Step 1. Identification of the problem.

Identification of necessities.

- They need to be able to register 7 to 10 hours riders.
- They need to order hours rides in order of registration.
- They need to register the bettors when the racecourse doors open.
- The registration time of bettors is 3 minutes.
- it must be possible to visualize the winners of the race.
- Users should be able to quickly check, from their ID, the record of their bet where it should also appear now if their horse won or lost the race.
- The racecourse needs a rematch option that be able to create a new race.
- if the rematch button was used, then the rider who won the race must be in the last position and the last one to arrive will be in first place.

Problem definition.

- The “El indomable spirit” need to incorporate a system where the flow of the entire racecourse operation can be managed.

Step 2. Information gathering.

Definitions

Racecourse:

The racecourse is an arena suitable for horse racing. The interior has bleachers on the perimeter, and the center is made of earth or grass. In the center there is an oval bordering the steps that form the track. Horse races are played on the track. The tracks can be dirt or grass.

Career:

A race is a speed competition, in which competitors must complete a certain path or distance using the shortest possible time or travel if possible, in a certain fixed time.

Stack:

Set of things placed on top of each other as a pillar or column.

Step 3. Search for creative solutions.

Alternative 1.

Use structures such as queues together with the hash table to be able to handle the process flow of the racetrack, allowing access to each node efficiently.

Alternative 2.

Create our own structures to do everything we need efficiently, allowing to order the horses in a stack and the bettors in a hash table.

Alternative 3.

Use own java structures that allow us to carry out the management we need, be it an arraylist or something similar, adding the attributes dynamically.

Step 4. Transition from Ideas to Preliminary Designs.

Alternative 1.

- It is good, but since it has to be implemented from java.util it will have many methods that will not help us.
- It will not allow us to know how this type of structure works inside.
- As it is not a generic solution, it would be necessary to implement it repeatedly to use it.

Alternative 2.

- It is a generic solution, which allows us to reuse it.
- It allows us to know how these structures work inside.
- There is nothing to import.

Step 5. Evaluation and Selection of the Best Solution.

Criterion A. Make it a generic solution:

- Code can be reused for other classes [4]
- Cannot reuse code for other classes [2]

Criterion B. Be efficient for the search:

- When searching is O(1) [2]
- When searching is different from O(1) [1].

Criterion C. Added as expected:

- Aggregated nodes are best suited for use [4]
- Tap on add additional methods to be able to use the nodes [1]

	Criterion A	Criterion B	Criterion C	Total
Alternative 1	2	1	4	7
Alternative 2	4	2	4	10

Selection:

We choose choice two because it is the one that best suits our needs, allowing us to solve everything we need to do in the most efficient way

Step 6. Preparation of Reports and Specifications.

Problem Specification:

Problem: manage the flow of the racecourse.

Input: An arrangement of riders such that A={a1,a2,…,an} /$7 \leq n \leq 10$ and an arrangement of bettors B={b1,b2,…,bn}.

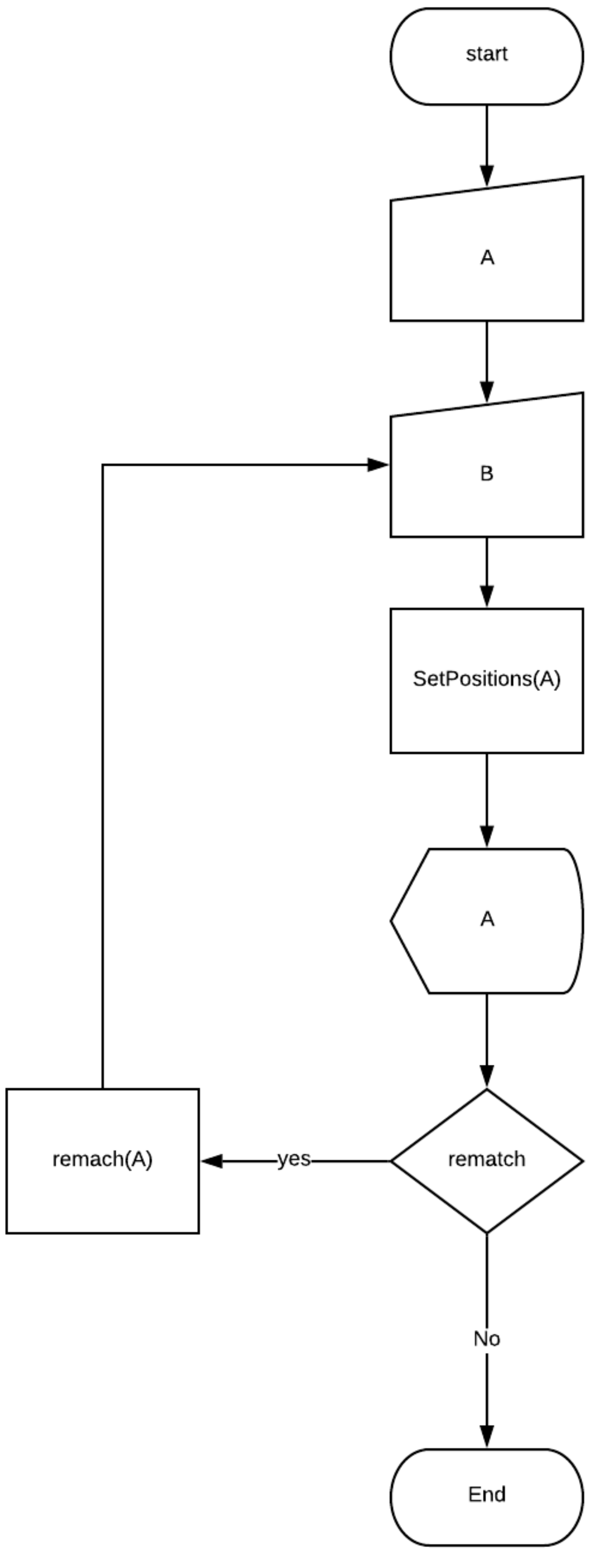
Output: An arrangement of riders in the order of first to arrive A={ a1,a2,…,an}.

```
setPosition(A)

print A

if(rematch == yes)

rematch(A)
```



Step 7. Design Implementation.

List of Tasks to implement:

- Add riders.
- Add bettors
- visualize the podium
- check the bet log.
- Rematch

Subroutine Specifications:

Name:	R1
Description:	Add riders in order of arrival
Input:	•Horse rider´s name •Horse´s name •Track
Output	< none>

Name:	R2
Description:	Add bettors in less than 3 minutes
Input:	•NIT •Name •HorseRider •Amount
Output	< none>

Name:	R3
Description:	Visualize the riders' podium
Input:	<none>
Output	Horsemen Arrangement

Name:	R4
Description:	Show the bet made by a bettor
Input:	NIT
Output	A bettor

Name:	R5
Description:	Lets make a rematch accommodating the rider who won in the last position and the last one who came first
Input:	<none>
Output	<none>

```
public void addHorseRider(HorseRider hr) {

                horseRiders.enqueue(hr);

        }

public void addBettor(Bettor b) {

                bettors.insert(b, b.getNit());

        }

public void run() {

        try {

        Thread.sleep(180000);

        } catch (InterruptedException e) {

        e.printStackTrace();

        }

}
public boolean consultBet(String
nit) {
                Bettor b =
bettors.search(nit);
                if(b == null) {
                        return false;
                }
                String horseName =
b.getHorse();
                boolean won = false;
                Iqueue<HorseRider> aux =
new Pqueue<HorseRider>();

        while(!horseRiders.isEmpty())
{
                        HorseRider hr =
null;
                        try {
                                hr =
horseRiders.dequeue();
                        } catch
(Exception e) {

                        }

        if(hr.getHorseName().equals(ho
rseName)) {

        if(hr.getPosition() == 1) {
                                        won
= true;
                                }
                        }
                        aux.enqueue(hr);
                }
                setHorseRiders(aux);
                return won;
        }

public void rematch() {
                bettors = new
HashTable<Bettor>();
                int last =
horseRiders.size();
                int tracks = 1;
                Stack<HorseRider> st =
new Stack<HorseRider>();
                Iqueue<HorseRider> queue
= new Pqueue<HorseRider>();
                boolean finished =
false;
                while(!finished) {

        while(!horseRiders.isEmpty())
{
                                HorseRider
hr = null;
                                try {
                                        hr =
horseRiders.dequeue();
                                } catch
(Exception e) {
                                                        //
TODO Auto-generated catch block

        e.printStackTrace();
                                                }

        if(hr.getPosition() == last) {

        queue.enqueue(hr);

        hr.setTrack(tracks);

        tracks++;

        last--;
                                                } else {

        st.push(hr);
                                                }
                                        }
if(!st.isEmpty()) {

        while(!st.isEmpty()) {

        HorseRider hr = st.pop();

        if(hr.getPosition() == last) {

        queue.enqueue(hr);

        hr.setTrack(tracks);

        tracks++;

        last--;
                                                        }
else {

        horseRiders.enqueue(hr);
                                                        }
                                                }
                                        }
                                        if(st.isEmpty()
&& horseRiders.isEmpty()) {
                                                finished =
true;
                                        }
                                }
                                horseRiders = queue;
                        }
```