Q_1

PF:

As R, S_1, S_2, T aren't modified during the rotation, we only need to convince that v, x, s are balanced after double rotation for the proof of the success of the rebalanced.

Let $r = \text{size}(R)$, $s_1 = \text{size}(S_1)$, $s_2 = \text{size}(S_2)$, $t = \text{size}(T)$

v is right-heavy, so either:

1. A node was added to x to cause imbalance

2. A node was removed from R to cause imbalance

Case I : a node was added to x to cause an imbalance

Assumptions:

$s_1 + s_2 + t + 3 > 3(r+1)$ ① v is right-heavy

$s_1 + s_2 + 2 \leq 3(t+1)$ ② x is balanced

$\wedge t + 1 \leq 3\,(s_2 + s_1 + 2)$③

$s_1 + 1 \leq 3\,(s_2 + 1)$ ④ s is balanced

$\wedge s_2 + 1 \leq 3\,(s_1 + 1)$⑤

$s_1 + s_2 + 2 \geqslant 2(t+1)$ ⑥ assumption

Before addition:

$3(r+1) \geq s_1 + s_2 + t + 2$ ⑦ v was balanced

$\wedge r + 1 \leq 3\,(s_1 + s_2 + t + 2)$ ⑧

$$\left.\begin{array}{l}(t \leq 3\,(s_1 + s_2 + 2) \wedge (s_1 + s_2 + 2) \leq 3t) \vee \\ (t + 1 \leq 3\,(s_1 + s_2 + 1) \wedge (s_1 + s_2 + 1) \leq 3(t+1))\end{array}\right\} \Rightarrow \begin{array}{l} t \leq 3\,(s_1 + s_2 + 2)\,⑨ \\ \wedge\,(s_1 + s_2 + 1) \leq 3(t+1)⑩\end{array}$$

x was balanced

$$\left.\begin{array}{l}(s_1 \leq 3\,(s_2 + 1) \wedge s_2 + 1 \leq 3s_1) \vee \\ (s_2 \leq 3\,(s_1 + 1) \wedge (s_1 + 1) \leq 3s_2)\end{array}\right\} \Rightarrow \begin{array}{l} s_1 \leq 3\,(s_2 + 1)\,⑪ \\ \wedge s_2 \leq 3\,(s_1 + 1)\,⑫\end{array}$$

s was balanced

WTP:

$r + s_1 + 2 \leqslant 3\,(t + s_2 + 2)$⑬ s is balanced

$\wedge t + s_2 + 2 \leq 3\,(r + s_1 + 2)$⑭

$r + 1 \leqslant 3\,(s_1 + 1)$ ⑮ v is balanced

$\wedge(s_1 + 1) \leqslant 3(r + 1)$⑯

$s_2 + 1 \leq 3(t + 1)$⑰ x is balanced

$\wedge(t + 1) \leq 3\,(s_2 + 1)$⑱

By ①②③④⑤⑥⑦⑧⑨⑩⑪⑫

We have:

$s_1 + s_2 + t + 3 > 3(r + 1)$ ⑲

$s_1 + s_2 + 2 \leq 3(t + 1)$ ⑳

$t + 1 \leq 3\,(s_1 + s_2 + 2)$㉑

$s_1 + 1 \leq 3\,(s_2 + 1)$ ㉒

$s_2 + 1 \leq 3\,(s_1 + 1)$㉓

$s_1 + s_2 + 2 \geq 2(t + 1)$㉔

$3(r + 1) \geq s_1 + s_2 + t + 2$㉕

$r + 1 \leq 3\,(s_1 + s_2 + t + 2)$㉖

By ⑲ ⑳:

$3(r + 1) < s_1 + s_2 + t + 3 = s_1 + s_2 + 2 + t + 1 \leq 4(t + 1)$

$\Rightarrow r + 1 < \frac{4}{3}(t + 1) < 3(t + 1)$㉗

Consider ⑬:

$r + s_1 + 2 = r + 1 + s_1 + 1 < 3(t + 1) + 3\,(s_2 + 1)$ by ㉒, ㉗

$= 3\,(t + s_2 + 2)$, as wanted

By ㉕

$3(r + 1) \geqslant s_1 + s_2 + t + 2 > t + 1$ ㉘

Consider ⑭:

$t + s_2 + 2 = t + 1 + s_2 + 1 < 3(r+1) + 3(s_1+1)$ by ㉓ ㉘

$= 3(r + s_1 + 2)$, as wanted.

Consider ⑮:

$$\begin{aligned} r+1 &< \frac{4}{3}(t+1) \text{by } ㉗ \\ &= \frac{2}{3} \cdot 2(t+1) \\ &\leq \frac{2}{3}(s_1 + s_2 + 2) \text{ by } ㉔ \\ &= \frac{2}{3}(s_1+1) + \frac{2}{3}(s_2+1) \\ &\leq \frac{2}{3}(s_1+1) + 2(s_1+1) \text{ by } ㉓ \\ &< 3(s_1+1) \end{aligned}$$

$\Rightarrow r + 1 \leq 3(s_1 + 1)$, as wanted

Consider ⑯:

$s_1 + 1 < s_1 + s_2 + t + 2 \leq 3(r+1)$ by ㉕

$\Rightarrow s_1 + 1 \leqslant 3(r+1)$, as wanted

Consider ⑰:

$s_2 + 1 < s_1 + s_2 + 2 \leq 3(t+1)$ by ⑳

$\Rightarrow s_2 + 1 \leqslant 3(t+1)$ as wanted

Consider ⑱:

$$
\begin{aligned}
t+1 &\leq \frac{1}{2}(s_1+s_2+2) \quad \text{by } \textcircled{24} \\
&= \frac{1}{2}(s_1+1+s_2+1) \\
&\leq \frac{1}{2}(4(s_2+1)) \text{ by } \textcircled{22} \\
&= 2(s_2+1) \\
&< 3(s_2+1) \\
&\Rightarrow t+1 \leq 3(s_2+1) \text{ as wanted}
\end{aligned}
$$

Therefore, $v.x.s$ are balanced after double rotation

Case II: a node was removed from R to cause imbalance

Assumptions:

$s_1+s_2+t+3 > 3(r+1)$ ㉙ v is right-heavy

$s_1+s_2+2 \geqslant 2(t+1)$ ㉚ assumption

Before removal:

$3(r+2) \geq s_1+s_2+t+3$ ㉛ v was balanced

$\wedge r+2 \leq 3(s_1+s_2+t+3)$ ㉜

$s_1+s_2+2 \leq 3(t+1)$ ㉝ x was balanced

$\wedge t+1 \leq 3(s_2+s_1+2)$ ㉞

$s_1+1 \leq 3(s_2+1)$ ㉟ s was balanced

$\wedge s_2+1 \leq 3(s_1+1)$ ㊱

WTP:

$r+s_1+2 \leq 3(t+s_2+2)$ ㊲ s is balanced

$\wedge t+s_2+2 \leq 3(r+s_1+2)$ ㊳

$r+1 \leq 3(s_1+1)$ ㊴ v is balanced

$\wedge (s_1+1) \leq 3(r+1)$ ㊵

$s_2+1 \leq 3(t+1)$ ㊶ x is balanced

$\wedge(t+1) \leq 3\,(s_2+1)$ ㊷

By ㉙ ㉝

$3(r+1) < s_1+s_2+t+3 = s_1+s_2+2+t+1 \leqslant 4(t+1)$

$\Rightarrow r+1 < \frac{4}{3}(t+1) < 3(t+1)$ ㊸

Consider ㊲:

$r+s_1+2 = r+1+s_1+1 \leqslant 3(t+1)+3\,(s_2+1) = 3\,(t+s_2+2)$ by ㉟ ㊸

Consider ㊳:

$t+s_2+2 \leq s_1+s_2+t+2 \leq 3(r+2) \leq 3\,(r+s_1+2)$ by ㉛

Consider ㊴:

$$\begin{aligned} r+1 &< \frac{4}{3}(t+1) \text{by } ㊸ \\ &= \frac{2}{3}\cdot 2(t+1) \\ &\leq \frac{2}{3}\,(s_1+s_2+2) \text{ by } ㉚ \\ &= \frac{2}{3}\,(s_1+1)+\frac{2}{3}\,(s_2+1) \\ &\leq \frac{2}{3}\,(s_1+1)+2\,(s_1+1) \text{ by } ㊱ \\ &< 3\,(s_1+1) \end{aligned}$$

$\Rightarrow r+1 \leq 3(s_1+1)$, as wanted

Consider ㊵:

$3(r+1) \geqslant s_1+s_2+t$ ㊹ by ㉛

if $s_2+t=0$

As $s_2 \geqslant 0, t \geqslant 0$,

$s_2 = t = 0$

$s_1+2 \leq 3\times 1$ by ㉝

$s_1 \leq 1$

$\therefore s_1 = 1$ or 0

$s_1 + 0 + 0 + 3 > 3(r+1) \geqslant 3$ by ㉙

$s_1 > 0$

$\Rightarrow s_1 = 1$

$4 > 3(r+1)$ by ㉙

$3r < 1$

As $r \geqslant 0$,

$r = 0$

$3(r+1) = 3 \geqslant 1 = s_1 + 1$

else:

As $s_2 \geqslant 0, t \geqslant 0$ and $s_2 + t \neq 0$,

$\Rightarrow s_2 + t \geqslant 1$

$3(r+1) \geqslant s_1 + s_2 + t$ by ㊹

$\geqslant s_1 + 1$

$\Rightarrow 3(r+1) \geqslant s_1 + 1$

By two cases:

$3(r+1) \geqslant s_1 + 1$

Consider ㊶

$s_2 + 1 < s_1 + s_2 + 2 \leq 3(t+1)$ by ㉝

$\Rightarrow s_2 + 1 \leq 3(t+1)$ as wanted

Consider ㊷

$$
\begin{aligned}
t+1 &\leq \frac{1}{2}\left(s_1+s_2+2\right) \text{ by}\textcircled{30} \\
&= \frac{1}{2}\left(s_1+1+s_2+1\right) \\
&\leq \frac{1}{2}\left(4\left(s_2+1\right)\right) \text{ by}\textcircled{35} \\
&= 2\left(s_2+1\right) \\
&< 3\left(s_2+1\right) \\
&\Rightarrow t+1 \leq 3\left(s_2+1\right) \text{ as wanted}
\end{aligned}
$$

According to the two cases mentioned above, $v.s.x$ are always balanced after double-rotation

$\Rightarrow$ In this case, the double counter-clockwise rotation restores the balance as wanted.

QED.

Q_2

1.

```
split(T,k) -- pseudocode
  if T == nil:
      return (nil, nil, false)
  if k == T.key:
      return (T.left, T.right, true)
  if k < T.key:
      (L, R, b) = split(T.left, k)
      T' = new node(key = T.key, size = 1,
                    left = nil, right = nil)
      R_1 = join(R, T')
      R_2 = join(R_1, T.right)
      return (L, R_2, b)
  if k > T.key:
      (L, R, b) = split(T.right, k)
      T' = new node(key = T.key, size =1,
                    left = nil, right = nil)
      L_1 = join(T.left, T')
      L_2 = join(L_1, L)
      return (L_2, R, b)
```

2.

```
remove_max(T) -- pseudocode
  if T == nil:
```

```
      return (nil, none)
  if T.right == nil:
      return (T.left, T.key)
  (T.right, max) = remove_max(T.right)
  T.size -= 1
  T' = rebalance(T)
  return (T', max)

join(L,G) -- pseudocode
  if L == nil:
      return G
  if G == nil:
      return L
  (L', max) = remove_max(L)
  T = new node(key = max, size = L.size + G.size,
               left = L', right = G)
  return rebalance(T)
```

3.

```
difference(T_1, T_2) -- pseudocode
  if T_1 == nil:
      return nil
  if T_2 == nil:
      return T1
  k = T_2.key
  (L, R, b) = split(T_1, k)
```

```
L' = difference(L, T_2.left)
R' = differnece(R, T_2.right)
return join(L', R')
```

Q_4

1.

Node	is adjacent to
1	2, 4, 3
2	5, 6
3	5, 7
4	6, 7
5	
6	
7	

Tabula 1: Adjacency List for left tree

Node	is adjacent to
1	4, 3, 2
2	5, 6
3	5, 7
4	6, 7
5	
6	
7	

Tabula 2: Adjacency List for right tree

2.

Suppose the given tree can be a breadth-first tree of G.

In this breadth-first tree, 3, 5 is a breadth-first tree edge. However, as 2 and

3 both can be the predecessor of 5, we need to dequeue 3 before 2 if we want to have 3, 5. Now, 4 can either be before 3 or after 2 in 1's adjacency list. As 2 and 4 both can be the predecessor of 6 and we now have the breadth-first tree edge 2, 6, 2 should be before 4. So, 4 can only be the last one in 1's adjacency list. Therefore, 3 is before 4 in 1's adjacency list and 3, 7 will be a breadth-first tree edge (the only two numbers that 7 is adjacent to are 3 and 4), which lets 4 be unable to be the predecessor of 7. But in the given tree, 4, 7 is a breadth-first tree edge while 3, 7 is not. There is a contradiction. So the supposition is false and the given tree cannot be a breadth-first tree of G.

Q_5

The given problem can be converted into a problem that is coloring an undirected graph. Each wrestler can be considered as a vertex, and if two wrestlers are a pair of rivalry, then the two vertices that related to them have an edge between them. We need to color the two vertices of an edge with two different colors. So we need to generate an algorithm to color the graph's vertices with two different colors and return true if success, otherwise return false. We given each vertex a number to link it with the wrestler who has the same number. Then we will finally return an array A that A[i] is the color of vertices i(i.e. the side we choose for the wrestler i, 0 is one side and 1 is the other).

```
bfs_color (start) -- pseudocode
  queue = new Queue()
  queue.enqueue(start)
  //define two colors are 0 and 1
  //define -1 means the vertices hasn't been colored
  start.color = 0
  while not queue.is_empty():
      u = queue.dequeue()
      for each v in u's adjacency list:
          if v.color == -1:
              queue.enqueue(v)
              v.color = 1 - u.color
          else if v.color == u.color:
              return false
```

```
  return true

color_whole_graph (vertices_array) -- pseudocode
  for v in vertices_array:
      if v.color == -1:
          if bfs_color (v) == false:
              return false
  return true

generate_solution (vertices_array, vertices_num) -- pseudocode
  if color_whole_graph(vertices_array):
      //it is possible to divide the wrestlers into two sides
      result[vertices_num]
      //set every elements in result be -1
      for i in range(n):
          result[i] = vertices_array[i].color
      return result
  // it is impossible to divide the wrestlers into two sides
  // no array to output
```

bfs_color (start) is an $O(n + r)$-time algorithm
color_whole_graph (vertices_array) is also an $O(n + r)$-time algorithm
generate_solution (vertices_array, vertices_num) is an $O(n)$-time algorithm
So the whole algorithm is $O(n + r)$.

Q_6

1.

a	$b \to \cdot$
b	$a \to \cdot$
c	$b \to g \to \cdot$
d	$c \to \cdot$
e	$a \to f \to \cdot$
f	$a \to c \to \cdot$
g	$f \to h \to \cdot$
h	$d \to \cdot$

2.

$d(a) = 1, d(b) = 2, d(c) = 3, d(d) = 4$

$d(e) = 11, d(f) = 10, d(g) = 6, d(h) = 5$

$f(a) = 16, f(b) = 15, f(c) = 14, f(d) = 9$

$f(e) = 12, f(f) = 13, f(g) = 7, f(h) = 8$

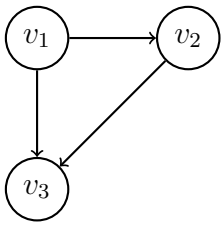
$R = [a, b, c, f, e, d, h, g]$

3.

$[a, b], [c, g, f, h, d], [e]$

4.

$v_1 = [a, b], v_2 = [c, g, f, h, d], v_3 = [e]$



5.

PF:

For a graph G, it has the same SCCs with its transpose graph.

Now we prove this statement.

Consider an arbitrary SCC in G, we call it C_1. Then all the points in this SCC is reachable from each other. Suppose the points are $\{u_1, u_2, ..., u_n\}$. Consider two arbitrary vertices from this set u_i, u_j. We have path p_i from u_i to u_j and path p_j from u_j to u_i. In G^T, as all the edge is reversed, the path p_i and p_j that are make up of several edges are also reversed in G^T. Let p_i is reversed to p_i' and p_j is reversed to p_j'. Then p_i' become the path from u_j to u_i and p_j' become the path from u_i to u_j. Thus, u_i and u_j are still reachable from each other in G^T. As u_i and u_j are arbitrary, all the point in C_1 are still reachable from each other in G^T. So $\{u_1, u_2, ..., u_n\}$ can still be in an SCC C_1'in G^T. Then we need to prove that C_1' will not have more new vertices. We prove it from contradiction. Suppose there is a new vertex w in C_1'. Therefore, w and vertices in $\{u_1, u_2, ..., u_n\}$ are reachable from each other. We choose w and an arbitrary $u_k(k \in \{1, 2, ...n\})$ to discuss. As in G^T,we have path p_w from w to u_k and path p_k from u_k to w. In G, path p_w will reverse and become the path from u_k to w, and path p_k will reverse and become the path from w to u_k. This means w and u_k are also reachable from each other. As u_k is arbitrary, w and vertices in $\{u_1, u_2, ..., u_n\}$ are also reachable from each other in G. Then the SCC C_1 of G is not the maximal subset of vertices reachable from each other in G, which is contradict to the definition of SCC. So our supposition is false and C_1' will not have more new vertices. Then we have $C_1 = C_1'$. As C_1 is arbitrary, all the SCCs C_t in G will have a corresponding C_t' in G^T that have the same vertices. Finally, we have

the statement that for a graph G, it has the same SCCs with its transpose graph as wanted.

Therefore, the component graphs of these two graph will have the same vertices.

Consider an arbitrary edge (v_i, v_j) in G^{SCC}.

Now we know that there is some vertex u in v_i has an edge to some vertex v in v_j, and this edge will reverse to let the vertex v in v_j go to the vertex u in v_i if the graph G transposes to G^T. This will cause $(G^T)^{SCC}$ will have an edge (v_j, v_i). As the edge we choose is arbitrary, all the edge in G^{SCC} will be reversed to be the new edge in $(G^T)^{SCC}$ and no new edge will generate in $(G^T)^{SCC}$, or the G^{SCC} has to have the original reversed one. Therefore, $(G^T)^{SCC}$ have the same vertices as G^{SCC} and the edges are the reverse of G^{SCC}'s. This means $(G^T)^{SCC}$ is the transpose of G^{SCC} and also G^{SCC} is the transpose of $(G^T)^{SCC}$. Then we have $((G^T)^{SCC})^T = G^{SCC}$ as wanted.

QED.

6.

The modified algorithm doesn't work.

Suppose we start to visit a vertex v_1 whose SCC C_1 is linked by an edge from some vertex in another SCC C_2. Then the minimum discovery time in C_2 is larger than the minimum discovery time in C_1, as dfs(v_1) will visit C_1 and the SCCs it link to first. Therfore, in the discovery time queue, the first vertex of C_2 is after the first vertex of C_1. So in the DFS(G^T), we will visit C_1 first. However in the original graph G, C_1 is linked by an edge from some vertex in another SCC C_2, which means C_1 will linked to some vertex in C_2 in graph G^T. Thus, dfs(v_1) will put both C_1 and C_2 in one SCC list. This is

unacceptable, so the modified algorithm doesn't work.