Esteban Ariza Acosta, Johan Sebastián Giraldo Rubio y Mateo Valdés Otero

Lab 2 Documentation

1. Identifying the problem

The situation: A company has ventured into the horse racing business and has launched a new racecourse. However, it does not yet have a way to keep track of all of the racecourse's operations.

The company is in trouble because:

- The company needs a way to know the name of the horses competing, the name of the jockeys and the order in which they signed up
- The company needs to keep track of what horse won the race and who the next racers will be
- The company needs a way to keep track of the bets and the payouts

If the company does not have a way to keep track of all of this then managing the horse racing business might prove to be extremely difficult.

Problem: There is not a way to keep track of all of the racecourse's operations.

2. Research

Functional requirements:

Name	FR1: Register rider and horse
Summary	The system allows registering a rider with its respective horse. There must be between 7 and 10 riders (with their respective horses) and their lane is defined by their registration order.
Input	The **rider name** and the **horse name**
Output	The rider and the horse were registered successfully.

Name	FR2: Register User
Summary	The system allows registering a user, the users only can be registered for 3 minutes.

Input	(1) User **Id**, (2) User **name**, (3) **horse** which the user bet for, (4) **money** betted by the user
Output	The user was registered successfully

Name	FR3: Show Winners
Summary	The system allows showing a podium with the 3 first places
Input	-
Output	A podium with the 3 first riders with their horses

Name	FR4: Search User by Id
Summary	The system allows to find a user by his id and show if the horse of the user won or not.
Input	The user **id**
Output	The user was found and shows if the horse of the user won or not successfully

Name	FR5: Rematch
Summary	The system allows doing a rematch where the race is repeated with the same riders and horses and the lanes are inverted (The horse in the first lane in the last race is now in the in the last lane) and the bets are opened again.
Input	-
Output	The race is repeated with the inverted lanes and the bets are opened again successfully

Name	FR6: Start new race

Summary	The system allows starting a new race.
Input	-
Output	A new race started successfully

Stacks: Stacks are data structures wherein the last element that enters is the first element that leaves (LIFO, or Last In First Out).

Queues: Queues are data structures wherein the first elements that enters is the first element that leaves (FIFO, or First In First Out)

Hash Tables: Hash tables are data structures that map keys to values. In the Java programming language, keys and values can be any object that is not null. The hash tables use a hash function to map the key to the value. This means that to retrieve a value the user must enter the key associated with that value.

A problem that might arise with hash tables are collisions, which happen when a given key has more than 2 values associated with it. These problems are solved with open addressing and chaining. In open addressing, the second value is stored in another address. In chaining, both values are stored in a linked list.

Horse racing betting: Horse racing betting usually works in the following manner: the company in charge of managing the best collects money from the betters first. A certain percentage of this money (normally between 14% and 25%) is reserved to pay for operating costs, management, taxes, etc. The rest is divided and equal parts are paid to the people who wagered on the horse who won. It is common to represent the payout as “odds”. Odds of “3-1” means that a person who wagered $1 will be paid back their original amount ($1) plus $3 extra.

References:

Narang, Mehak. Stack Class in Java. GeeksForGeeks.
https://www.geeksforgeeks.org/stack-class-in-java/

Mahrsee, Rishabh. Queue Interface in Java. GeeksForGeeks.
https://www.geeksforgeeks.org/queue-interface-java/

Verma, Abhishek. Hashtable in Java. GeeksForGeeks.
https://www.geeksforgeeks.org/hashtable-in-java/

Horse Racing History. Winningponies.com.
https://www.winningponies.com/horse-racing-history.html

3. **Creative solutions**

Brainstorming:

Alternative #1
Hire more employees to keep track of the racetrack operations manually using a notebook.

Alternative #2
Use an online database to save the racetrack's important data and manage the racetrack operations.

Alternative #3
Make the employees keep track of all the racecourse's operations using a basic text-editing program such as Microsoft Notepad

Alternative #4
Make the employees keep track of all the racecourse's operations using Microsoft Office, Google docs, or any similar third party software that lets many employees edit a single document.

Alternative #5
Create a program from scratch to manage and keep track of all the racecourse's operations digitally.

Alternative #6
Create a program using a website to keep track of all the racecourse's operations online.

Alternative #7
Outsource the job of keeping track of the racecourse's operations to another country where labor is cheaper and possibly more efficient.

Alternative #8
Use a third party software specialized in managing these types of businesses instead of creating the program from scratch.

4. **Transformation to preliminary designs**

Alternative #1
The company might not have enough funds to pay the salaries and benefits of the new employees. The new employees might also need training or assistance, and it might be hard to manage everything using only notebooks. The notebooks might also take up a lot of space after

a while. This solution soaks up a lot of time, effort and money, so we will discard it and look for a better alternative.

Alternative #2
Having an online database might be costly and if the internet or the servers are down then the company will be in trouble because it will not be able to manage the racecourse properly. Therefore we must discard this alternative and look for a better one.

Alternative #3
Since with software such as Notepad only one person can edit at a time, it might be very difficult for one person to track all of the operations, and this could lead to problems later on. Thus we discard this alternative and look for a better one.

Alternative #4
Since software like Microsoft Excel or Google Spreadsheets allow many people to work on a single document, this option might be viable because a team can work on keeping track of the operations in an organized and efficient manner, and everything could be saved digitally. All of this indicates that this is a viable option.

Alternative #5
Since the program that this business is not extremely difficult or complicated to create, this might be a viable option because developing a viable program might be cheaper and preferable than using a third party program. This is definitely a viable alternative.

Alternative #6
This alternative is viable because of all the reasons Alternative #5 is viable, plus using the internet makes the registration process easier. This is also a viable alternative.

Alternative #7
Although outsourcing could be cheaper than hiring new employees, it could still be costly and might bring unintended consequences if the outsourcing company has any sort of complication.

Alternative #8
Since good third party software can be greatly effective in helping to manage all sorts of businesses, this alternative must be considered.

5. Evaluation and selection of the best solution

Criterion B: The difficulty of utilization.

- [3] The solution is very intuitive and dynamic.
- [2] The solution is relatively easy to use.
- [1] The solution is even difficult for the most experienced professionals.

Criterion A: The security of the information.

- [3] The information is very secure and can not be hacked.
- [2] The information is secure enough.
- [1] The information can be modified easily by an external entity.

Criterion C: The difficulty of implementation.

- [3] The implementation is simple, even trivial
- [2] Average people can implement this solution.
- [1] Very difficult to actually implement

Criterion D: The profitability of the solution.

- [3] The solution does not require a lot of money.
- [2] The money spent in the solution is fair enough.
- [1] The solution is too expensive.

Solution	**Criterion A**	**Criterion B**	**Criterion C**	**Criterion D**	**Total**
Excel	2	1	3	3	9
Own Program	3	3	2	2	10
Website	3	2	1	1	8
Third Party Program	3	3	2	1	9

Justification:

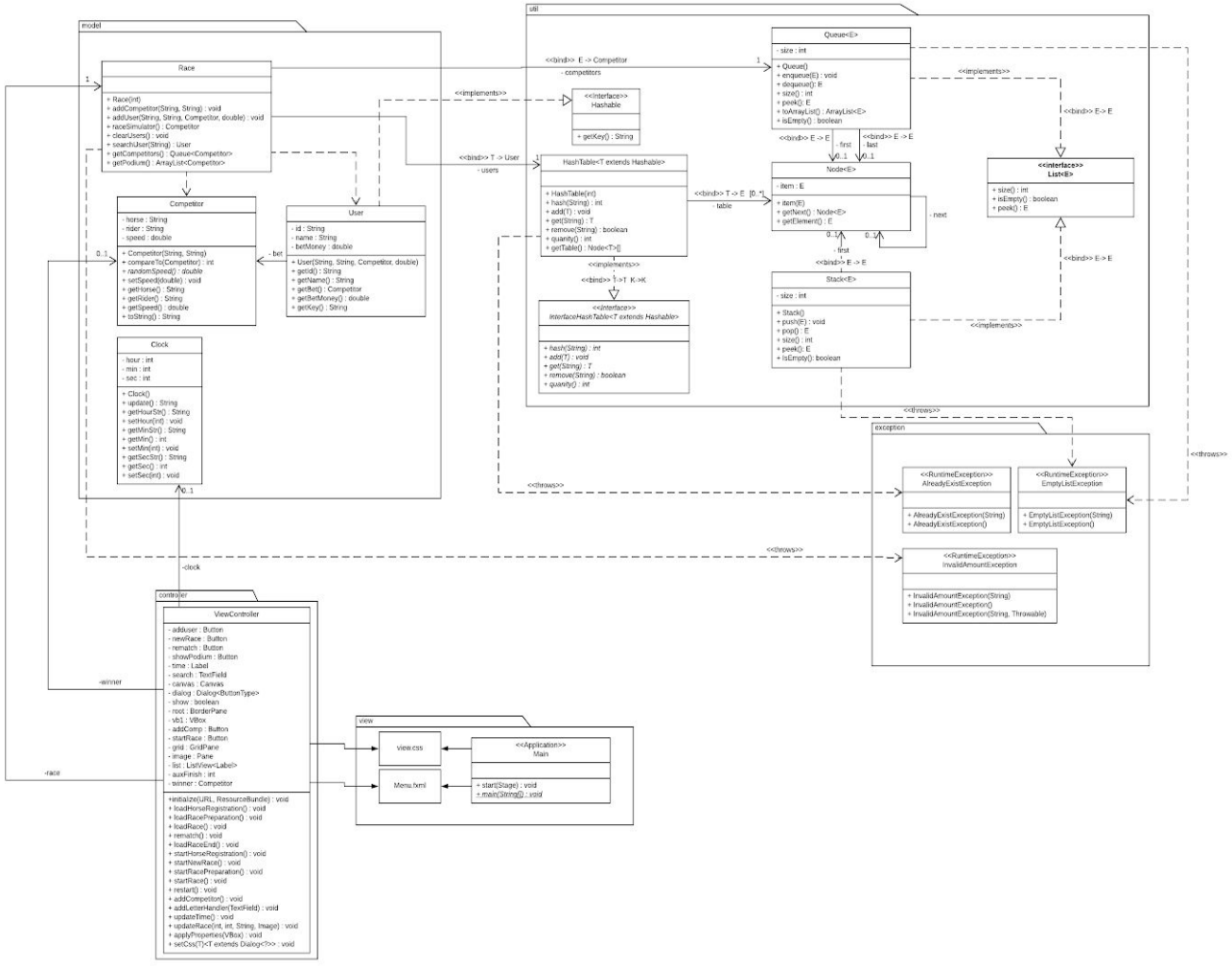
- Using excel is a good alternative, but is not secure and intuitive for the users.
- Creating our own program can be a good solution if the enterprise has very well prepared software engineers in their installations.
- A Website is a good solution for the users, but is very expensive and difficult to maintain.
- Finally, an external company software has the same advantages of the second solution, excluding the money they need to expend for an existing program.

Final decision:

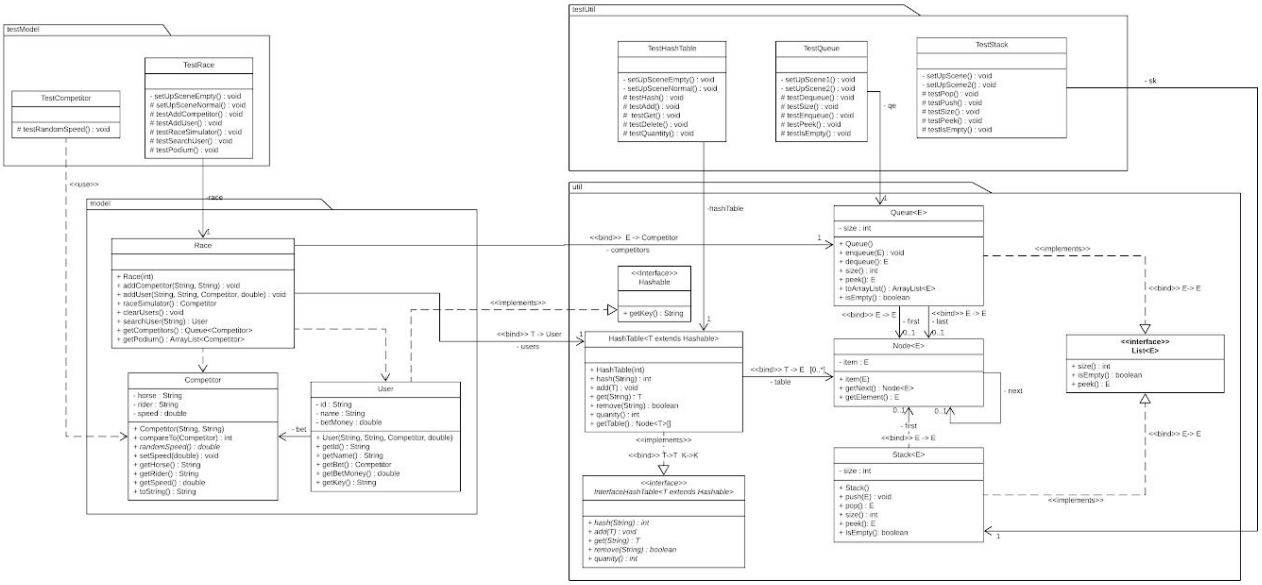
The company relies on some well experienced software engineers, that's why the second solution (Own Program) is the best for the enterprise.

6. Evaluation and selection of the best solution

Class Diagram



Test Class Diagram



ADT

ADT Queue
Queue $Q = \ < a_1, a_2, \ \ldots \ , \ a_{n-1}, \ a_n >, front \ = a_1 \wedge back \ = a_n$
$\lvert Q \rvert \geq 0$
Primitive operations: • CreateQueue: → Queue : Creator • Enqueue: Queue x Element → Queue : Mutator • Dequeue: Queue → Queue : Mutator

• Size:	Queue	→	Integer : Observer
• Peek:	Queue	→	Element : Observer
• Empty:	Queue	→	Boolean : Observer

CreateQueue()

"Creates a queue"

{ pre: true }
{ post: $\exists Q \wedge Q = \{\}$ }

Enqueue(queue, element**)**

"Adds an element in the end of the queue"
{ pre: $\exists Q$ }
{ post: $e \in Q$ }

Dequeue(queue**)**

"Removes the first element in the queue"

{ pre: $\exists Q \wedge Q \neq \{\}$ }
{ post: $e \notin Q \wedge e$ }

Size(queue**)**

"Returns the size of the queue"

{ pre: $\exists Q$ }
{ post: $|Q|$ }

Peek(queue**)**

"Returns the front of the queue"

{ pre: $\exists Q \wedge Q \neq \{\}$ }
{ post: a_1 }

Empty(queue**)**

"Evaluates if the queue is empty"

{ pre: $\exists Q$ }
{ post: $V \in \{true, false\}$ }

ADT Stack

Stack $S = < a_n, a_{n-1}, \ldots, a_2, a_1 >, top = a_n$

$|S| \geq 0$

Primitive operations:

• CreateStack:		→	Stack : Creator
• Push:	Stack x Element	→	Stack : Mutator
• Pop:	Stack	→	Stack: Mutator
• Size:	Stack	→	Integer: Observer
• Peek:	Stack	→	Element : Observer
• Empty:	Stack	→	Boolean : Observer

CreateStack()

"Creates a Stack"

{ pre: true }
{ post: $\exists S \wedge S = \{\}$ }

Push(stack, element**)**

"Adds an element in the top"

{ pre: $\exists S$ }
{ post: $e \in S$ }

Pop(stack**)**

"Removes the element in the top"

{ pre: $\exists S \wedge S \neq \{\}$ }
{ post: $e \notin S \wedge e$ }

Size(stack**)**

"Returns the size of the stack"

{ pre: $\exists S$ }
{ post: $|S|$ }

Peek(stack**)**

"Returns the element in the top"

{ pre: $\exists S \wedge S \neq \{\}$ }
{ post: a_n }

Empty(stack**)**

"Evaluates if the stack is empty"

{ pre: $\exists S$ }
{ post: $V \in \{true, false\}$ }

ADT Hash Table

Hash Table $T = < a_1, a_2, \ldots, a_{n-1}, a_n >$, $a_i = \{(e_n, k_n), (e_{n-1}, k_{n-1}), \ldots, (e_2, k_2), (e_1, k_1)\}$

$|T| \geq 0 \wedge T[h(k_i)] = e_i \wedge |a_i| \geq 0 \wedge \{ki \neq kj \mid j \neq i\}$

Primitive operations:

- CreateHashTable: Integer → HashTable
- Hash HashTable x Key → Integer
- Add HashTable x Element → HashTable
- Get HashTable x Key → Element
- Remove HashTable x Key → HashTable
- Quantity HashTable → Integer

CreateHashTable(Integer**)**

"Create a Hash Table with specific table size."

{ pre: Size > 0 }

{ post: $\exists T \ \wedge \ T = \ < a_1, a_2, \ \ldots \ , \ a_{n-1}, \ a_n > \ \wedge \ a_i = \{\}$ }

Hash(HashTable, Key**)**

“Transform a key into an index value that represents the position.”

{ pre: { $\exists T \ \mid \ |Key \ Text| \ \geq \ 1$ } }
{ post: {index | index $\varepsilon \ Z+$ } }

Add(HashTable, Element**)**

“Add a not repeated element into the hash table in a position specified by the key.”

{ pre: { $\exists T$, $\exists e \ \mid e \neq e_i$, $\exists e \ .key$, $|k \ text| \geq 1$ } }
{ post: $e \ \varepsilon \ T \ \wedge \ T[h(k)] = e$ }

Get(HashTable, Key**)**

“Return an element with the specified key. If there is not an element with this key, the method does not return an element.”

{ pre: $\exists T \ \wedge \ |Key \ Text| \ \geq \ 1$ }
{ post: $e_{h(k)}$ }

Remove(HashTable, Key**)**

“Remove an element from the hash table with a specified key.”

{ pre: $\exists T \wedge \ |Key \ Text| \ \geq \ 1$ }
{ post: $e_{h(k)} \notin T$ }

Quantity(HashTable**)**

“Return the amount of elements in the hash table.”

{ pre: $\exists T$ }
{ post: |k| $\mid |k| \ \varepsilon \ Z+$ }

Test Cases

Stack:

Test case 1:

Stack S = <>

→ push(1)

Stack S = <1>

→ push(2)

Stack S = <2, 1>

→ push(3)

Stack S = <3, 2, 1>

→ push(4)

Stack S = <4, 3, 2, 1>

→ push(5)

S.size = 5
Stack S = <5, 4, 3, 2, 1>

→ pop()

S.size = 4
Stack S = <4, 3, 2, 1>

→ pop()

S.size = 3
Stack S = <3, 2, 1>

→ pop()

S.size = 2
Stack S = < 2, 1>

→ pop()

S.size = 1
Stack S = <1>

→ pop()

S.size = 0
Stack S = <>

Test case 2:

Stack S = <>

→ push(9)

S = <9>

→ push(90)

S = <90, 9>

→ push(2)

S = <2, 90, 9>

→ push(76)

S = <76, 2, 90, 9>

→ push(34)

S.size = 5
S = <34, 76, 2, 90, 9>

→ pop()

S.size = 4
S = <76, 2 , 90, 9>

→ pop()

S.size = 3
S = <2, 90, 9>

→ push(0)

S.size = 4
S = <0, 2, 90, 9>

→ pop()
→ pop()

S.size = 2
S = <90, 9>

→ push(3)
→ push(89)

S.size = 4
S = <89, 3, 90, 9>

→ pop()
→ pop()
→ pop()

S.size = 1
S = <9>

→ pop()

S.size = 0
S = <>

Queue:

Test case 1:

Queue Q = <>

→ enqueue(1)

Q = <1>

→ enqueue(2)

Q = <1, 2>

→ enqueue(3)

Q = <1, 2, 3>

→ enqueue(4)

Q = <1, 2, 3, 4>

→ enqueue(5)

Q.size = 5
Q = <1, 2, 3, 4, 5>

→ dequeue()

Q.size = 4
Q = <2, 3, 4, 5>

→ dequeue()

Q.size = 3
Q = <3, 4, 5>

→ dequeue()

Q.size = 2
Q = <4, 5>

→ dequeue()

Q.size = 1
Q = <5>

→ dequeue()

Q.size = 0
Q = <>

Test case 2:

Queue Q = <>

→ enqueue(9)

Q = <9>

→ enqueue(90)

Q = <9, 90>

→ enqueue(2)

Q = <9, 90, 2>

→ enqueue(76)

Q = <9, 90, 2, 76>

→ enqueue(34)

Q = <9, 90, 2, 76, 34>

→ enqueue(1)

Q.size = 6
Q = <9, 90, 2, 76, 34, 1>

→ dequeue()

Q.size = 5
Q = <90, 2, 76, 34, 1>

→ dequeue()

Q.size = 4
Q = <2, 76, 34, 1>

→ enqueue(4)
→ enqueue(20)

Q.size = 6
Q = <2, 76, 34, 1, 4, 20>

→ dequeue()

→ dequeue()
→ dequeue()

Q.size = 3
Q = <1, 4, 20>

→ dequeue()

Q.size = 2
Q = <4, 20>

→ enqueue(1)

Q.size = 3
Q = <4, 20, 1>

→ dequeue()
→ dequeue()

Q.size = 1
Q = <1>

→ dequeue()

Q.size = 0
Q = <>

HashTable:

Test Case 1 - hash:

"111" →'1' - '1' - '1' → 49 - 49- 49
Radix-3 → $(49 \times 3^{0}) + (49 \times 3^{1}) + (49 \times 3^{2}) = (49) + (147) + (441) = 637$
$637 \% 10 \ (size) = 7 \rightarrow$ index = 7

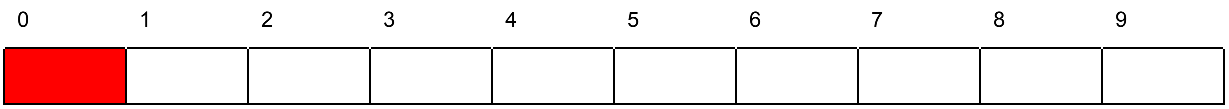
"1111" →'1' - '1' - '1' - '1' → 49 - 49- 49 - 49
Radix-3 → $(49 \times 3^{0}) + (49 \times 3^{1}) + (49 \times 3^{2}) + (49 \times 3^{3}) = (49) + (147) + (441) + (1323) = 1960$
$1960 \% 10 \ (size) = 0 \rightarrow$ index = 0

"~~1~~1111" →~~'1'~~ - '1' - '1' - '1' - '1' → ~~49~~ - 49- 49 - 49 - 49
Radix-3 → $(49 \times 3^{0}) + (49 \times 3^{1}) + (49 \times 3^{2}) + (49 \times 3^{3}) = (49) + (147) + (441) + (1323) = 1960$
$1960 \% 10 \ (size) = 0 \rightarrow$ index = 0

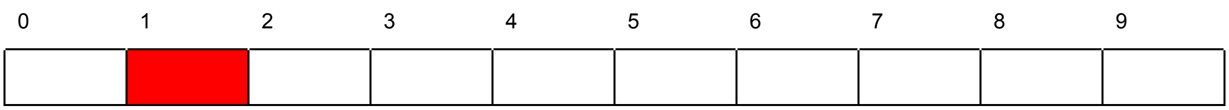
Test Case 2 - quantity:

quantity = 0

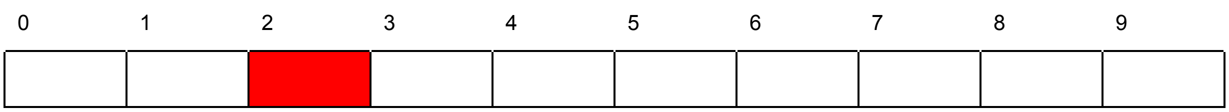
$H =$



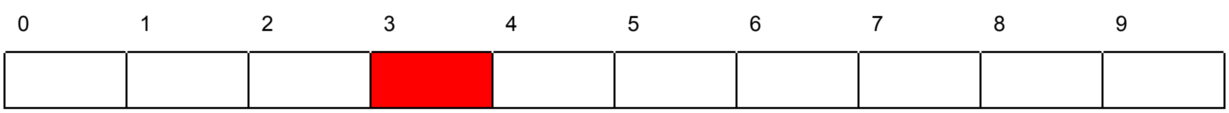
$H =$



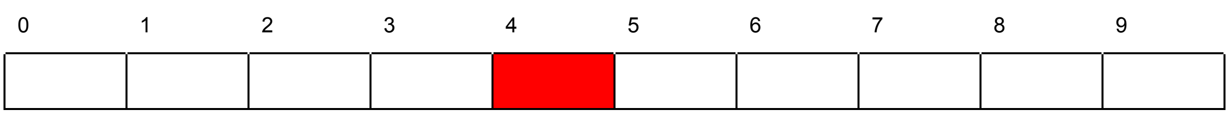
$H =$



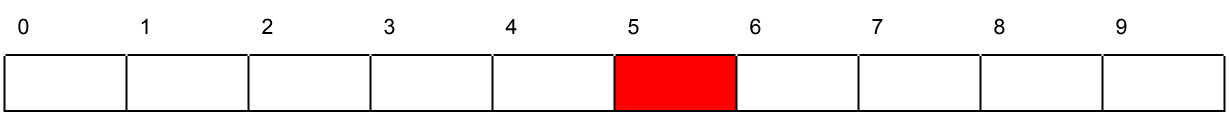
$H =$



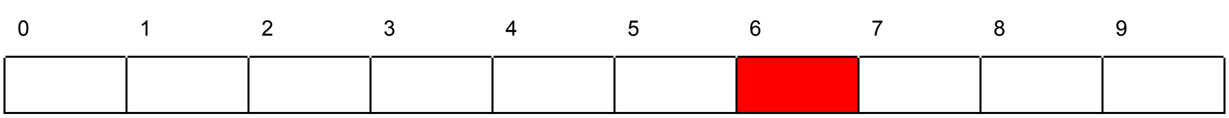
$H =$



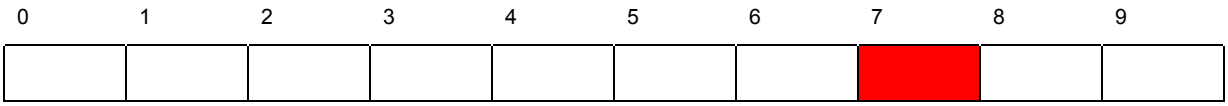
$H =$



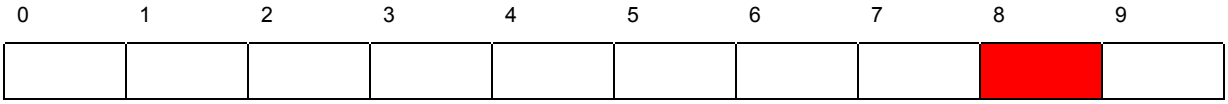
$H =$



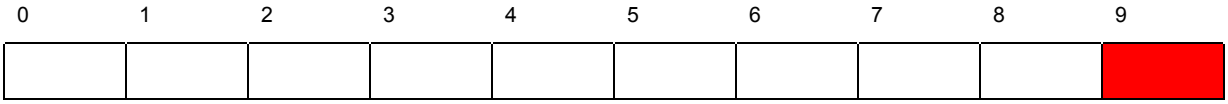
$H =$



H =



H =

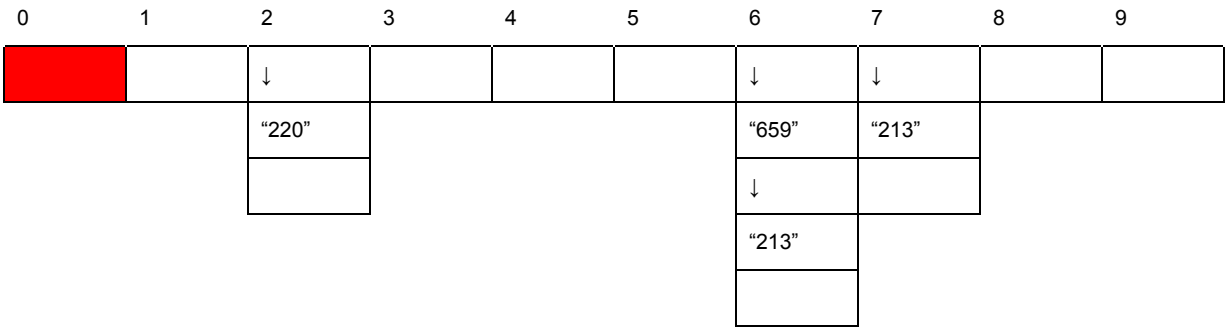


quantity = 0

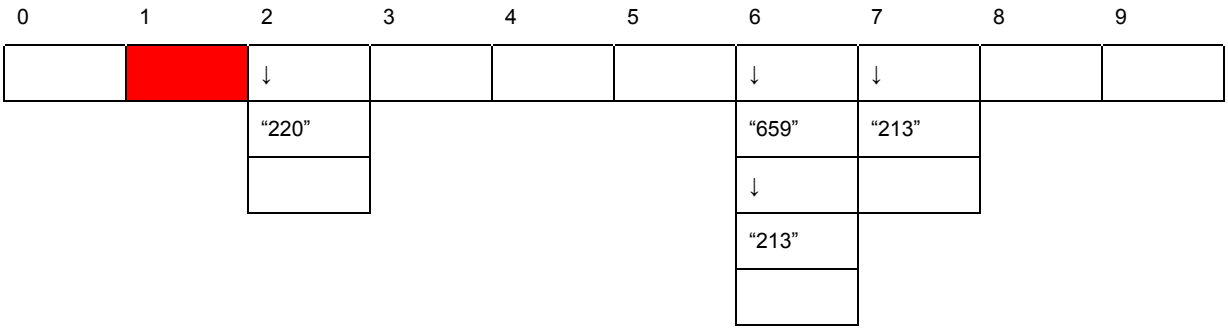
---

quantity = 0

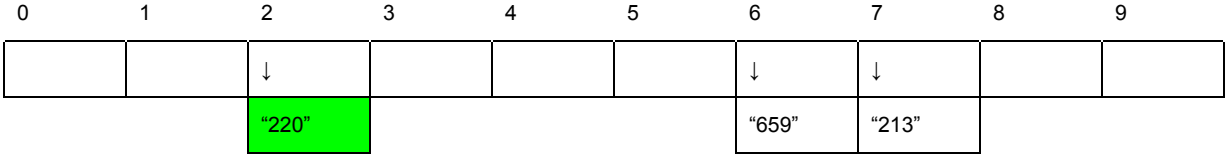
H =



H =

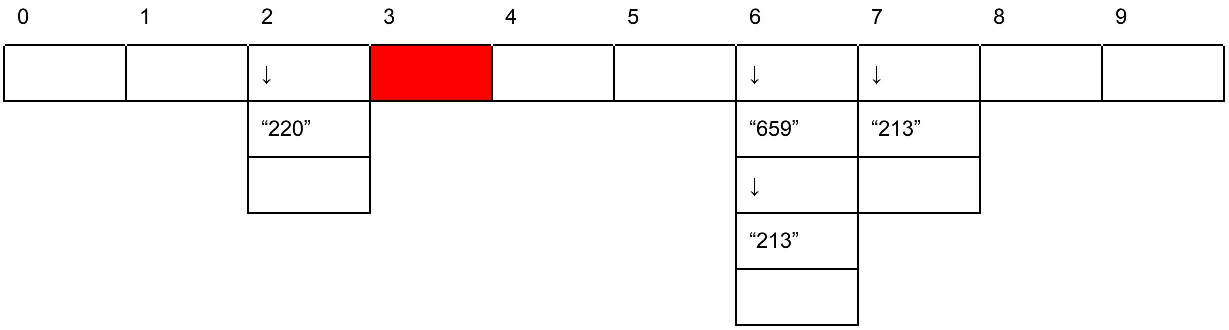


H =

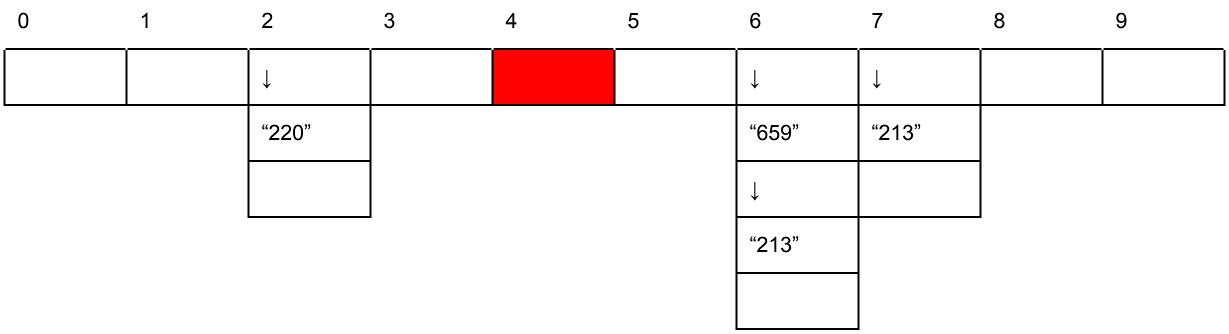


quantity++

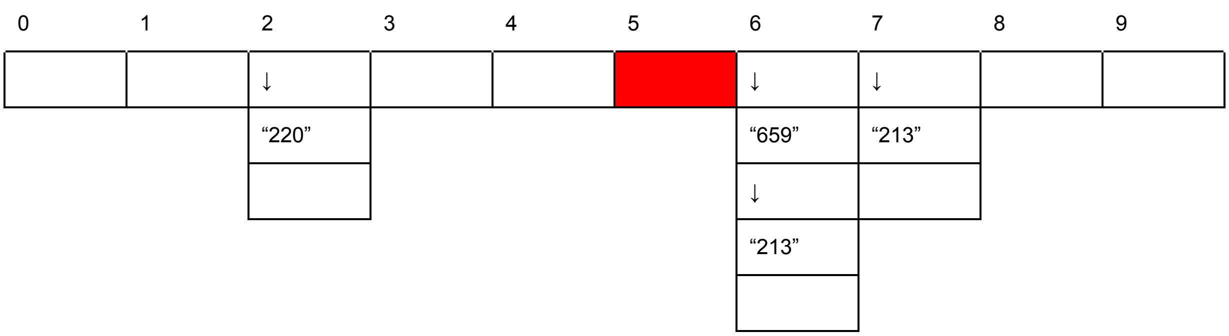
H =



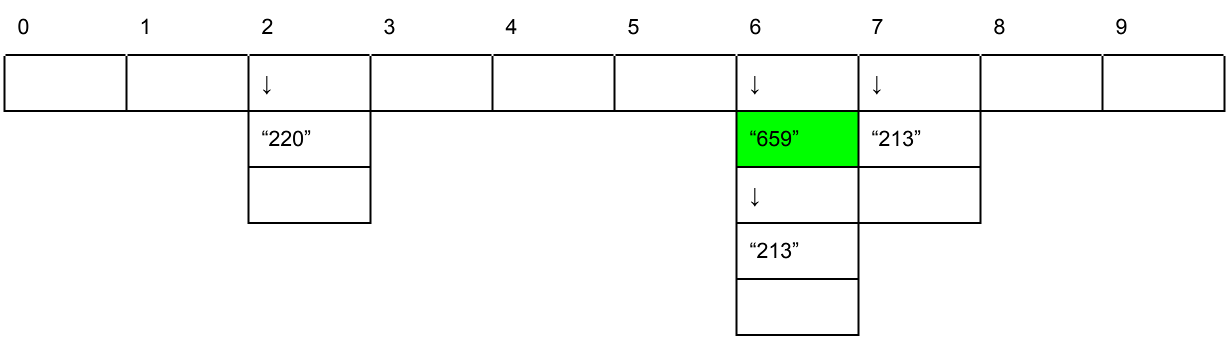
H =



H =

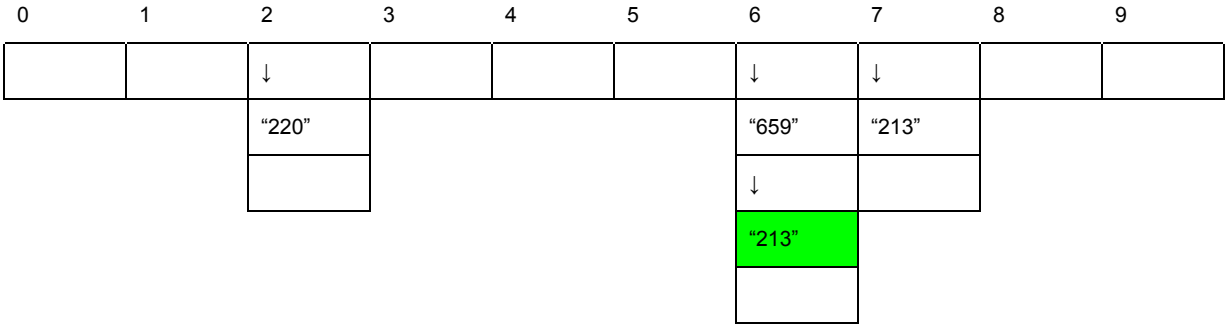


H =



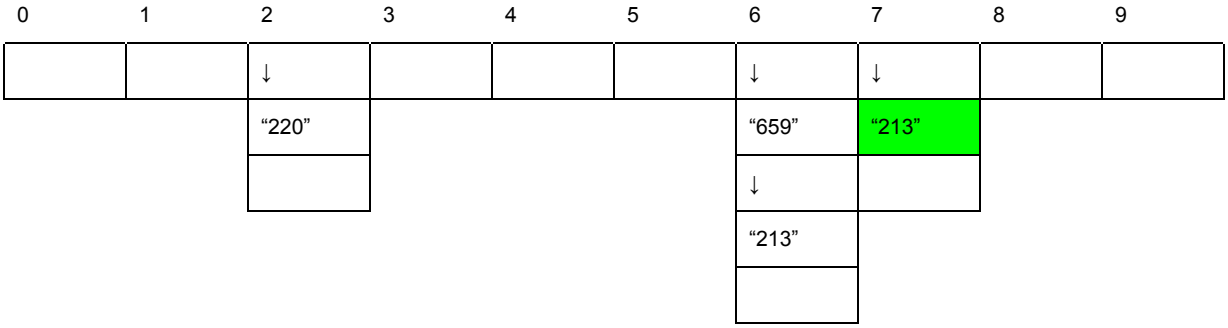
quantity++

H =



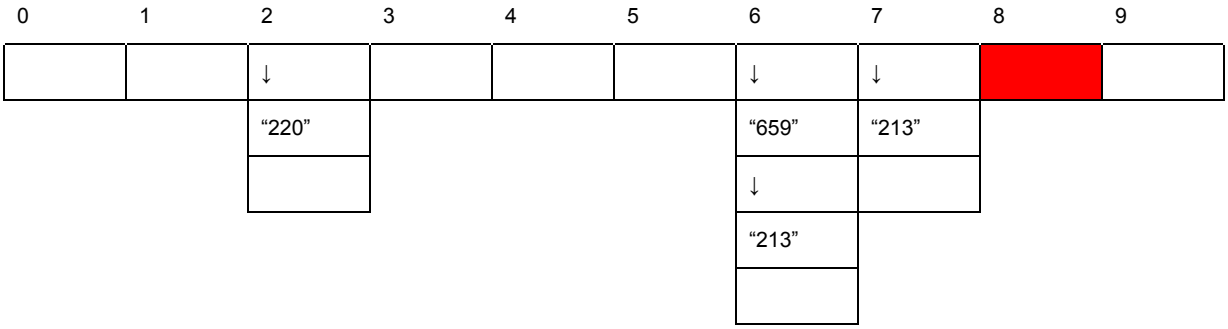
quantity++

H =

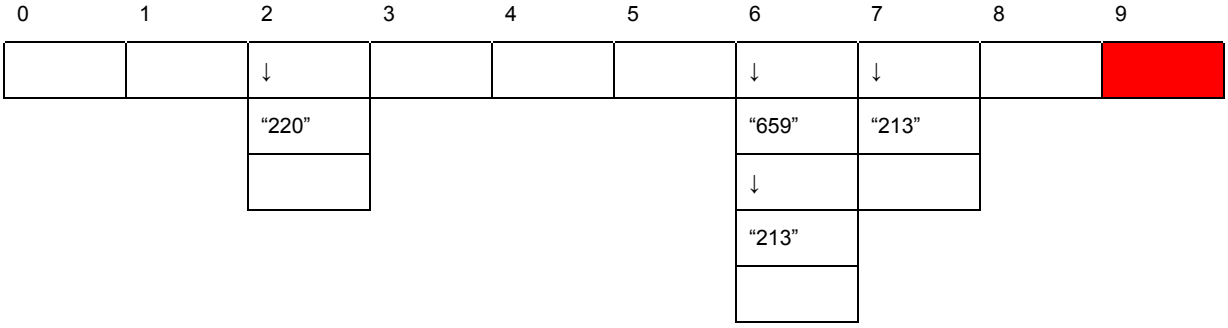


quantity++

H =



H =

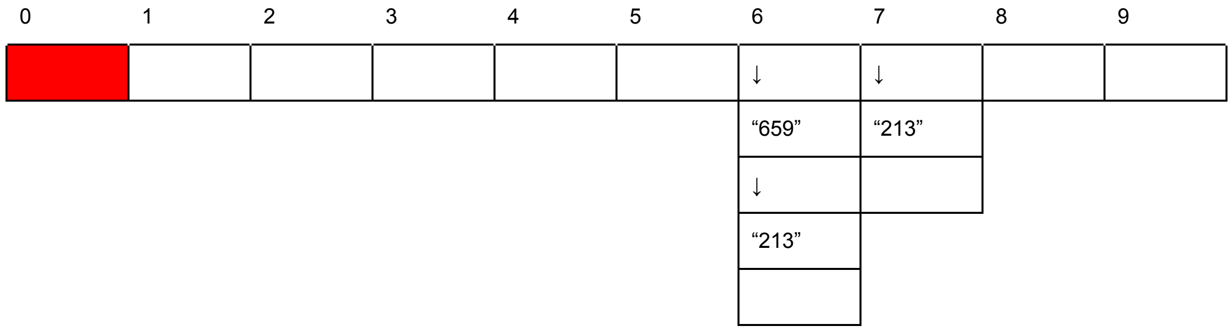


quantity = 4

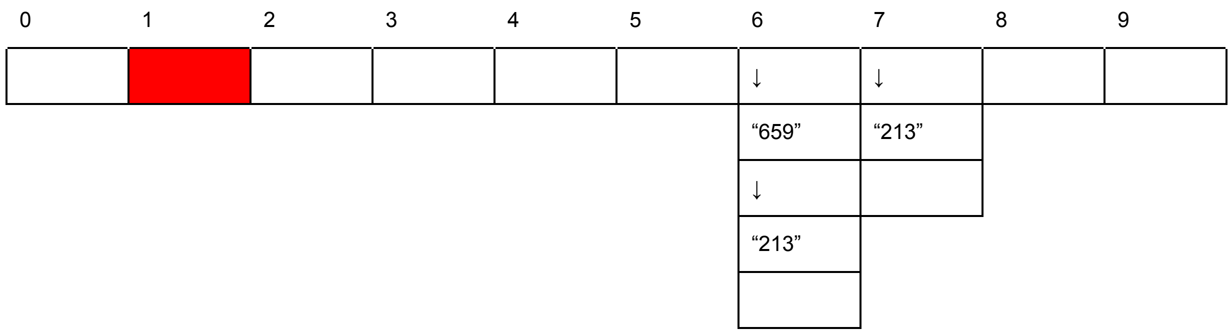
---

quantity = 0

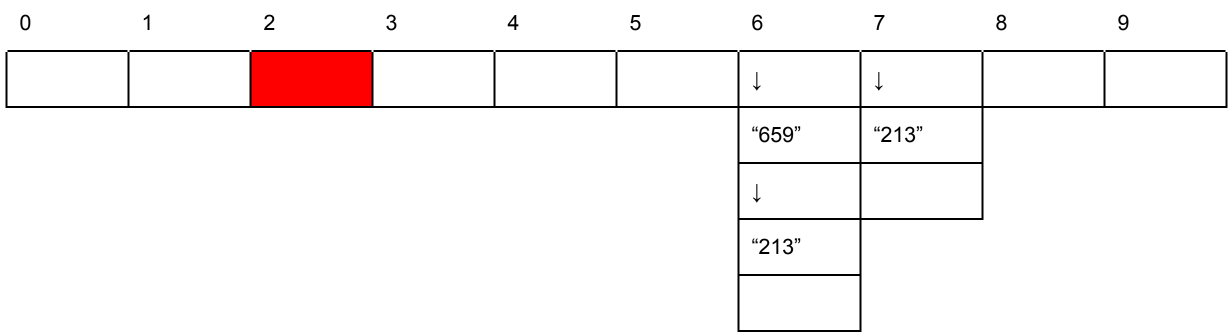
H =


0 1 2 3 4 5 6 7 8 9
“659”
“213”
“213”

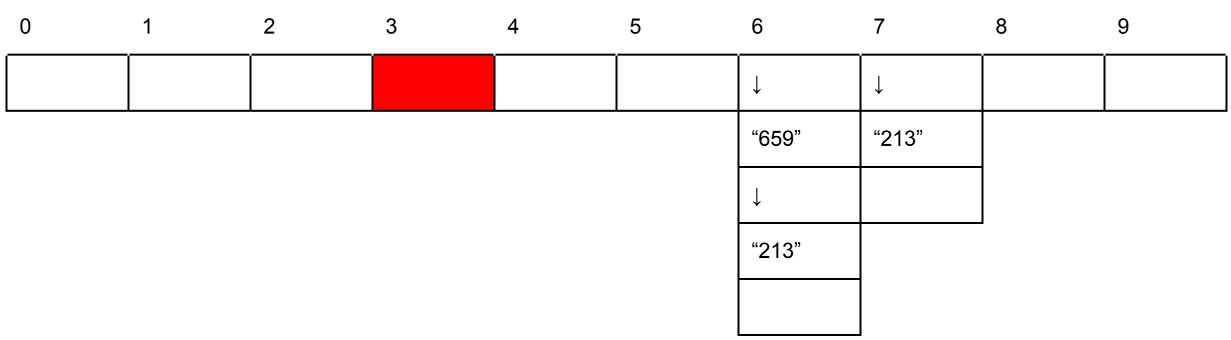
H =


0 1 2 3 4 5 6 7 8 9
“659”
“213”
“213”

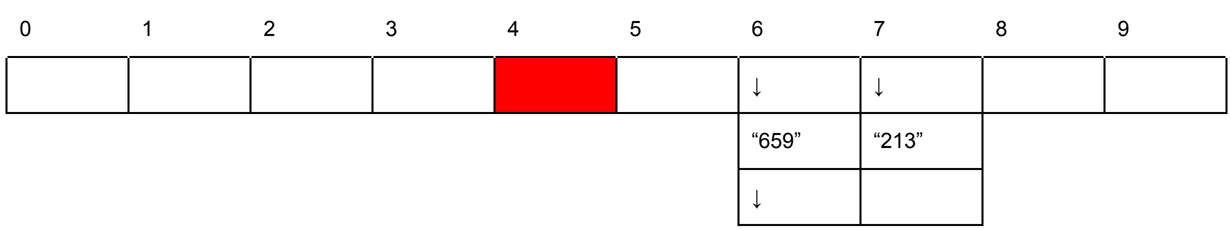
H =


0 1 2 3 4 5 6 7 8 9
“659”
“213”
“213”

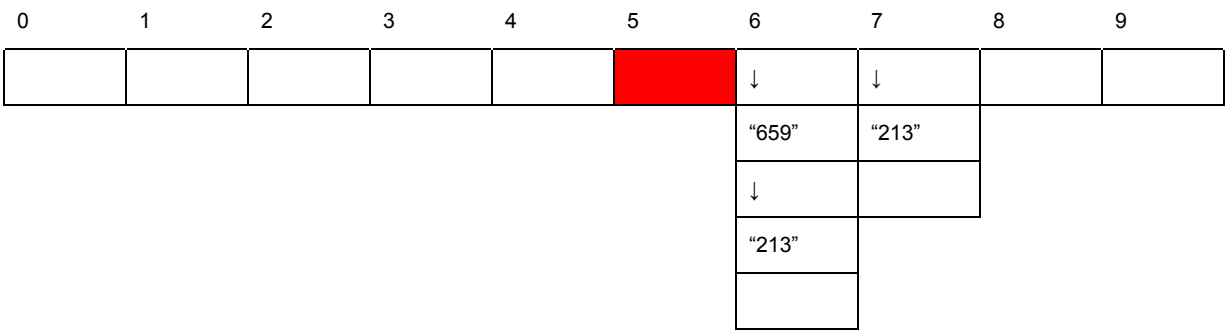
H =


0 1 2 3 4 5 6 7 8 9
“659”
“213”
“213”

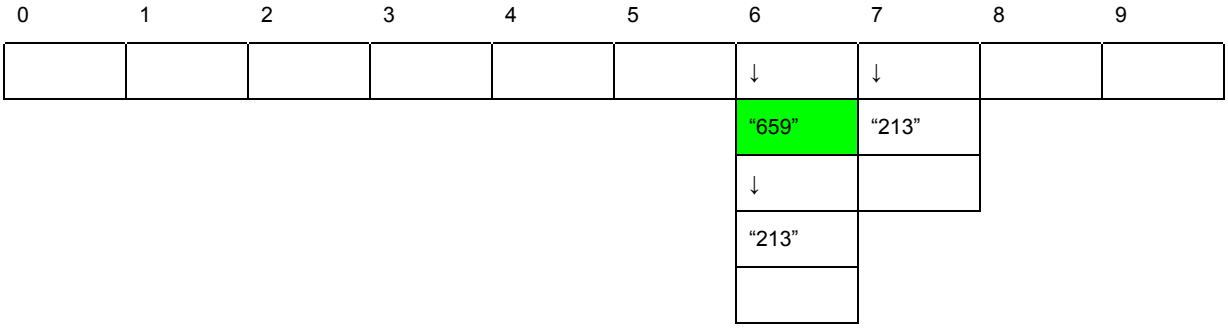
H =


0 1 2 3 4 5 6 7 8 9
“659”
“213”

H =

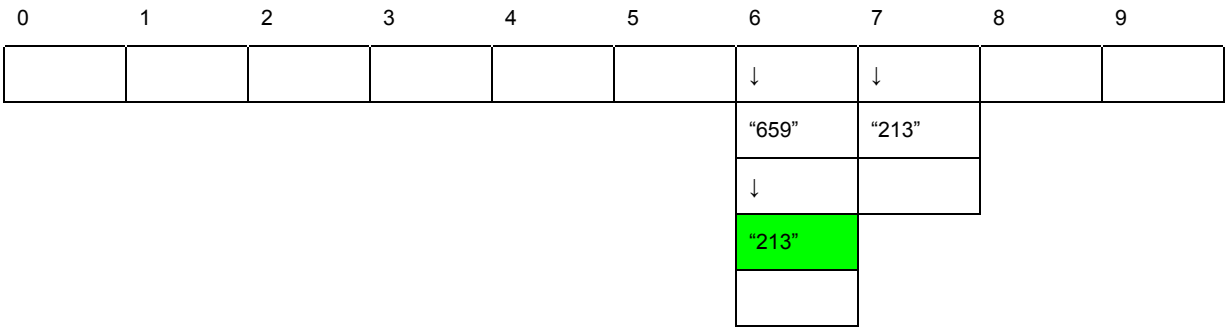


H =



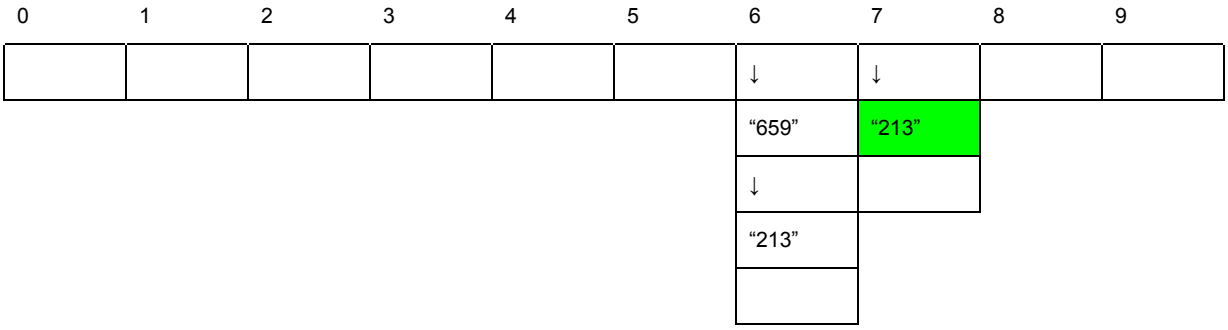
quantity++

H =



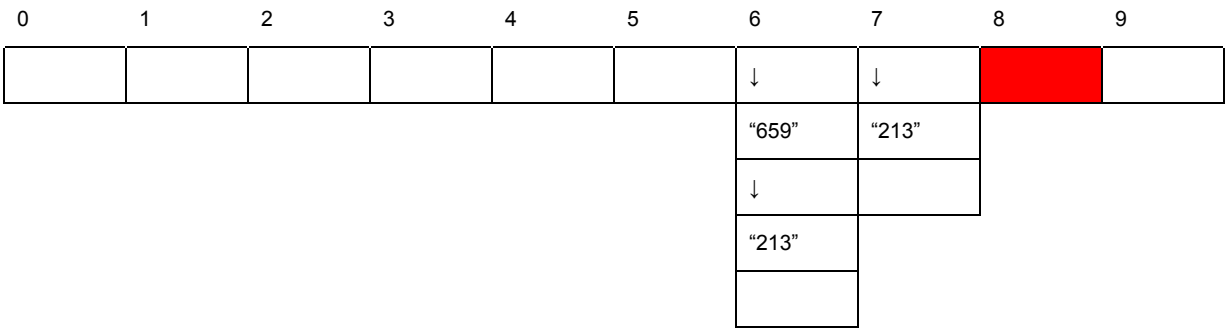
quantity++

H =

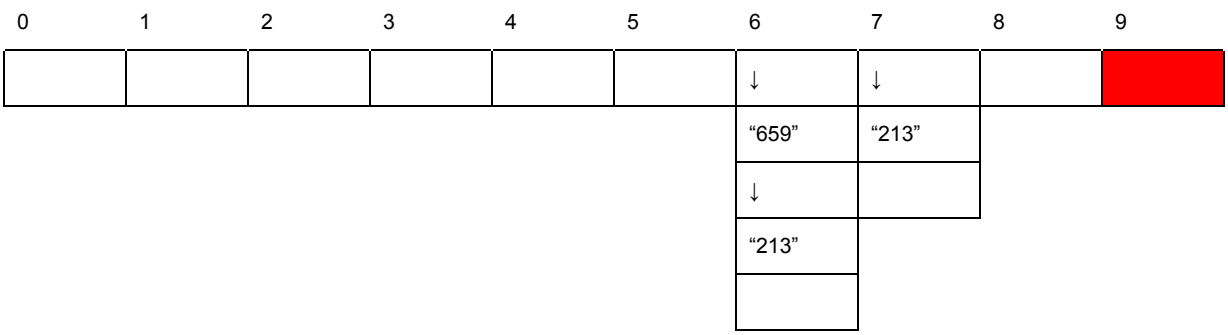


quantity++

H =



H =



quantity = 3

Class	Method	Scene	Input	Result
HashTable	hash	An empty hash table with size 10.	Do the hash function to elements with keys “111”, “1111” y “11111”.	The index of the elements is 7, 0 and 0 respectively.
HashTable	add	An empty hash table with size 10.	Add the elements with keys “220”, “213”, “326”, “659” and “220”.	They are in the correct position.The last element can not be added.
HashTable	get	A hash table with elements “220”, “213”, “326” and “659” with size 10.	Get the elements with keys “220”, “213”, “326”, “659” and “793”.	It returns all the elements except the last one.
HashTable	delete	A hash table with elements “220”, “213”,	Delete the elements with keys “220”,	The elements are not in the hash table and

		“326” and “659” with size 10.	“659” and “793”.	the inner elements are in the right position. It is impossible to delete the last element.
HashTable	quantity	An empty hash table with size 10.		There are 0 elements in the hash table.
		A hash table with elements “220”, “213”, “326” and “659” with size 10.		There are 4 elements in the hash table. After element “220” was deleted the quantity is 3.

Class	**Method**	**Scene**	**Input**	**Result**
Stack	pop	A stack with elements from 1 to 5. The 5 is on top.	Pop the stack 6 times.	It return the elements in the exact order until the 5 pop. It can not pop an empty stack on the 6 pop.
Stack	push	A empty stack.	Push 2, 5, 6, 7 and 3.	The stack have 5 elements and they are in order
Stack	size	A empty stack.		The size is 0.
		A stack with elements from 1 to 5. The 5 is on top.		The size is 5.
Stack	peek	A stack with elements 2, 5, 6, 7, 3 in that order.	First, is normal. Then, pop 2 times. Finally, pop 3 times.	First, peeks 3. Then, peeks 6. Finally, can not peek an element.

Stack	size	A empty stack.	First, is normal. Then, push 1 element. Finally, pop all elements.	First, is empty. Then, is not empty. Finally, is empty.

Class	**Method**	**Scene**	**Input**	**Result**
Queue	dequeue	A queue with elements from 1 to 5.	Dequeue 6 times.	It dequeues the elements in the correct order. Can not dequeue the 6 time, because the queue is empty.
Queue	size	A queue with elements from 1 to 5.	Dequeue 5 times.	Size is 5, then 4, then 3, then 2, then 1, then 0.
Queue	enqueue	An empty queue.	Enqueue 6, 5, 6, 4, 1.	The queue now has 5 elements and they are in the correct positions.
Queue	peek	An empty queue.	Enqueue 3, then 9, then dequeue, then dequeue.	First peek 3, then 3, then 9 and finally can not peek at an element, because it is empty.
Queue	isEmpty	An empty queue.	First, it is normal. Then, enqueue an element. Finally, dequeue.	First, it is empty. Then, it is not empty. Finally, it is empty.

Class	**Method**	**Scene**	**Input**	**Result**

Race	addCompetitor	An empty race.	Add the competitors “Esteban”, “Johan”, “Mateo”, “Juan”, “David”, “Samuel”, “Isabella”, “Diana”, “Fredy”, “Alberto” and “Jose” in the respective order.	They were added in the expected order. “Jose” can not be added.
Race	addUser	An empty race.	Add the users with id “1000”, “1001”, “1002”, “1003” and “1000”.	They are in the correct position.The last user can not be added.
Race	raceSimulator	A race with competitors “Esteban”, “Johan”, “Mateo”, “Juan”, “David”, “Samuel”, “Isabella”, “Diana”, “Fredy”, “Alberto” and “Jose”, and users with id “1000”, “1001”, “1002”, “1003” and “1000”. Johan is the fastest and Alberto is the lowest.		The winner is Johan and they are ordered now by their positions.
		An empty race.		It can not start.
Race	searchUser	An empty race.	Search the user with id “1000”.	Can not return an element that do not exist.
		A race with users with id	Search the user with id “1000”.	Return the user with the id.

		“1000”, “1001”, “1002”, “1003” and “1000”.		
Race	getPodium	A race with competitors “Esteban”, “Johan”, “Mateo”, “Juan”, “David”, “Samuel”, “Isabella”, “Diana”, “Fredy”, “Alberto” and “Jose”, and users with id “1000”, “1001”, “1002”, “1003” and “1000”. Johan is the fastest , then Esteban and then Mateo.	Race starts.	First place, Johan. Second place, Esteban. Third place, Mateo.

Class	**Method**	**Scene**	**Input**	**Result**
Competitor	randomSpeed		Get a random speed.	The speed is random and is between 80 and 100.

Github:

https://github.com/Esarac/Hippodrome