EE 441 HOMEWORK #2

Gamers Database

Due: December 8, 2016, 23:59
For questions: ccakmak@metu.edu.tr
eylen@metu.edu.tr

Introduction

Well done creating the Gamer Database! You have successfully created a list of users to play games in our system. Now it is time to help them join teams for competitive multiplayer games.

One of the games in our database will let the players play an N-player game against another N-player team. The player arrives to a virtual room where there are two virtual teams each awaiting for players to join their side to start a game. For a better gaming experience, a matchmaking algorithm is to be implemented so that both sides are of similar strength. According to this algorithm, a player is to be accepted to one of the teams if the player's score is within an acceptable range, otherwise the player enters a waiting queue to be considered for admission to a team later in due course. The players waiting try their chance again when the team score averages change. The game will starts immediately when both teams have acquired N players on their side.

For this task, you can use the classes you have created in HW1. However, since we deal with a single game here, it would be simpler if you define and use a new *User* class and forget about *Game* class. Each user will have User ID and Score as attributes (plus any additional attributes you may want to define).

In HW1, you had created static *User* objects. This time, you will read the score and ID of each User from a text file and create *User* objects dynamically.

Matchmaking Mechanics

You are expected to implement the matchmaking algorithm using two containers representing the two teams and a queue structure representing the waiting queue. The containers will store *User* objects. Your program should allow N-player team vs N-player team games, but N is fixed as 4 in the following examples.

(Hint: Stack and queue are also containers. Your container should have some functions such as averaging stored *User* scores.)

Players are placed into teams in turns. In other words, your program will check a player to be admitted only one of the teams at a time. If placement into one team fails, the player will enter the queue, rather than checking directly whether he/she can be admitted to the other team.

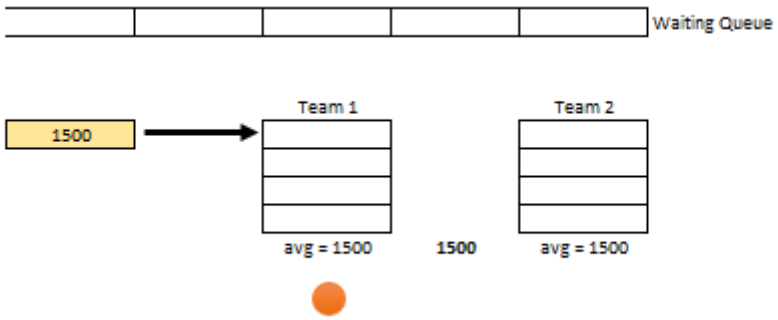
The average score of a team is the average of all the players' score in the team. The overall average is the average of the two team averages. This value is marked as **BOLD** in the flowing examples.

The overall team formation algorithm is described in the form of a pseudocode in the following:

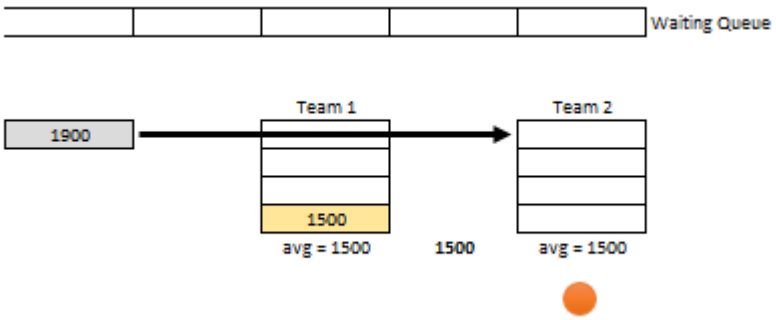
```
initialize Team1_avg, Team2_avg = 1500
repeat the following lines for each team in turns until no more player is available
   read a new player from txt file as an arriving player
   while (team in turn is not full with players)
      if (Score(Team_avg)>Score(Bold_avg) and (Score(player )< Score(Bold_avg))
           OR (Score(Team_avg)<Score(Bold_avg) and (Score(player)>Score(Bold_avg))
                OR (Score(Team_avg)=Score(Bold_avg))
                     add player to team and update averages
      if the arriving player is added to team then
           repeat until no more admission from the queue is possible
                consider adding a player from the queue to other team such that the player
                   that satisfies the above logical test and that has the longest waiting time is
                   admitted
                update averages if admitted
                change team order
          else insert the player into waiting queue
      change team in turn
```

Example of Matchmaking

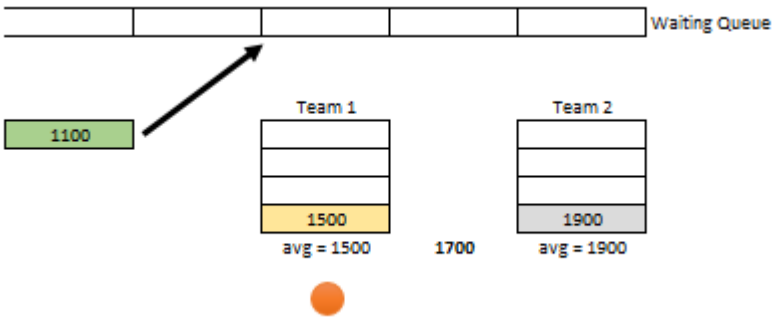
Some possible cases in our player admission algorithm are illustrated below:



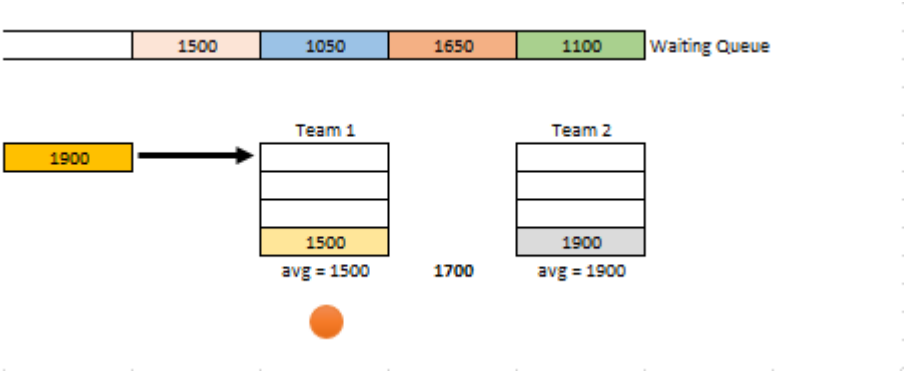
The above is the initial state. The red dot shows which team has the turn to get a player, which is Team 1 in this case. A player with 1500 points arrives at the room (read from the text file in your case). We compare team average to bold average. The average score of Team 1 (1500) is **equal** to the bold average (1500). If this is the case, **any player can join the team**, so the brown colored player joins Team 1. (Note that empty container is assumed to have 1500 points as average score.)



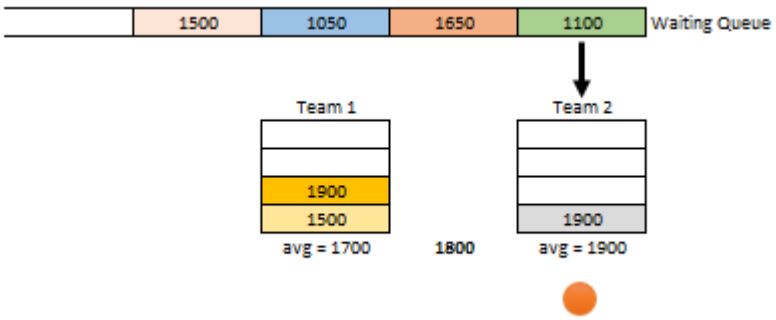
Then a player with 1900 points comes and it is Team 2's turn. We compare Team 2 average (1500) with bold average (1500). Similar to above case the average score of Team 2 (1500) is equal to the bold average (1500), so the grey colored player joins Team 2.



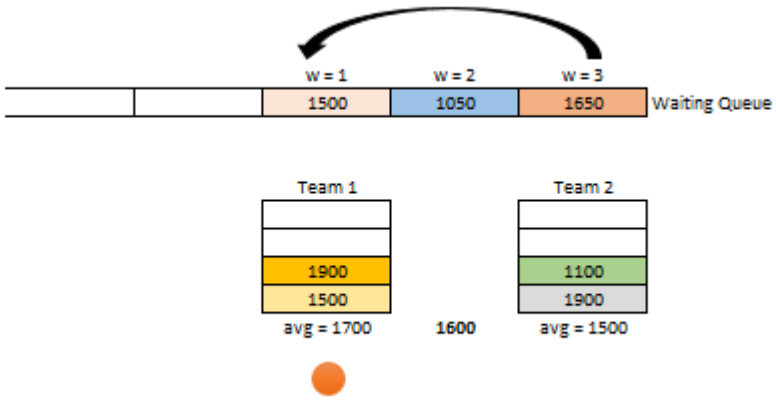
Now it's Team 1's turn again, and a player with 1100 points comes. Team 1 (1500) < Bold (1700), i.e. **Team 1 average is lower**. In this case, we don't want to decrease Team 1 average further, hence Green player is not accepted and it goes to a waiting queue.



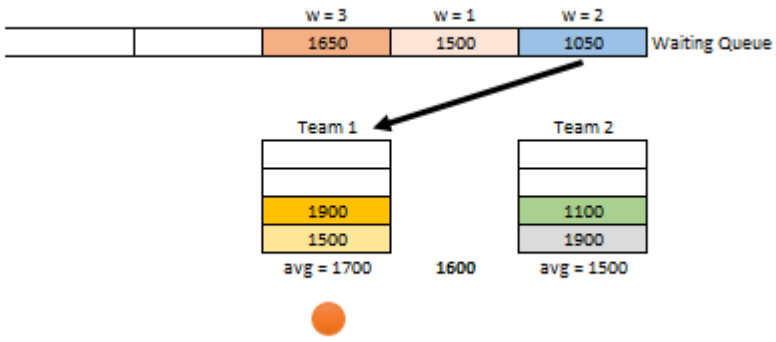
Now assume that red, blue and pink players have arrived after green player in due course and have not been admitted to the teams and are inserted into the queue. Assume that it is still Team 1's turn Yellow player with 1900 points arrives now and is admitted to Team 1 since it fits the rule.



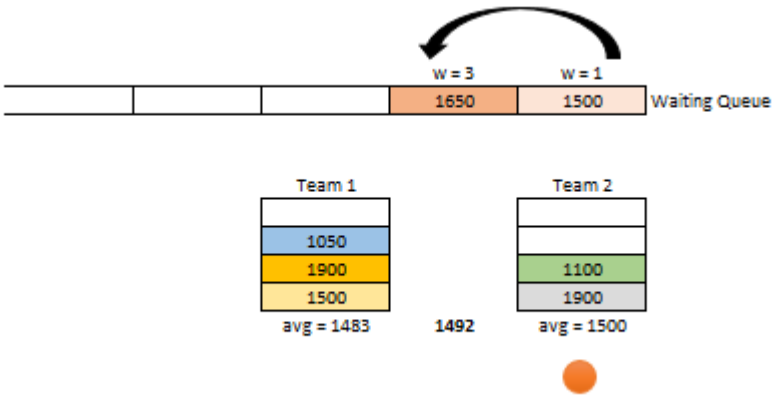
The yellow player joins Team 1. The new average score of Team 1 is 1700 and the new Bold average is 1800. It is Team 2's turn. Just after a new player is placed into a team, i.e. the Bold average is updated, the matchmaking algorithm must check the waiting queue and try to place all the players waiting there. Please keep in mind that the same rules for placement apply here. **Team 2 is higher**, so we want to increase the team average. In this case, Team 2 accepts players only with **Score < Bold average**. Green player joins Team 2.



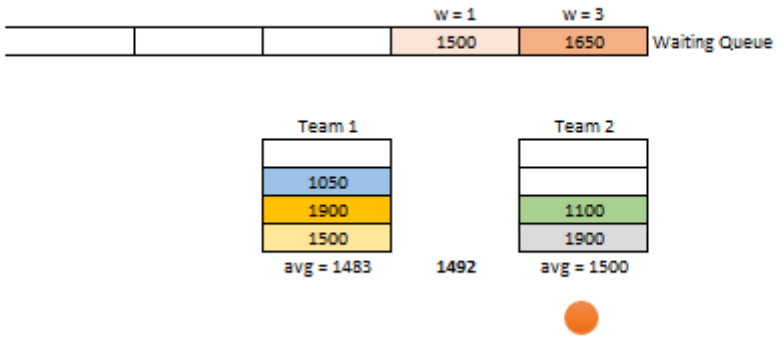
Red player cannot enter Team 1, so it goes back to the end of the line.



Blue player enters Team 1.



Pink player cannot enter Team 2. Goes back to the end of the line.



Red player is checked again. It cannot enter Team 2. After a certain number of cycles, we are sure that no one in the queue can enter now. This is the final state of the queue before taking a new player into the room. Please note that the red player, who has waited longer, is before pink player in the line, i.e. the order of the queue must not be changed in the end. You will have to keep this in mind while writing your code.

This algorithm stops when both sides have 4 players. The players in the teams will be popped out then.

Input and Output

You will create a list of users in a text file. Your program will read these users from the text file one by one, i.e. take them into the room one by one, and try to place them into a team according to the algorithm described above. When the process is completed, it will output who are playing for each team and who are in the queue.

To be able to read from and write to a text file, you should include <fstream> library. Then you can use 'ifstream' and 'ofstream' classes which are called as stream classes. An example for reading from a txt file given below.

```
ifstream inputtxt;
inputtxt.open("in.txt");
char output[100];
if (inputtxt.is_open()) {
        while (!inputtxt.eof()) {
                inputtxt >> output;
                cout << output;
        };
};
inputtxt.close();
```

'in.txt' is your input txt file. It should include number of players for a team as the first line. Remaining lines should include score and id of the *User* object which are separated by comma. Above code segment reads and outputs lines one by one for every iteration in while loop.

An example for writing to a txt file given below.

```
ofstream outputtxt;
outputtxt.open("out.txt");
outputtxt << "example";
outputtxt.close();
```

The above code segment writes "example" to the out.txt file.

Examples of input and output txt files are given below.

Output txt file:

```
Team 1
1350,109
1050,105
1900,107
1500,101

Team 2
1650,104
1200,108
1100,103
1900,102

Waiting Queue
1500,106
```

Input txt file:

```
4
1500,101
1900,102
1100,103
1650,104
1050,105
1500,106
1900,107
1200,108
1350,109
```

Regulations:

1. You should insert comments to your source code at appropriate places without including any unnecessary detail. Comments will be graded. You have to write to-the-point comments in your code, otherwise it would be very difficult to understand. If your output is wrong, the only way we can grade your homework is through your comments.

2. Use **Code::Blocks IDE** and choose GNU GCC Compiler while creating your project. Name your project as "e<student_ID>_HW2". Send the whole project folder compressed in a rar or zip file. You will not get full credit if you fail to submit your project folder as required.

3. Your C++ program should follow object oriented principles, including proper class and method usage and should be correctly structured including private and public components. Your work will be graded on its correctness, efficiency and clarity as a whole.

4. Late submissions are welcome, but penalized according to the following policy:

    - 1 day late submission: HW will be evaluated out of 70.
    - 2 days late submission: HW will be evaluated out of 50.
    - 3 days late submission: HW will be evaluated out of 30.
    - 4 or more days late submission: HW will not be evaluated.

Good Luck!