

LABORATORY 2

GONZALO DE VARONA

JUAN JOSÉ RESTREPO

A00358687

A00359137

ALGORITHMS AND DATA STRUCTURES

ICESI UNIVERSITY

MARCH 24 2020

Phase 1. Problem identification

A business needs to manage, create, delete, its sports bets, on horse racing in track, besides, it needs to manage the information for each match, and even rematch.

Phase 2. Gathering information

Horse or equestrian bets have two general modalities, turf and jumps. Turf racing is very popular all over the world, but especially in the United States where there are no jumps. They run on oval-shaped tracks either on land or grass called the racecourse. It is important to make that distinction because horses that run on land do not usually perform well on grass, and vice versa.

In England both exist, being the extremely popular jumps while in the United States only those of turf are run. The routes are between 2000 and 4000 meters. In turf only one complete circuit is made, while in the jumps 2 or 3 circuits are usually made. During the jumps the riders only have to jump some obstacles once, while other obstacles of greater difficulty, usually those that are close to the goal, can be omitted in the final 2 rounds..

Categories:

Maiden Race or Inaugural Weight - It is the initial auction where the foals are exposed to be purchased. It is his first career and during the same any buyer can make an offer.

Starter: Horses that are beginners, of which the winners cannot participate already won once

Allowance or Conditional: These are certain horses that must meet certain requirements such as age, weight, gender, number of races won.

Stake or Classic: It's the big leagues, the stakes come in three grades; Grade 1, Grade 2 and Grade 3. In Grade 1, the most winning horses from around the world or region compete.

Types:

Winner or Straight: You bet that horse will win the race.

Second position or Place: Bet on which horse comes second.

Third Position or Show: Betting on which horse comes from third.

In the Money: Betting that a horse arrives in first, second or third place.

Placed: It is not a bet that is commonly offered, but you can choose in which position the horse is not the first 3.

Exact: One of the most popular bets for its high payout, the exact ones ask that you choose which horse will arrive first and which second. You have to hit both.

Imperfect: The same as an exact one but the chosen horses can arrive in any order between first and second place.

Trifectas: As exact but with 3 horses. Exactly the order of the winner, second and third. High payment.

Fourfold: As exact but with the first 4 horses.

Doubles or Daily Double: Bet on two different races and you must hit the winners of both races.

Triplet or Pick3: As Doubles, but in 3 races you must hit all the winners.

Pick 4: As Doubles but 4 races you must hit the winners.

Pick 5 and Pick 6: Same as Doubles, but in 5 or 6 races.

Phase 3. Seek for creative solutions

Idea 1: Use Java libraries to implement data structures like Queue, Stack, Hash Table in order to model the problem.

Idea 2: Create our own data structures like a Queue, Stack, Hash Tables by making them from scratch from their ideal methods for each data structure.

Idea 3: Use data structures that we know very well such as Array, Vector and ArrayList but aren't as efficient as other data structures for this problem.

Idea 4: Implement a well done GUI that simulates the horses race, from the beginning to the end of the race, with detailed simulation of every second of the race

Idea 5: Creating random horses when starting a new match, in order to save time so the user does not need to add horse by horse.

Idea 6: Setting the final positions in a random way, so the program does not always repeat the same position for each row. This could be very useful in casinos.

Idea 7: Using threads to make a 3 minute timer when there is at least 7 horses registered to race.

Phase 4. Transition from ideas to preliminary designs

Idea 1: *Discarded.* Java libraries can implement what we are looking for, with *Stack class* or *Queue interface* however it can not be seen how they specifically work, so the whole point of analyzing and manipulating Queues, Stacks and Hash tables will make no sense.

Idea 2: Create our data structures from scratch is a very simple and effective way to analyze Queues, Stacks and Hash tables; besides we have total control of how they work, in case we need to do something else with them that the Java libraries do not do. In the other hand, we know how they work and it makes more easy the implementation and it reduces the working time, because we can modify the methods and logic easier.

Idea 3: *Discarded.* Although we know really good ArrayList and Array data structures, they are not efficient nor good enough to work with in this specific problem, so they are not the ideal data structures for the job.

Idea 4: *Discarded.* Considering that a nice, friendly and pleasant GUI is important for programs and final users, we had it in mind, however, making the best and detailed GUI would take way more time and resources than what it already took to get over with the project, so we chose not to do it that way since it is not seem to be too much important for the client, as a matter of fact, it is never said to us that GUI had to be like some top of the application. Even though we made a decent GUI that gets the job done.

Idea 5: Thinking in the future of the program and its use, we find it interesting to give some participation to the odds in this project, so in case the Casino wants to take its game indoors with the same idea of horse racing but wants to make it quick, we developed the option "*New Match*" in which the program itself generates random horses with random horsemen, so it takes less time to get started with the race, and there are more races happening than it would be if every race info is filled manually with "real competitors", this means more bets and more earnings for the Casino, and another way of horse racing without actually "making a race".

Idea 6: Having in mind that we don't have a way to put an end to a race, because they are not happening in real life and this program was made for a Casino, setting the positions for horses in a random way for each race seem to be the most proper way to set an end for each race, guaranteeing randomness as it must be in a Casino. Besides, it makes gambling more "fair" due to gamblers can not establish patterns for winners per race.

Idea 7: We need to use threads for the project, because they do tasks that makes the perception the user like stuff is happening in real time. So threads are the best option to make a 3 minute timer and make it seem like it is counting each second in real time.

Phase 5. Evaluation and selection of the best solutions

Item 1: Course objectives

[0] The solution does not focuses on developing and analyzing Queues, Stacks and Hash Tables

[1] The solution focuses on developing and analyzing Queues, Stacks and Hash Tables

[2] Is easy to manipulate, develop and analyze Queues, Stacks and Hash Tables, in order to solve the problem.

Item 2: Ease of implementation

[0] The solution uses operations that are difficult to understand or implement due we've not seen it or worked with it before much.

[1] The solution uses basic operations to solve the problem.

[2] The solution uses basic operations to solve the problem and can be implemented in a compact and short method.

Item 3: Exact solution

[0] The algorithm gives an approximated solution

[1] The algorithm gives an exact solution.

Item 4: Usefulness for Casino

[0] The solution does not implies randomness, needed in a gambling game.

[1] The solution implies randomness, needed in a gambling game.

[2] The solution implies randomness, needed in a gambling game and allows the client to make the most profit out of it.

Ideas feasibility table

Idea	Item 1	Item 2	Item 3	Item 4	Total
Idea 2	2	1	1	0	4
Idea 5	0	1	0	2	3
Idea 6	0	1	1	2	4
Idea 7	0	2	1	0	3

Functional Requirements

F.R.1	Add horse to the race
Abstract	The application is able to add a horse to the race.
Input	-Horseman's name -Horse's name
Output	Horse is added

FR.2	Search horse
Abstract	The application is able to search horse's information by name, information such as horseman's name, horse's name, and horse's row number
Input	-Horse name
Output	Horse's information is shown

FR.3	Add gamble
Abstract	The application is able to add a gamble.
Input	-ID of the gambler -Name of the gambler -Amount of money to gamble

	-Supposed winner
Output	Gamble is added

FR.4	Search gamble
Abstract	The application is able to search horse's information by name, information such as horseman's name, horse's name, and horse's row number
Input	-ID of the gambler -Name of the gambler -Amount of money to gamble -Supposed winner
Output	Gamble is added

FR.5	Rematch race
Abstract	The application is able to create a rematch race for the previous race
Input	-
Output	Rematch happens

Non-Functional Requirements

NFR.1	Limit of horses per race
Abstract	There can only be from 7 to 10 horses registered and able to race.
Output	7 to 10 horses can participate in the race

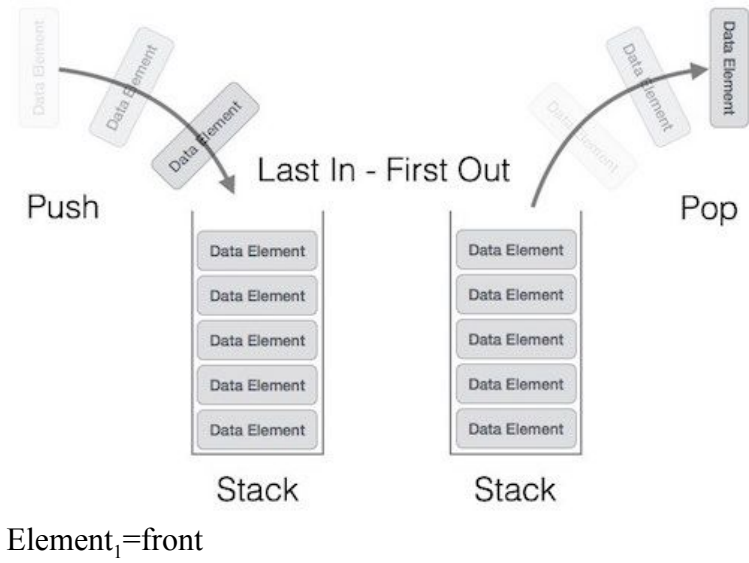
NFR.2	Horses row is determined by arrival order
Abstract	The row for each horse is determined by arrival order, first horse gets first row, second horse gets second row, etc
Output	Row number for each horse

NFR.3	Rematch row determined by podium
Abstract	Rows for each horse on the rematch are determined by the podium on the previous race, the best place, gets the last row on rematch and the worst place gets the first row on the rematch
Output	Rows for each horse on the rematch

NFR.4	Rematch row determined by podium
Abstract	Rows for each horse on the rematch are determined by the podium on the previous race, the best place, gets the last row on rematch and the worst place gets the first row on the rematch
Output	Rows for each horse on the rematch

NFR.5	Information shown for each horse
Abstract	When a horse is added to the race, its name, horseman's name, and row must be visible in the information of the current race
Output	Rows for each horse on the rematch

ADT

Name	Stack
Abstract representation	 <p>Element_l=front</p>
Invariant	size ≥ 0
Operations	Stack \rightarrow Stack add:element unstack \rightarrow element, size,clear size \rightarrow integer clear isEmpty \rightarrow boolean

add:element

“Add an element at the top of the list”

Pre: element \neq NIL

Pos: Add the element at the top (front) of the Stack

unstack → **Element**

“Delete the first element in the top of the Stack”

Pre: $\text{Element}_1 \neq \text{NIL}$

Pos: Remove the element in the top (front) of the Stack

size → **integer**

“Return the size of the Stack”

Pre: $\text{Stack.Size} \geq 0$

Pos: Returns number of items in the Stack

isEmpty → **boolean**

“Return true if the Stack is empty and false otherwise ”

Pre: none

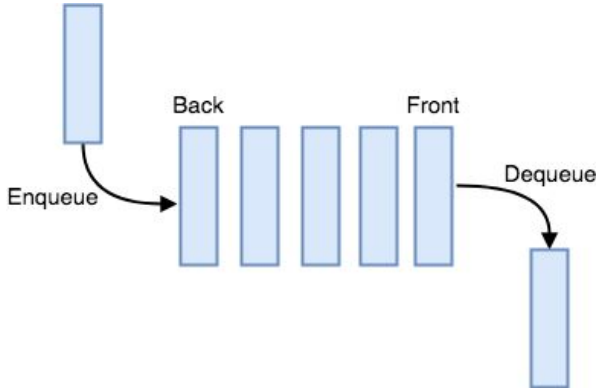
Pos: Returns true if the queue is empty, otherwise returns false.

clear

“Delete all of the elements in the Stack”

Pre: none

Pos: $\text{Stack.size} = 0$

Name	Queue
Abstract representation	 <p>each rectangle = Element $\text{Element}_1 = \text{front}$ $\text{Element}_{\text{size}} = \text{back}$</p>
Invariant	$\text{size} \geq 0$
Operations	Queue \rightarrow Queue enqueue:element dequeue \rightarrow element size \rightarrow integer clear isEmpty \rightarrow boolean

enqueue:Element

“Insert a new element in the end (back) of the Queue”

Pre: Element \neq NIL

Pos: The element is inserted in the end (back) of the Queue

dequeue → **Element**

“Delete a the element that is in the top (front) of the Queue”

Pre: $\text{Element}_1 \neq \text{NIL}$

Pos: Retrieves and removes the item at the top (front) of the Queue.

Size → **integer**

“Return the size of the list”

Pre: The list ≥ 0

Pos: Returns number of items in a list

isEmpty

“Return true or false if the list is empty or not”

Pre: None

Pos: Returns true if the queue is empty, otherwise returns false.

Clear

“Delete all of the elements in the list”

Pre: none

Pos: $\text{Queue.size} = 0$

Name	Hash Table
Abstract representation	<p>HashTable</p> <pre> graph LR subgraph HashTable direction TB S0[0: #] S1[1: #] S2[2: #] S3[3: #] S4[4: #] S5[5: #] S6[6: #] S7[7: #] end S0 --> N0["#"] S3 --> N3["#"] N3 --> N11["11"] N11 --> N27["27"] N27 --> N19["# 19"] S6 --> N6["#"] N6 --> N22["22"] N22 --> N6_6["# 6"] </pre>
Invariant	$\{size \geq 0\} \wedge \{ \forall i, j / a \in keys, a_i \neq a_j, i \geq 0, j \geq 0, i \neq j \}$
Operations	HashTable \rightarrow HashTable add:(key,element) remove \rightarrow element search:key \rightarrow element get:key \rightarrow element clear

add:(Key,element)

“Insert the element in a specific position using the Key”

Pre: index indicates the position that the element will be insert

Pos: Insert the element in the position, if in this position is another element, using chaining

to insert the element in this position

remove:

“Remove the element in a specific position using the Key”

Pre: hash Function(key) = position

Pos: Remove the element, if the position doesn't have any element in this position, do nothing.

Search:Key → Element

“Search a element using the key in the list”

Pre: hash Function(key) = position

Pos: Return the element if that is exists.

get:key → Element

“Search a element using the key in the list”

Pre: hash Function(key) = position

Pos: Return the element if that is exists.

clear

“Delete all of the elements in the list”

Pre: none

Pos: HashTable.size = 0

Bibliography

¿Cómo funcionan las apuestas hípicas?. Trébol Apuestas Deportivas. Retrieved 7 March 2020, from <https://www.trebol-apuestas.com/como-funcionan-las-apuestas-hipicas/>