栈入队出队 O(1)

从栈中删除，从队种删除一个元素O(n)

总操作次数n

空间复杂度为 O(n)

### 用户使用说明：

设n=2,输入数据为:('A',1,5)，('A',2,10)，('D‘ ,1,15)，('A',3,20)，('A',4，25)，('A',5,30)，('D' ,2,35)，('D' ,4,40)，('E',0,0)。其中:'A'表示到达(Arrival);'D'表示离去(Departure);'E'表示输入结束(End)。

### 测试结果：

输入

2

A 1 5

A 2 10

D 1 15

A 3 20

A 4 25

A 5 30

D 2 35

D 4 40

E 0 0

输出

输入停车场车位：2

A 1 5

1进入停车场

A 2 10

2进入停车场

D 1 15

1离开停车场,停留时间为10

A 3 20

3进入停车场

A 4 25

4进入便道

A 5 30

5进入便道

D 2 35

2离开停车场,停留时间为25

4由便道进入停车场

D 4 40

4离开停车场,停留时间为5

5由便道进入停车场

E 0 0

### 附录：

1. #include <bits/stdc++.h>
3. **using** **namespace** std;
5. **template** <**typename** T>
6. **struct** Stack {
7. T \*elem; // 顺序存储
8. **int** cnt, cap; // cnt 当前数量, cap 停车场大小
9. // 初始话,传入停车场大小
10. **void** init(**int** n) {
11. cap = n;
12. elem = **new** T[n];
13. cnt = 0;
14. }
15. **bool** empty() {
16. **return** !cnt;
17. }
18. **int** size() {
19. **return** cnt;
20. }
21. **int** capacity() {
22. **return** cap;
23. }
24. // 插入栈顶元素
25. **bool** push(T x) {
26. **if** (cnt >= cap) **return** **false**;
27. **return** elem[cnt++] = x, **true**;
28. }
29. // 删除栈顶元素
30. T pop() {
31. **if** (!cnt) **return** T();
32. **return** elem[--cnt];
33. }
34. // 获取栈顶元素
35. T top() {
36. **if** (!cnt) **return** T();
37. **return** elem[cnt-1];
38. }
39. // 删除任一元素
40. **int** erase(T x) {
41. Stack<T> tmp;
42. **int** time = -1;
43. tmp.init(cap);
44. **while** (cnt && top() != x) {
45. tmp.push(pop());
46. }
47. **if** (cnt) time = pop().time;
48. **while** (tmp.size()) push(tmp.pop());
49. **return** time;
50. }
51. };
53. **template** <**typename** T>
54. **struct** Queue {
55. // 链表结构存储
56. **struct** Node {
57. T elem;
58. Node \*pre, \*nex;
59. };
60. **int** cnt; // 队列元素数量
61. Node \*head, \*tail;
62. // 初始化
63. **void** init() {
64. cnt = 0;
65. head = tail = **new** Node();
66. head->nex = tail;
67. tail->pre = head;
68. }
69. **bool** empty() {
70. **return** !cnt;
71. }
72. **int** size() {
73. **return** cnt;
74. }
75. // 在队尾插入元素
76. **void** push\_back(T x) {
77. ++cnt;
78. tail->nex = **new** Node();
79. tail->nex->pre = tail;
80. tail->elem = x;
81. tail = tail->nex;
82. }
83. // 在队首插入元素
84. **void** push\_front(T x) {
85. ++cnt;
86. head->pre = **new** Node();
87. head->pre->nex = head;
88. head = head->pre;
89. head->elem = x;
90. }
91. // 在队首删除元素
92. T pop\_front() {
93. --cnt;
94. T res = head->elem;
95. head = head->nex;
96. **delete** head->pre;
97. head->pre = nullptr;
98. **return** res;
99. }
100. // 在队尾删除元素
101. T pop\_back() {
102. --cnt;
103. T res = tail->elem;
104. tail = tail->pre;
105. **delete** tail->nex;
106. tail->nex = nullptr;
107. **return** res;
108. }
109. // 获取队首元素
110. T front() {
111. **return** head->elem;
112. }
113. // 获取队尾元素
114. T back() {
115. **return** tail->pre->elem;
116. }
117. // 删除任一元素
118. **bool** erase(T x) {
119. Queue<T> tmp;
120. **int** flag = 0;
121. tmp.init();
122. **while** (cnt && back() != x) {
123. tmp.push\_back(pop\_back());
124. }
125. **if** (cnt) flag = 1, pop\_back();
126. **while** (tmp.size()) {
127. push\_back(tmp.pop\_back());
128. }
129. **return** flag;
130. }
131. };
133. **struct** Car {
134. string id; // 车牌信息
135. **int** time; // 时间信息
136. **bool** operator ==(Car &c) **const** {
137. **return** c.id == id;
138. }
139. **bool** operator !=(Car &c) **const** {
140. **return** c.id != id;
141. }
142. };
144. Stack<Car> stk;
145. Queue<Car> que;
147. // 汽车到达，判断进入停车场还是便道
148. **void** CarArrival(Car car) {
149. **if** (stk.size() < stk.cap) {
150. stk.push(car);
151. cout << car.id << "进入停车场" << endl;
152. } **else** {
153. que.push\_back(car);
154. cout << car.id << "进入便道" << endl;
155. }
156. }
158. // 汽车离去
159. **void** CarDeparture(Car car) {
160. **int** last = stk.erase(car);
161. // 返回-1表示汽车在便道
162. **if** (last == -1) {
163. **if** (que.erase(car)) {
164. cout << car.id << "离开便道" << endl;
165. } **else** {
166. cout << car.id << "不存在" << endl;
167. }
168. } **else** {
169. cout << car.id << "离开停车场,停留时间为" << car.time-last << endl;
170. // 停车场有车离去，便道的车进入停车场
171. **if** (que.size()) {
172. cout << que.front().id << "由便道进入停车场" << endl;
173. stk.push(Car{que.pop\_front().id, car.time});
174. }
175. }
176. }
178. **int** main() {
179. **int** n, t;
180. string id;
181. **char** op;
183. cout << "输入停车场车位：";
184. cin >> n;
185. stk.init(n);
186. que.init();
187. **while** (**true**) {
188. cin >> op >> id >> t;
189. **if** (op == 'A') {
190. CarArrival(Car{id, t});
191. } **else** **if** (op == 'D') {
192. CarDeparture(Car{id, t});
193. } **else** {
194. **break**;
195. }
196. }
197. **return** 0;
198. }