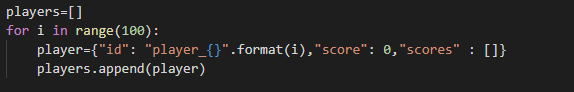
## **Step 1: To organize the tournament**

### Propose a data structure to represent a Player and its Score

We choose to use a list of dictionaries to represent a Player and its Score. Each of the 100 dictionaries will contain three keys: id, score and scores which represent the id of a player, actual score and the scores of his games. List are easy to create, modify and downscale and upscale. Dictionary are easy to use and to organize data. The combination of the them makes it easy to organize and downscale.



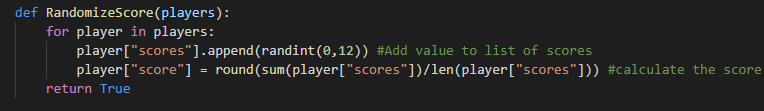
### Propose a most optimized data structures for the tournament (called database in the following questions)

The most optimized data structure for the tournament is the AVL-tree. By using a self-balancing tree structure, it is possible to implement a list that can access and modify single elements in just Θ(log n) worst-case time, where n is the current size of the list.



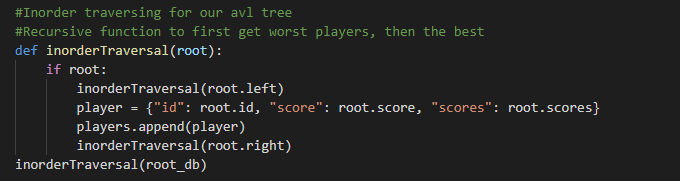
### Present and argue about a method that randomize player score at each game (between 0 point to 12 points)

We will simply use the **randint()** function form random library in order to randomize player score at each game. We set the min and max value at 0 and 12 to follow the rules. We will return a list of value that another method will be able to use.

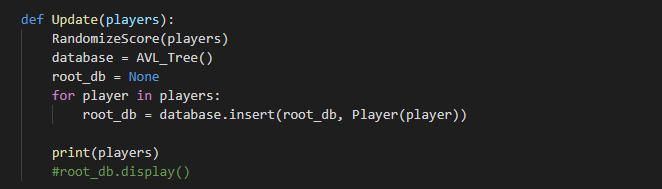


### Present and argue about a method to update the database

Our **inorderTraversal()** method, that performed in 𝑂(𝑛), is a recursive function that allows us to get each in order each elements by traversing recursively the left subtree of the root node accessing the node itself and then recursively crossing the right subtree of the node. It will get the worst players first then the best ones.



First, we apply our **inorderTraversal()** to our database. Each player node will get copy in a new players array. Using our **Update()** function we will then generate random scores for each player, compute score of the new game generated and create a new sorted AVL tree. Our part 4 is done in 𝑂(n \* 𝑙𝑜𝑔 𝑛).

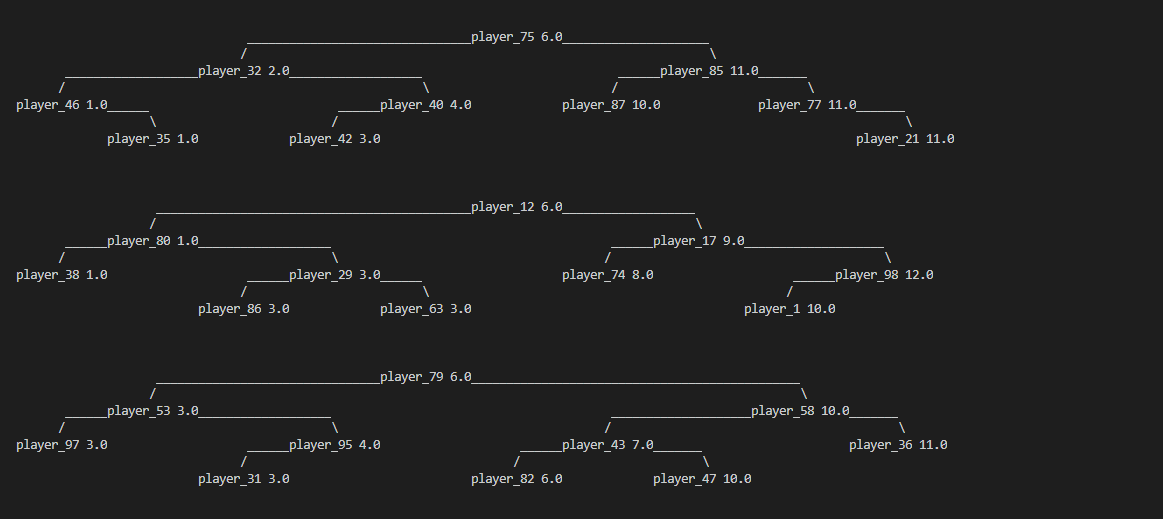
****

### Present and argue about a method to create random games based on the database

In our method we started to create 10 empty AVL tree a root node associated. Then we just randomly, using **sample()** method of random library, sort a list of value from 0 to 99 to represent the 100 players that play during the random games based on the database. We then fill the 10 AVL tree with player corresponding to the randomly sorted list.

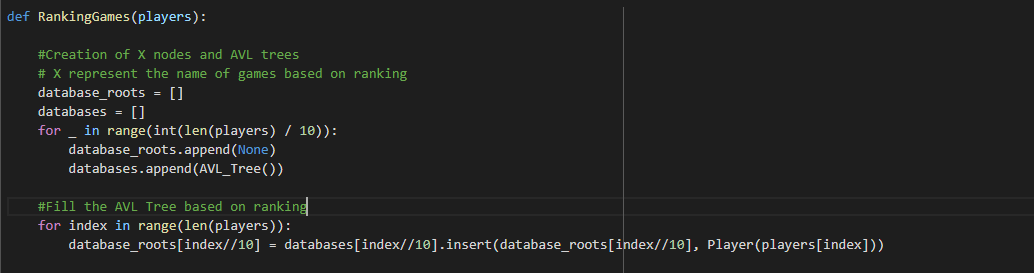


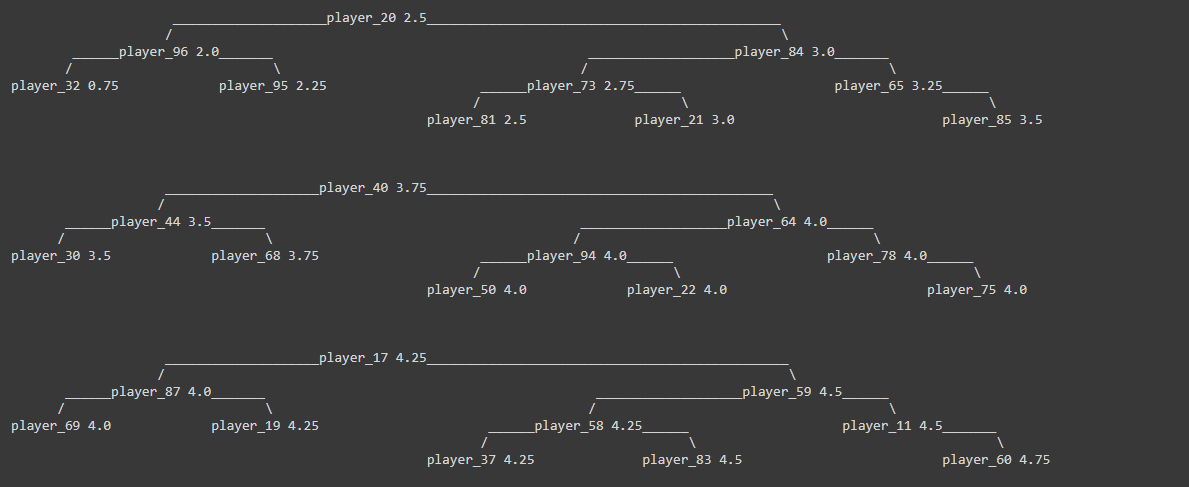
Some samples of the AVL tree after generated some random values and used a display method not presented yet:



### Present and argue about a method to create games based on ranking

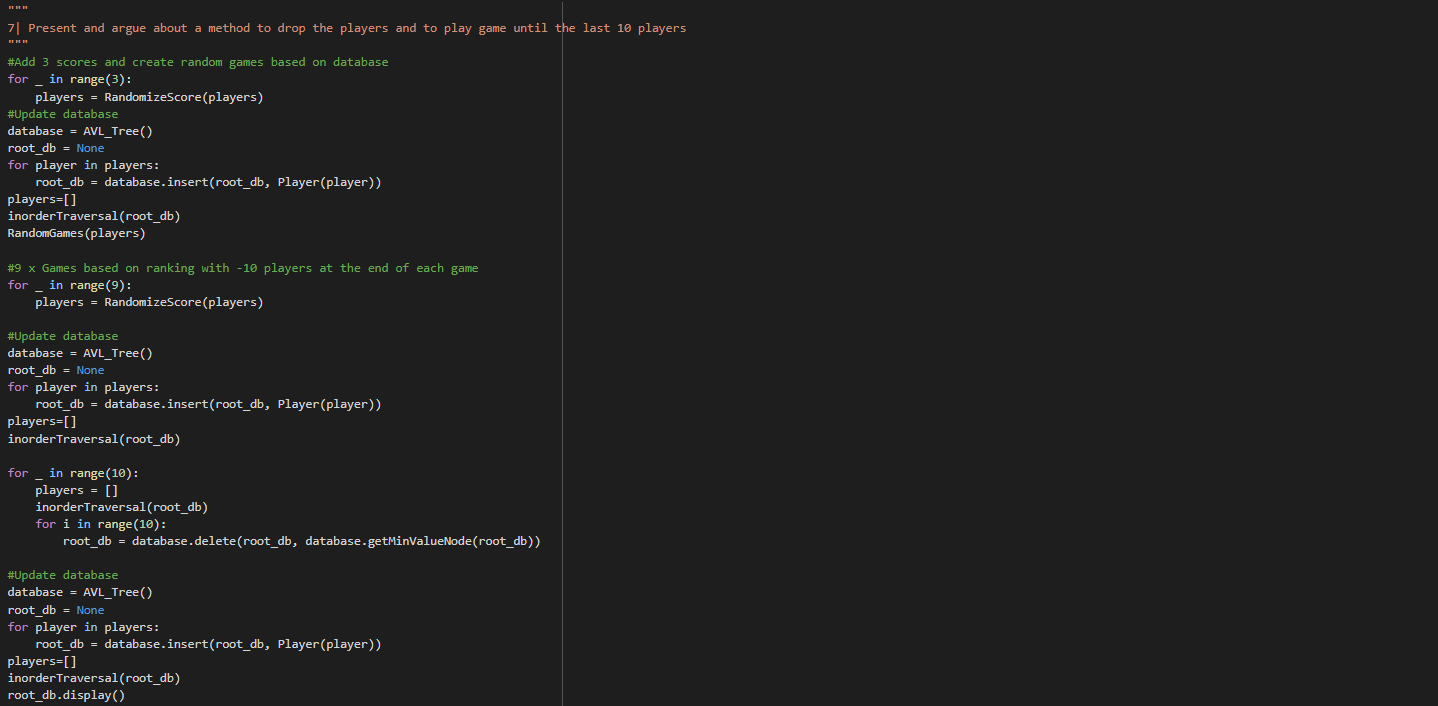
In order to create games based on ranking we first get the ranking of the players based on their average score, then we filled our database in the order of the ranking. Then our function below will create the right numbers of games, based of the number of players left in the tournament, and will fill them in order (from the worst player to the best player).

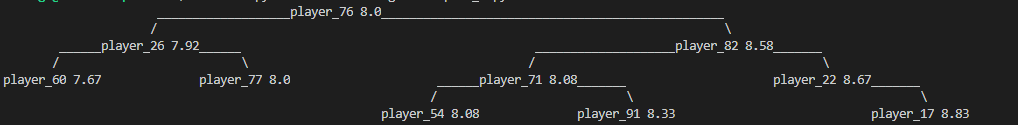




### Present and argue about a method to drop the players and to play game until the last 10 players

We simulate the tournaments rules by first adding three random scores, update the database, create games based on ranking, drop the 10 last players at the end of each match and update the database after each match to get the 10 best players.

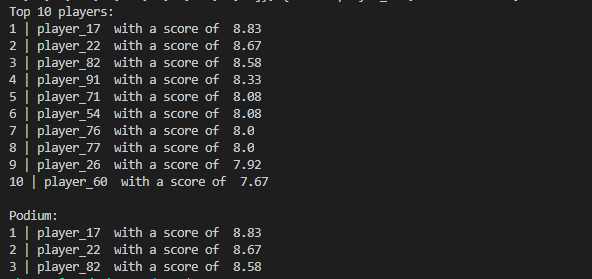




### Present and argue about a method which display the TOP10 players and the podium after the final game.

In this part we just have to print in the right order the player listed in the database. Because the database was updated, the players are listed from the worst to the best. So we juste have to display players from the end of the list to the beginning.

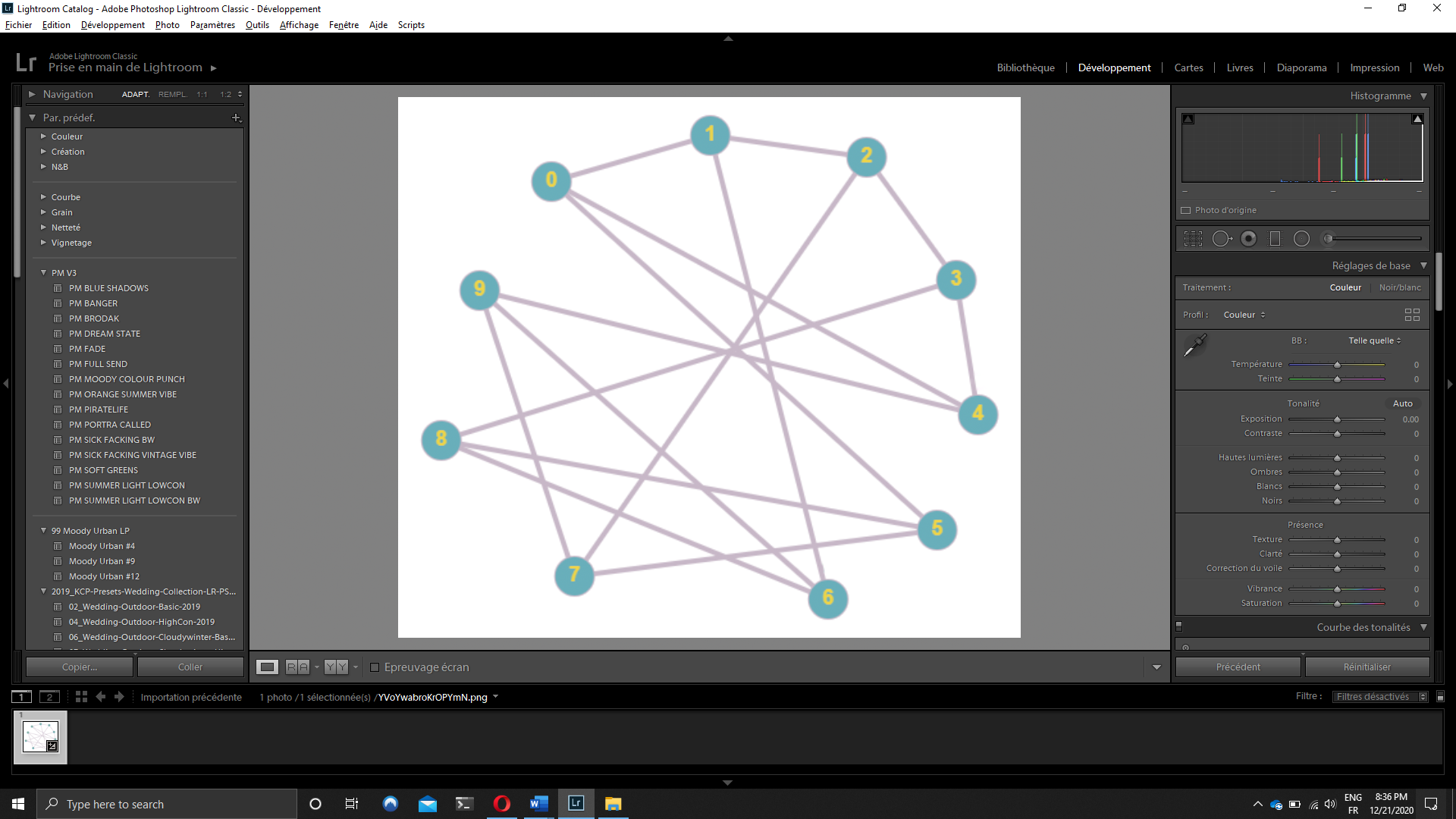
## 



## **Step 2: Professor Layton < Guybrush Threepwood < You**

### Represent the relation (have seen) between players as a graph, argue about your model.

We represent the relation between players by using an undirected graph as shown below:



Players are represented by nodes (or vertices) and if a player has seen another player, we draw a link between them. We chose the undirect graph to represent the problem because what is interesting is to know which player have walk with who. So, if a player A has seen a player B but the player B hasn’t seen player A, we consider that the players walked together.

### Thanks to a graph theory problem, present how to find a set of probable impostors.

We consider that player 0 was killed by player 1, 4 or 5, that Impostors never walk together, and the second impostor hasn’t seen player 1, 4 or 5.

To find a set of probable impostors we will go through three operations. First regroup players that didn’t walk together by using graph coloring. Then we will eliminate the group of players in which player 0 is because the other players of this group haven’t seen the player 0 so they couldn’t kill him. And finally, we will check that in the remaining selected groups, there is a group in which a player hasn’t seen player 1, 4 or 5. That’s how we will find a set of probable impostors.

### Argue about an algorithm solving your problem.

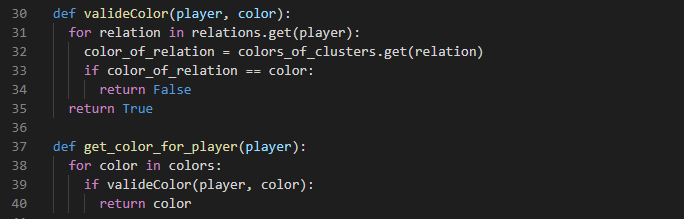
We will implement Greedy algorithm in order to solve our graph coloring problem. It is simple, easy to implement and run fast. It will build up the solution piece by piece.

We only have to set all the data of the problem and set a rule for which elements to add to the solution at each step of the algorithm.

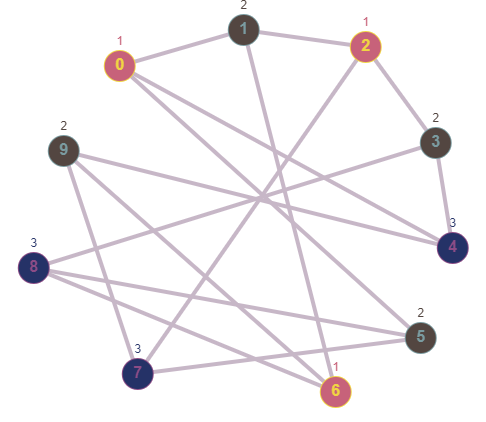
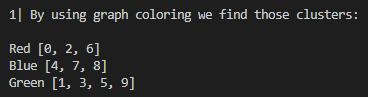
### Implement the algorithm and show a solution.

In our code we called **colors** the list representing the set of colors which a player can have, **players** the list representing each player and **relations** the linked between players (if a player have seen a player).

This function will find the associate color to a player by checking the color of players’ relation. If a color is not attribute to one of his relation, the player will take this color.

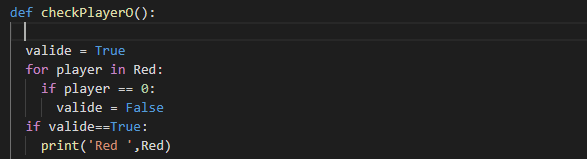


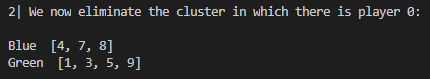
By repeating this function to every player, we will establish group of players that we will call clusters:



After using our greedy algorithm to find clusters, we will have two other steps to find the set of Impostors.

We will check for every cluster if the player 0 is list inside. Is the player 0 is found, we don’t print the cluster in which he is.





And finally, we will check that in the remaining selected groups, there is a group in which a player hasn’t seen player 1, 4 or 5



Our set of Impostors is 1, 3, 5 and 9.